Algorithms for Profiling and Representing Programs with Applications to Speculative Optimizations

A Thesis
Submitted For the Degree of
Doctor of Philosophy
in the Faculty of Engineering

by
Subhajit Roy

Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JUNE 2010
Abstract

Speculative optimizations are now acknowledged as a prime gear wheel in the modern compilation machinery. Considerably more aggressive than their traditional “safe” counterparts, they are biased towards optimizing the *frequently-executed* program paths, even with detrimental effects on the remaining paths; any such penalty is easily counterweighted by the sheer execution frequency of the optimized paths.

A *speculative* optimizer requires a good indication of a program’s run-time behaviour, represented in convenient data-structures, for employment in the analysis and transformation phases. Acyclic path profiles — execution frequencies of “short” paths that terminate at backedges — is the most popular control-flow indicator for the current compilers. These profiles, however, miss control-flow information across loop iterations; many optimizations have been found to be more effective with this information available. Also, in our present compilation infrastructures, the static program representation and the dynamic control-flow information are generally maintained in isolated data-structures, making it cumbersome for speculative optimizers, needing to work with two watertight data-structures.

We propose a richer control-flow profiling entity — the *k-iteration paths* — to capture a program’s execution pattern more precisely. The k-iteration paths are “longer” paths spanning across at most k iterations of a loop. Being overlapping profiles, these k-iteration profiles are strictly more informative than the acyclic path profile on a loop unrolled (k-1) times. Essentially via a generalization of the Ball-Larus acyclic path-profiling algorithm, we show that it is possible to number these multi-iteration paths *perfectly*, allowing for an efficient k-iteration profiling algorithm. Experimental results show that k-iteration profiling is realistic.

We, then, saunter into the area of program representation, presenting an effective way of amalgamating profile information with the static program representation in a novel form — the *Hot Path SSA Form (HPSSA)* form — to be savoured as a single unit by speculative optimizers. An Static Single Assignment (SSA) form for the speculative domain, the HPSSA form encourages the development of new, efficient path-profile guided speculative optimizations. Via the design of a novel extension of the Wegman and Zadeck’s Sparse Conditional Constant Propagation (SCC) algorithm — the *Speculative Sparse Conditional Constant Propagation (SSCC)* — we demonstrate that, befriended by the HPSSA
form, many existing SSA-based traditional optimizations can encourage a corresponding ally in the speculative orbit.
Acknowledgements

First and foremost, I am deeply indebted to my research supervisor, Prof. Y. N. Srikant, for being my “friend, philosopher and guide” — literally. A great motivator, he has often pulled me up from the deepest trenches of frustration and egged me ahead. I have freely sought his wise advise on matters even other than academics. His comments on my research has frequently opened up a fresh line of thought, and I have learnt quite a few things from him on presenting my work. From the deepest recesses of my heart, I express my sincere gratitude to him.

I thank Philips Research India for their generous doctoral fellowship.

The Compiler Laboratory has been a land where I have been able to reap multiple fervorous friendships. It has been a fertile playground that has been the centre of feverish discussions on subjects from computer architecture, compilers, machine learning, formal methods, and algorithms — to more colorful areas like cricket and movies. I extend my earnest appreciation to all my “lab” friends.

I am also grateful to the office staff of the department for always being there for us. Mrs. Lalitha and Mrs. Suguna have always answered promptly to my uncountable requests and queries.

I, still, delightfully savor the cherished memories of the charged up table tennis games at the gymkhana which shaped many a firm friendships. These friends not only helped me "switch-off" and relax, but also enriched me with with their extensive knowledge and experience in their areas of specialization.

And most importantly, I thank my family who have always stood behind me, supporting and motivating me all through the years of my life.
Contents

Abstract i

Acknowledgements iii

1 Introduction 1
  1.1 Background ......................................................... 6
    1.1.1 Control-flow Profiling ....................................... 6
    1.1.2 Intermediate Representations ............................... 8
    1.1.3 Speculative Optimizations .................................. 10
  1.2 Our Contributions .................................................. 14
    1.2.1 The k-Iteration profiling algorithm ........................ 14
    1.2.2 The Hot Path SSA form ...................................... 15
  1.3 Organization of the thesis ......................................... 15

2 Profiling k-Iteration Paths 17
  2.1 The k-Iteration Paths: A Novel Control-Flow Profiling Entity .... 18
  2.2 Path Numbering .................................................... 20
    2.2.1 The loop-unrolled DAG ..................................... 21
    2.2.2 The Path Numbering Algorithm .............................. 22
      2.2.2.1 Numbering all the k-iteration static paths .......... 22
      2.2.2.2 Eliminating invalid k-iteration paths ................ 24
    2.2.3 The Path Identification Algorithm ........................... 26
    2.2.4 Correctness Results ......................................... 28
  2.3 Program Instrumentation ........................................... 31
    2.3.1 Removal of the dummy edges ................................ 31
    2.3.2 The Instrumentation Algorithm .............................. 32
  2.4 Mixed-Mode Profiling: Profiling different regions with varied length paths .... 35
  2.5 Experimental Results ............................................. 37
3 The Hot Path SSA Form

3.1 Introduction .................................................................................. 40
3.2 Preliminaries .................................................................................. 41
  3.2.1 Thermal States of Program Entities .......................................... 41
  3.2.2 The structure of profiled acyclic paths ...................................... 44
3.3 The Hot Path SSA (HPSSA) form .................................................. 45
3.4 Constructing the HPSSA Form ....................................................... 46
  3.4.1 Thermal Frontiers: Placing \(\tau\)-functions ................................. 47
    3.4.1.1 Caloric Connector ........................................................... 50
    3.4.1.2 Computing Thermal Frontiers ....................................... 52
  3.4.2 Allocating \(\tau\)-function arguments ......................................... 53
    3.4.2.1 Understanding pseudo definitions .................................. 53
    3.4.2.2 The \(\tau\)-function Argument Allocation Algorithm ............. 54
3.5 Exiting the HPSSA form ................................................................. 61
3.6 Speculative Sparse Conditional Constant Propagation ................. 61
3.7 Implementation and Experiments ................................................ 64
3.8 The \(\tau\)-function as a generalized \(\phi\)-function ............................ 65
3.9 Conclusions .................................................................................... 66

4 Related Work .................................................................................... 67
  4.1 Control-Flow Profiling ................................................................. 67
  4.2 The SSA Form: Extensions and Applications ............................... 69
  4.3 Profile-Guided Analysis and Optimizations .................................. 71

5 Conclusions ...................................................................................... 74

Bibliography ....................................................................................... 77
## List of Figures

1.1 Constant Propagation. ................................................................. 1
1.2 Infeasible Paths that an optimizer might not be able to prune. .................. 2
1.3 *Speculative* Constant Propagation. ........................................... 3
1.4 Modern Compilation Machinery. ................................................... 4
1.5 Our contributions. ................................................................. 5
1.6 Numbering static paths in a DAG [1]. ........................................... 7
1.7 An example control-flow graph. .................................................. 8
1.8 The Ball-Larus Path Profiling Algorithm [1]. .................................... 8
1.9 The Static Single Assignment Form. ............................................. 10
1.10 The variable renaming phase of the SSA construction algorithm [2]. ............ 11
1.11 Control speculation. ............................................................. 12
1.12 Data speculation. ................................................................. 12
1.13 Motivation for k-iteration path profiling. ......................................... 15
1.14 The Hot Path SSA (HPSSA) Form. ............................................. 16

2.1 An optimization enabled by k-Iteration Profiling. ............................... 17
2.2 Our running example. ............................................................. 19
2.3 Acyclic and Two-Iteration Static Paths. ......................................... 20
2.4 Two-iteration Loop-Unrolled DAG. ............................................ 22
2.5 Edge-weights for tracing two-iteration paths. ..................................... 22
2.6 Removing dummy edges. .......................................................... 34
2.7 The instrumented CFG for two-iteration path profiling. ......................... 34
2.8 Mixed-mode profiling. .......................................................... 36

3.1 The Static Single Assignment Form. ............................................. 42
3.2 The Hot Path SSA (HPSSA) Form. ............................................. 43
3.3 The HPSSA form of a program with loops. ..................................... 44
3.4 Constructing the HPSSA Form. ................................................ 48
3.5 Materialization of cold paths. .............................................. 52
3.6 Violation of the second condition in Lemma 3.6. ..................... 53
3.7 The dominator tree of the CFG in Figure 3.1. .......................... 59
3.8 Steps in the execution of the $\tau$-argument allocation algorithm. 60
3.9 The SCC Lattice. ............................................................. 62
3.10 A case that requires meet with its old value in the $\tau$-statement transfer function for SCC. 63
3.11 Cases in the HPSSA form that need special handling. ............... 63
3.12 The Generalized HPSSA Form. .......................................... 66

4.1 Overlapping Paths (OL) for estimating longer paths spanning loop iterations [3]. 68
4.2 The $\chi$ and $\mu$ functions in the Hashed SSA Form [4]. ................ 70
4.3 The $\psi$-SSA form [5]. ....................................................... 71
4.4 Partial Redundancy Elimination [6]. ...................................... 73
4.5 Speculative Partial Redundancy Elimination [7, 8, 9]. .................. 73

5.1 The Hot Path SSA form on multi-iteration profiles. ................. 75
List of Tables

2.1 An execution trace of the path-numbering algorithm. .......................... 26
2.2 Execution trace of the instrumented code. ........................................... 35
2.3 Profiling cost for k-Iteration Profiling. ............................................. 38
2.4 Growth in the number of static paths with increasing values of k. .......... 38

3.1 Speculative constants discovered by the SSCC algorithm. ...................... 65
List of Algorithms

2.1  The k-Iteration Profiling Algorithm: Path Numbering. .......................... 23
2.2  The k-Iteration Profiling Algorithm: Path Identification. ......................... 27
2.3  The k-Iteration Profiling Algorithm: Program Instrumentation. .................. 33
3.1  Constructing the HPSSA Form: Computing the set of Caloric Connectors. .... 52
3.2  Constructing the HPSSA Form: Inserting τ-statements. ............................. 53
3.3  Constructing the HPSSA Form: Allocating arguments to τ-functions. ............ 56
Chapter 1

Introduction

You bump into the coffee table (late for office, again!), crash the vase, trip on the doormat, stumble out of the front-door — to be greeted by a cloudy morning sky. Is it going to rain today? Nah! You don’t savor the big black umbrella trudging along; it devastates your style statement, and it slows you down on the sidewalk. Picture yourself losing the umbrella: a stylish figure meandering swiftly though the maze of human forms — clearly a winning decision, if it does not rain. Otherwise, a rain-drenched creature will have to double back to pick the big black unstylish canopy. With the umbrella, though stodgy and slower, you are safe. To drop the umbrella, you need to speculate on the chances of rain — and you better be right!

Any modern compiler applies a plethora of optimizations to improve the run-time performance of a program — and, traditionally, these optimizations have been necessarily safe. Let us consider

![Diagram](image_url)

**Figure 1.1:** Constant Propagation.
Figure 1.2: Infeasible Paths that an optimizer might not be able to prune.

an extremely potent optimization: constant propagation. This optimization attempts to ascertain at compile-time, as to what expressions in the program will always evaluate to the same value at run-time. For example, in Figure 1.1, irrespective of the program inputs, the expression in the basic-block 6 will always evaluate to the same constant value. The compiler, thus, replaces the evaluation of the expression by a cheaper operation of loading of the constant value (5) into the variable $z$. This transformation is safe, as, applying the transformation does not alter the results of the computation in the live$^1$ variables along any static program path from the program’s start node to the exit node. However, in the basic-block 7, the compiler restrains from applying any such transformation: if the statement $z = x * y$ is replaced by $x = 6$, though the computation along the path 1-3-5-7 holds, that along 1-3-4-7 gets disturbed; hence, this transformation is unsafe.

Though extremely important to a program’s runtime performance, such safe optimizations tend to be conservative: if the optimizer is not “sure” about the safety of a transformation, it votes against applying it. Consider the example in Figure 1.2(a): although the contents of basic-block 3 always evaluates to true (where 3 → 4 is the true edge), thus making the expression $x*y$ in basic-block 7 a compile-time constant, an optimizer that is unaware of the Fermat’s Little Theorem$^2$ would fail to detect so. Inability of an optimizer at pruning such infeasible paths might also be due to such conservative assumptions on the part of client analyses; for example, in Figure 1.2(b), the constant-propagation algorithm would depend on the results of pointer analysis to infer infeasible paths.

Furthermore, not all paths in a program are executed with the same frequency: some paths — a

---

$^1$A variable is live at a program point if it has a use along any path from that point.

$^2$If $p$ is a prime and $a$ an integer, then $a^p - a \equiv (\mod p)$. 
very small fraction of all possible static paths — engross a large fraction of the total compute cycles. Traditional safe optimizations blatantly disregard this well-known demeanor of executing programs.

A new avatar of optimizations, Speculative Optimizations, are becoming increasing popular due to their adroitness at exploiting the above program behaviour. In contrast to their “safe” counterparts, which attempt to benefit all program paths that they reason feasible, the speculative optimizations embrace speculative decisions to favor the frequently-executed hot paths. Such speculative decisions (analogous to the speculative decision of pushing aside the umbrella on a cloudy day) do incur a penalty whenever the speculation fails. But, even with detrimental effects on the rarely executed paths (due to failed speculations), the benefits on the hot paths — by the sheer number of times the optimized paths are taken — heavily outweighs the slowdown on the remaining paths. Much more aggressive than their cautious cousins in the safe domain, speculative optimizations are now accredited as primary players in the arena of modern compilation.

Let us study the transformation illustrated in Figure 1.3: if only the paths 1–2–6, 1–3–4–6 and 1–3–5–7 were found to be hot, the speculative optimizer could infer that most of the time, the result of the expression $x \times y$ in basic-block 7 would yield a value 10. This might tempt the optimizer to directly load 10 into the variable $z$ (in basic-block 7) — a speculative transformation that, though “technically incorrect”, mostly stands true. This transformation is clearly unsafe: whenever the program saunters along the rarely executed, but feasible path, 1–3–4–7, the value in the variable $z$ would tumble incorrect. Speculative optimizers insert additional instructions to detect when such speculations go awry, and provide appropriate mechanisms to recover from them. Modern architectures, acknowledging the potency of speculation, are beginning to cater relevant support; for example, the IA64 instruction
set [10, 11] provides ld.s and ch.s instructions: the ld.s instruction performs a load speculatively, allowing a chk.s instruction to catch and recover from failed speculation.

To understand how speculative optimizers position themselves in the compilation machinery, consider the simplified schematic in Figure 1.4. After the source program passes through lexical analysis, syntax analysis and semantic analysis, it is translated to an intermediate representation (IR), which is fed into the optimizer. After sprinting through a chain of analysis and transformations in the optimizer, the program, finally, arrives at the Machine Code Generator to be cast into object code. Modern compilers, however, typically provide multiple intermediate representations: the source program is translated into object code in multiple steps — the IR successively “lowered” during each translation. If profile-guided speculative optimizations are supported, the compiler also uses a profiler — which is fed with the program and a representative set of inputs — to understand the program’s run-time demeanor.

A Speculative optimizer needs a good indication of a program’s run-time behaviour, represented in convenient data-structures, for use by its analysis and transformation phases. In this thesis, we contribute to the speculative optimization machinery in the following areas:

- **Control-flow profiling**: Currently, the acyclic path [1] is the most popular control-flow profiling entity employed to capture a program’s control-flow behaviour. The acyclic path profiles have
proved to be a successful abstraction of the program trace, with a horde of optimizations designed around them. However, such profiles fail to capture the hot correlations within loop iterations, inhibiting the development of more powerful speculative optimizations. We propose a novel control-flow profiling entity — the \textit{k-iteration path} — that captures longer paths spanning multiple loop-iterations. Figure~1.5(a) shows that the k-iteration path profiling algorithm sits in the “profiler” component of the compilation system.

- \textbf{Intermediate representation for the design of efficient speculative optimization algorithms}: The safe domain of compiler optimizations are fortunate to relish an extremely powerful representation, the \textit{Static Single Assignment (SSA) Form} \cite{Dybvig1991}, that has facilitated the design of efficient algorithms of known optimizations, and enabled the design of new ones. Unfortunately, such an intermediate representation had been absent in the speculative domain for enabling path-profile guided optimizations. We propose a novel intermediate representation — the \textit{Hot Path SSA Form (HPSSA)} — that fills this void. The HPSSA form relieves the optimizer from requiring to juggle with the static program representation (in the form of the intermediate representation) on one hand, and the program behavioral statistics (in the form of frequently executed program paths)
on the other. With the HPSSA form, the optimizers can savour an intermediate representation that has the information about hot paths “weaved” in. Figure 1.5(b) shows how the compilation flow changes with the HPSSA form.

1.1 Background

This section provides necessary background, laying a foundation for the following chapters.

1.1.1 Control-flow Profiling

Profiling is an indispensable tool for both compiler writers and computer architects, assisting them decipher an executing program’s run-time demeanor. From simple node and edge profiles to whole program paths [12], control-flow profiles span a whole range. The node/edge profiles, though comparatively inexpensive to collect, are not potent enough; and, the whole program paths are expensive to amass.

An efficient algorithm for acyclic path profiling — profiling paths that either terminate at backedges or at procedure exits — was proposed by Ball and Larus [1]. The key idea of the algorithm was to assign unique path-identifiers to all acyclic paths in a fashion, such that, the path identifiers of the traversed paths can be efficiently reconstructed during a program’s profile run. More importantly, the algorithm computes a perfect numbering of these path-identifiers: if there are n static acyclic paths in the program, the paths are assigned identifiers from 0 to n-1.

The Ball-Larus algorithm essentially consists of two steps:

• Path Numbering

The Ball-Larus algorithm assigns weights to the edges in the control-flow graph in such a manner, that, the sum of the edge-weights on each acyclic path is unique — a path-identifier. The Ball-Larus algorithm reduces a graph with loops into a DAG, while retaining all acyclic static paths using a set of dummy edges — edges from a dummy entry node\(^3\) to all targets of backedges and from each source of a backedge to a dummy exit node (see Figure 1.8(a)).

The algorithm, then, traverses the nodes in the DAG in a reverse topological order, assigning weights to each outgoing edge of a node as it is visited. The core idea of the Ball-Larus path-numbering algorithm is shown in Figure 1.6: if all the paths from the nodes a, b and c were uniquely numbered from 0 to \(n_1\), 0 to \(n_2\) and 0 to \(n_3\) respectively, then, assigning a weight of 0 to \(x \rightarrow a\), \(n_1\) to \(x \rightarrow b\) and \(n_1 + n_2\) to \(x \rightarrow c\) will uniquely number each path from \(x\).

\(^3\)Dummy entry (connecting to the root node) and exit nodes (connected from all the procedure return nodes) are added to each graph.
Following the same notations as used in [1], let \texttt{numPaths}(u) be the number of paths from node \( u \), and \texttt{val}(e) be the weight assigned to the edge \( e \). Figure 1.8(a) shows the weights assigned to each edge by the Ball-Larus path numbering algorithm. The algorithm commences from the \texttt{End} node: \texttt{numPaths(End)}=1 for this trivial single-node path. Next, the node 6, which has a single outgoing edge \( 6 \rightarrow \texttt{End} \), is assigned a weight 0 (not shown), while \texttt{numPaths}(6) is assigned 1. The outgoing edges of node 5 are assigned weights 0 and 1 respectively, and \texttt{numPaths}(5) is set as 2. The numbering can be optimized so that the number of edges assigned non-zero weights is minimum (please refer to [1] for details on this optimization scheme).

- **Program Instrumentation**

  This phase adds instrumentation code on the edges of the control-flow graph. The instrumentation code employs a dedicated register to accumulate the respective weight as each edge is traversed during the execution of the instrumented program. Figure 1.8 shows the instrumented program for the example; note how the backedge 5 \( \rightarrow \) 2, which marks the end of an acyclic path, updates the frequency count of the last path into a path-frequency table (which may be implemented as an array \texttt{freq}), and resets the register.

The number of static paths in the flow-graph decides the implementation of the path-frequency table, and hence, also the performance of the profiled program. If the number of the static paths is small, the table can be implemented as a counter array; otherwise, one needs a more expensive structure, like a hashtable.
Figure 1.8: The Ball-Larus Path Profiling Algorithm [1].

1.1.2 Intermediate Representations

As can be seen in Figure 1.4, the source program is generally translated into an intermediate representation before it is cast into the target machine code. Some of the benefits of using such a machine-independent intermediate representation are retargetability (code for a different machine can be produced by plugging in a different backend), and machine-independent code optimization (optimizations
can be applied to the intermediate representation) [13].

Syntax trees (and DAGs), postfix notation, and three-address code are some common intermediate representations used by compilers. A syntax tree (and DAG) represent the structure of the source program in a hierarchical fashion, while the postfix notation is a linearized representation of the same. The three-address code represents the semantics of the program via statements (including control-flow statements), where each statement is allowed at most three addresses. Please refer to [13] for a more detailed discussion on these intermediate representations.

The Static Single Assignment (SSA) form is a relatively recent intermediate form. Program analysis and optimizations have benefited immensely from this intermediate representation for traditional compiler optimizations. The SSA form is based on an extremely simple idea: allow only a single definition of a variable to reach a statement using it, which it achieves by enforcing every definition in the program to possess an unique target. This constraint prunes out false dependencies, and factors long du-chains into a web of short, simple ones. A multitude of optimizations were either made possible, or were heavily empowered by the SSA form — global value numbering, sparse conditional constant propagation and register allocation to name a few.

In the Static Single Assignment (SSA) form, each use of a variable has exactly one reaching definition. This is made possible by a special operator, the \( \phi \)-function, that merges multiple definition from different paths into a single definition. This forces any subsequent use (of a variable) to witness exactly one definition (the newly inserted \( \phi \)-statement). Each argument in the \( \phi \)-function at a basic-block \( u \) corresponds to an incoming edge to \( u \).

Figure 1.9 provides an example of a program in the SSA form. Notice how the definitions for \( x \) at \( b_1 \), \( d_1 \) and \( e_1 \) are “merged” into a single definition at the statement \( f_1 \), thus making \( x_9 \) the only definition reaching the uses \( g_3 \), \( h_3 \) and \( i_1 \). Understandably, the use-definition structure of a program in SSA form is extremely simple, allowing the design of cleaner and faster algorithms.

The (minimal) SSA form construction algorithm [2] essentially comprises of two steps:

1. Insertion of the \( \phi \)-functions: This step inserts \( \phi \)-functions at appropriate points in the program to merge multiple reaching definitions at that point.

2. Variable Renaming: This step renames the variables in the program, such that, the target of each definition has a unique name.

A \( \phi \)-function for a variable \( x \) is inserted at the iterated dominance frontier of any definition to \( x \). A node \( u \) is said to dominate a node \( v \) if all paths from the start node to the node \( v \) must visit the node \( u \).

A node \( u \) is said to strictly dominate \( v \) if \( u \) dominates \( v \) and \( u \neq v \). A node \( u \) is said to be the immediate dominator of \( v \), if, \( u \) strictly dominates \( v \), and \( u \) does not strictly dominate any other node that also strictly dominates \( v \). For the example in Figure 1.9(a), the node \( f \) dominates — in fact, immediately
dominates — the nodes $g$ and $h$. The immediate dominators can be arranged in a tree (referred to as the dominator tree), where the children of a node are the nodes that it immediately dominates. A node $v$ is said to be in the dominance frontier of another node $u$ if, and only if, $u$ does not dominate $v$ though a predecessor of $v$ is dominated by $u$.

The variable renaming phase processes the basic-blocks in their depth-first order in the dominator tree. It uses a variable renaming stack to keep track of the reaching definitions to the current node; each definition causes the newly defined version to be pushed onto the stack. At the end of a basic-block $u$, the algorithm “peeks” into the $\phi$-functions of its successors $v \in \text{succ}(u)$ in the control-flow graph (CFG), and renames the argument corresponding to the respective outgoing edge $u \rightarrow v$. Figure 1.10 exemplifies the renaming algorithm.

### 1.1.3 Speculative Optimizations

Optimization opportunities in programs is limited by control-flow branches and indirect memory references. Consider Figure 1.11(a): the statement “$x = *p + a$” needs to be executed only if the predicate ($c = 0$) evaluates as true. Similarly, in Figure 1.12(a), the first operand (pointer dereference $*p$) of statement 4 can use the same value as found for the first operand of statement 2 — but only if $p$ and $q$ are not aliased.

Speculation is a potent instrument at surmounting the above barriers, thus conducing more powerful optimizations.
(a) Let us assume that at the entry to the basic-block $a$, the Variable Renaming Stack had some configuration.

(b) On encountering a definition of the base variable $x$, the target of the assignment is assigned a fresh name (say $x_2$); this new version is pushed in the Variable Renaming Stack.

(c) When a use of the base variable $x$ is encountered, the topmost entry in the stack is used to rename the used instance.

(d) On reaching the end of the basic-block $a$, the algorithm "peeks" into the $\phi$-functions of the successor(s) of $a$, and renames the argument corresponding to the respective outgoing edge(s).

**Figure 1.10:** The variable renaming phase of the SSA construction algorithm [2].
1  if (c == 0)
2  {
3    x = *p + a  .... (suppress exceptions)
4  }

(a) Optimization opportunity thwarted by a control-flow branch.

1  *p = ...
2  z = *p
3  x = z + a
4  y = z + b
5  *q = ...
6  check if p and q alias, and recover (if req.)
7  next:
8  
9  recover:
10     raise any exception generated by x = *p + a
11     goto next

(b) Control speculation allows hoisting of statements irrespective of control-flow branches.

Figure 1.11: Control speculation.

1  *p = ...
2  *q + a
3  y = *p + b

(a) Optimization opportunity thwarted by possibly aliasing store.

(b) Data speculation allows execution of instruction with predicted (and possibly wrong!) operands.

Figure 1.12: Data speculation.

- Control speculation pertains to executing an instruction before it is ascertained that the normal run of the program allows it. In figure 1.11(b), the statement \( x = *p + a \) is hoisted above the if branch — a transformation that helps conceal the latency of indirect load instruction whenever the predicate \( (c == 0) \) turns out true; such transformations can also be useful at removing partial redundancies speculatively. However, note that mis-speculation has a penalty: if \( (c == 0) \) turns out false, the transformation has added an instruction that would not have been executed in the original flow of instructions. Still worse, if this statement causes an exception (possibly due to an invalid address in \( p \)), it might alter the program behaviour; if control speculation is carried out on such unsafe instructions, hardware support is needed to suppress an exception due to the speculated instruction till it is ensured that the instruction is indeed executed in the normal course.
of the program as well.

- **Data speculation** refers to executing an instruction with predicted operands — arguments that are potentially faulty. For example, in figure 1.12(b), we replace the pointer dereference \*p by the temporary variable z — *speculating* that the pointers p and q do not alias. However, if our speculation fails, we need to recover by re-executing all the incorrectly executed instruction with correct operands.

Stirred by the effectiveness of such speculative decisions, modern architectures have started providing support for speculation. The IA64 architecture provides instructions to support both control and data speculation [10, 11]. For control speculation, it provides speculative instructions that are capable of recording the generation of an exception without raising it; a later check instruction branches to recovery code if the speculative instruction had generated an exception. For example, the *speculative load instruction* (ld.s) suppresses exceptions; any generated exception due to ld.s causes a later check instruction (chk.s) to branch to recovery code. For data speculation, IA64 provides the *advanced load* (ld.a) instruction that loads the intended address, as well as, records the details about the load instruction in the *Advanced Load Address Table (ALAT)*. A subsequent store to the same memory address marks the respective record in the ALAT invalid. Check and recovery is triggered by the *check load* (ld.c) instruction that repeats the load, and by the *advanced load check* (chk.a) instruction that branches to recovery code, if the respective ALAT entry is found invalid.

Control and data speculation has evoked considerable interest in the research community. Roy Dz-ching Ju et al. [14] propose a compiler framework for control and data speculation, providing extensive coverage of various compilation issues like generating recovery code and scheduling speculative instructions. Lin et al. [15] propose to include information from an alias profile and/or a set of heuristic rules into a speculative SSA form for exploiting data speculation. They select SSAPRE as an example to demonstrate the utility of their framework. Dai et al. [16] propose a framework for data speculative optimizations, such that, different optimizations can share the following:

- a common *speculative data dependence analysis* phase that ignores low probability data dependencies, illumining speculation opportunities;

- a *recovery code generation* phase that generates necessary recovery code depending on the speculative transformation applied.

Mahadevan et al. [17] propose to improve the instruction-level parallelism in modulo scheduled loops by employing data speculation. Lin et al. [18] tackle an important problem of effectively integrating the speculative instructions and recovery code generated by the different optimization phases. They correctly point out that this problem is of paramount value as recovery code generated in the earlier phases must not impede later optimizations.
Thread-level Speculation (TLS) [19, 20, 21, 22, 23, 24] is an attempt at extracting parallel workloads from sequential programs, thus improving the throughput on a multithreaded processor. Rather than proving that two regions of code are parallelizable, TLS optimistically spawn them into separate threads. During execution, if a data-dependency is encountered, the conflicting speculative thread is squashed.

In this thesis, we provide algorithms aimed at discovering path-profile directed speculative facts that can guide such control and data speculative optimizations. Please refer to the contributions discussed above for an understanding of how speculative transformations are implemented in compilation frameworks.

1.2 Our Contributions

1.2.1 The k-Iteration profiling algorithm

Though acyclic path profiles are useful at driving many compiler optimizations, more opportunities can be exploited in the presence of information about longer paths — paths extending across loop iterations. Figure 1.13 shows a control-flow graph being profiled, and the frequency counts for acyclic and two-iteration paths (paths including two iterations of the loop) for a program trace reading $1-(2-3-5-2-4-5)^{100}-6$ (i.e. the acyclic paths 2-3-5 and 2-4-5 execute alternately within the loop). Knowing the complete trace of the program, one would infer that it might be beneficial to unroll the loop, and trace schedule along the path 2-3-5-2-4-5. However, the acyclic path profile fails to suggest this optimization opportunity; it only shows that both the acyclic paths 2-3-5 and 2-4-5 are equally likely to occur in the trace, but misses out on any information across the loop boundaries. The only conclusion that can be drawn out of it for the longer set of paths is: paths 2-3-5-2-4-5 and 2-4-5-2-3-5, as well as $(2-3-5)^2$ and $(2-4-5)^2$, are all likely to be hot. Acyclic paths, due to the fact that they terminate at loop backedges, do not give any indication about the correlation among the loop iterations — multi-iteration paths can reveal such information.

Identifying this weakness of acyclic path profiling, Tallam et al. [3] proposed the use of longer paths — paths that cover two iteration of a loop — to enable multiple optimizations. They estimate the approximate frequencies of two-iteration paths within loops from Overlapping Ball-Larus paths — paths that extend a fixed number of branch nodes beyond the backedge.

In this thesis, we propose a profiling algorithm to collect the exact frequencies for such longer, multi-iteration paths. In contrast to the above proposal by Tallam et al., our algorithm provides the exact path frequency counts. Our algorithm is parameterized on the path-length (in terms of the maximum number of loop-iterations recorded), making it possible to profile paths spanning any number of loop-iterations. Essentially a generalization of the Ball-Larus profiling algorithm, our profiler reduces to the Ball-Larus profiler for one-iteration paths.
<table>
<thead>
<tr>
<th>Acyclic</th>
<th>freq</th>
<th>Two-Itt Path</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-3-5</td>
<td>1</td>
<td>1-2-3-5-2-4-5</td>
<td>1</td>
</tr>
<tr>
<td>2-3-5</td>
<td>99</td>
<td>2-3-5-2-4-5</td>
<td>98</td>
</tr>
<tr>
<td>2-4-5</td>
<td>99</td>
<td>2-4-5-2-3-5</td>
<td>99</td>
</tr>
<tr>
<td>2-4-5-6</td>
<td>1</td>
<td>2-3-5-2-4-5-6</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1.13: Motivation for k-iteration path profiling: For the graph in Figure 1.7, the two-iteration profile identifies two of the loop paths as hot; the acyclic path profile fails at delivering this information.

Note that, for $n$ static acyclic paths, there might be $O(n^k)$ static k-iteration paths in the worst case. An interesting aspect of our algorithm is that it allows mixed-mode profiling: the algorithm is capable of using different path-lengths (in terms of the number of iterations considered for the various loops) for each of the loops in the same procedure. The program analyst, thus, gets an option of employing k-iteration path-profiling for interesting regions, while exercising acyclic path profiling for the remaining program.

1.2.2 The Hot Path SSA form

In this work, we propose to weave a program’s run-time behaviour — as indicated by the frequently executed acyclic paths — directly in the program representation, easing the operation of speculative optimizers. In the proposed representation, which we refer to as the Hot Path SSA (HPSSA) form, an additional construct — the $\tau$-function — is added to capture information relevant to speculative analyses and optimizations. The $\tau$-functions act as “filters”, separating the more-likely reaching definitions from the lesser-likely ones. The first argument of the $\tau$-function is the traditional meet-of-all-paths reaching definition; the rest of the arguments are the hot reaching definitions: definitions that are more-likely to reach a respective program point.

Figure 1.14 shows the HPSSA form of the program in Figure 1.9. For example, the $\tau$-function at $g_1$ indicates that, although $x_9$ is the “safe” meet-of-all-paths reaching definition, it is the definitions of $x_7$ and $x_{17}$ that are more likely to reach this program point. Similarly, for $g_2$, $h_1$ and $h_2$, the hot reaching definitions are from the statements $y_4$, $x_{18}$ and $x_{19}$ — all of which are definitions to constants. By directly capturing the fact that the variables $y_{12}$, $x_{14}$ and $y_{15}$ are more likely to be constants with values 0, 3 and 2 (respectively), the HPSSA form empowers an optimizer to speculatively treat the statements $g_4$, $h_3$ and $h_4$ as assignments to constants.

1.3 Organization of the thesis

In this chapter, we introduced the idea of path-profile directed speculative optimizations, provided relevant background, and then, fluttered through the contributions of the thesis, laying a road-map
Figure 1.14: The Hot Path SSA (HPSSA) Form (The dotted lines indicate the hot paths: \( p_1: abfgi; p_2: acdfgi; p_3: aefhi \)).

for the following chapters. **Chapter 2** and **Chapter 3** detail the contributions of the thesis: the \( k \)-Iteration path profiling algorithm, and the Hot Path SSA form, respectively. **Chapter 4** outlines prior contributions in the areas of Control-Flow Profiling, extensions to the SSA Form, and Profile-Guided Optimizations. **Chapter 5** concludes the thesis, adumbrating the possible research directions opened by this thesis.

*The contributions of this thesis were published in [25, 26].*
Chapter 2

Profiling k-Iteration Paths

Program profiling is an indispensable tool for both compiler writers and computer architects for understanding the whims of an executing program. With an ecosystem of profile-guided optimizations surviving on reliable data about the run-time demeanor of programs, profiling has now engrossed an authoritative position in modern compilation.

As discussed in the introductory chapter, acyclic path profiling [1] is a popular choice for capturing a program’s run time control-flow behaviour. A vast planetary system of optimizations has materialized around acyclic path profiling.

However, many possible optimization opportunities are missed as these acyclic path profiles fail to perceive correlations across loop iterations. This necessitates a mechanism for capturing longer profile paths — paths spanning across loop iterations. Consider the example in Figure 2.1: in the loop, the

```
<table>
<thead>
<tr>
<th>acyclic</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-3-5</td>
<td>1</td>
</tr>
<tr>
<td>2-3-5</td>
<td>99</td>
</tr>
<tr>
<td>2-4-5</td>
<td>99</td>
</tr>
<tr>
<td>2-4-5-6</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>2-Itt Path</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-3-5-2-4-5</td>
<td>1</td>
</tr>
<tr>
<td>2-3-5-2-4-5</td>
<td>98</td>
</tr>
<tr>
<td>2-4-5-2-3-5</td>
<td>99</td>
</tr>
<tr>
<td>2-3-5-2-4-5-6</td>
<td>1</td>
</tr>
</tbody>
</table>
```

*Figure 2.1: An optimization enabled by k-Iteration Profiling.*
definitions of the variable $y$ (from the second iteration onwards) feed from the the value of $x$, set in the previous iteration. If the execution of the program runs as $1 - (2 - 3 - 5 - 2 - 4 - 5)^{100} - 6$, the definition of $y$ in basic-block 3 always grabs a constant value 2, while that in basic-block 4 always latches onto the constant 1. However, the acyclic path profile nonchalantly declares that both the loop paths $2 - 3 - 5$ and $2 - 4 - 5$ are equally likely to be executed. Any optimizer using this acyclic path profile, entirely ignorant of the above fact, would miss a (speculative) optimization opportunity. The two-iteration profile, on the other hand, screams out the desired fact: the presence of the path $2 - 3 - 5 - 2 - 4 - 5$ in the profile, and the absence of $2 - 4 - 5 - 2 - 4 - 5$, indicates that the definition of $y$ in basic-block 4 always witnesses a constant value 1 for the given trace; similar is the case for basic-block 3.

Tallam et al. [3] identifying the need for profiling such longer paths, proposed a scheme for capturing the approximate frequencies of such longer paths by collecting “slightly” longer, overlapping Ball-Larus paths — calling them Overlapping Paths (OL). Using these overlapping paths, they estimated two-iteration path profiles within loops, albeit tolerating some imprecision. They argued the importance of such profiles for optimizations like Partial Redundancy Elimination and Trace Scheduling; while applying Partial Redundancy Elimination, redundancy in instructions is often seen in loops, when the execution repeatedly runs into the same instruction in different loop iterations — a case often seen for arithmetic expressions and loads; for instruction scheduling, loop unrolling before scheduling necessitates profiles corresponding to multiple loop iterations [3].

In this chapter we outline an algorithm for k-Iteration profiling, which enables capturing profile information of multiple iteration paths — paths that are at most $k$ iterations long. In contrast to Tallam et al.’s work, we propose an algorithm for capturing the exact $k$-iteration path profile. Our algorithm is parametrized to allow any value of $k$, and requires no post-profiling activity.

Following are our contributions in this chapter:

- we generalize the Ball-Larus algorithm for more general $k$-iteration paths; we provide the path numbering and path identification algorithms for the same (section 2.2);

- we discuss the instrumentation scheme for $k$-iteration path profiling (section 2.3);

- we present experimental data on profiling such longer paths (section 2.5).

### 2.1 The k-Iteration Paths: A Novel Control-Flow Profiling Entity

This this section we formally describe $k$-Iteration paths; We first define what constitutes a valid $k$-Iteration path, pointing out what paths need to be marked invalid, with the help of some examples.

---

1 An earlier version of this work appeared in [25].
**Definition 2.1.** A $k$-iteration path is a path in the procedure’s flow graph that either ends at

- a backedge after the loop-body has been executed $k$ times, or
- the procedure exit.

Though the acyclic paths are essentially one-iteration paths, we will refer to them as Ball-Larus paths, BL paths or acyclic paths — the term *k-iteration paths* will strictly imply that $k > 1$. Figure 2.3 shows all the acyclic and $k$-iteration paths for the graph in Figure 2.2.

What is not a valid $k$-iteration path? A path must meet the following requirements to be a valid $k$-iteration path:

- Like acyclic paths, a $k$-iteration path may start only at the procedure entry or a loop-head (destination of a backedge), and end either at a loop-tail (source of a backedge) or the procedure exit.

- Any path that starts with the loop-head of loop $L$ must iterate through $L$ exactly $k$ times.

- Any path that terminates at the loop-tail of loop $L$ must also iterate through $L$ exactly $k$ times.

Note that the above set of requirements recognize paths that iterate a loop $L$ less than $k$ times as valid $k$-iteration paths if they do not start at the loop-head of $L$ and do not end at the loop-tail of $L$.

**Example 2.2.** Consider Figure 2.2: the paths 2-3-5-6 and 1-2-3-5 are invalid two-iteration path as they start with the loop-head (node 2), and terminate at the loop-tail (node 5) respectively, but iterate through the loop just once. However, the path 1-2-3-5-6, which also iterates through the loop just once is a valid two-iteration path as it neither starts with a loop-head, nor terminates at a loop-tail.
Acyclic Paths

0: 1-2-3-5-6 1: 1-2-3-5 2: 1-2-4-5-6 3: 1-2-4-5 4: 1-2-4-6
5: 2-3-5-6 6: 2-3-5 7: 2-4-5-6 8: 2-4-5 9: 2-4-6

Two-Iteration Paths

0: 1-2-3-5-2-3-5-6 1: 1-2-3-5-2-3-5 2: 1-2-3-5-2-4-5-6 3: 1-2-3-5-2-4-5 4: 1-2-3-5-2-4-6
5: 1-2-3-5-6 6: 1-2-4-5-2-3-5-6 7: 1-2-4-5-2-3-5 8: 1-2-4-5-2-4-5-6 9: 1-2-4-5-2-4-5
10: 1-2-4-5-2-4-6 11: 1-2-4-5-6 12: 1-2-4-6 13: 2-3-5-2-3-5-6 14: 2-3-5-2-3-5
15: 2-3-5-2-4-5-6 16: 2-3-5-2-4-5 17: 2-3-5-2-4-6 18: 2-4-5-2-3-5-6 19: 2-4-5-2-3-5
20: 2-4-5-2-4-5-6 21: 2-4-5-2-4-5 22: 2-4-5-2-4-6

Figure 2.3: Acyclic and Two-Iteration Static Paths.

If such invalid paths are not eliminated during path-numbering, the "perfect numbering" property (see section 1.1.1) would be violated.

The k-iteration path profiling algorithm encapsulates the following path numbering, path identification and program instrumentation algorithms:

- The k-iteration path numbering algorithm perfectly numbers the k-iteration paths, identifying and eliminating such invalid paths while assigning the path-identifiers (this problem does not occur for acyclic paths).

- The path identification algorithm reconstructs paths from a given path identifier.

- The program instrumentation algorithm uses an array of k counters to construct the k-iteration path-identifiers during the profile run of the program.

Understandably, the number of static paths increases exponentially in k — for n static acyclic paths, there will be $O(n^k)$ k-iteration static paths in the worst case. The number of static paths in a procedure is important; for a small number of static paths, the profiler can choose to use an inexpensive array-based implementation of the path-frequency table; otherwise, it would need an expensive hashtable-based implementation. To counter the same, our algorithm allows mixed-mode profiling: small, more interesting parts of the procedure can be profiled using k-iteration profiling, while employing acyclic path profiling for the rest of the procedure.

2.2 Path Numbering

We define a few terms to ease the following discussion: loop-head and loop-tail are the target and source nodes of a loop backedge, respectively. A loop-entry is a node in a loop that is the target of an edge from a node outside the loop; the respective edge is called the loop-entry edge. Similarly, a loop-exit is a node in a loop that is the source of an edge to a node outside the loop; the respective edge is called the loop-exit edge. We call the source of the loop-entry edge as the pre-entry node and the target of the
exit edge as the post-exit node. Any node that is part of a loop \( L \) is said to be a loop-node of \( L \). An edge \( u \rightarrow v \) is called a loop-edge of \( L \) if both \( u \) and \( v \) are loop-nodes of \( L \). For a reducible graph, the loop-head and the loop-entry nodes are the same. We add a dummy start node (denoted Start) and a dummy end node (denoted End) to the graph.

**Example 2.3.** In Figure 2.2, node 2 is the loop-entry node as well as the loop-head. Node 4 and 5 are the loop-exits. Node 5 is also the loop-tail. Edge 1\( \rightarrow \)2 is the loop-entry edge, 5\( \rightarrow \)2 is the backedge while 4\( \rightarrow \)6 and 5\( \rightarrow \)6 are the loop-exit edges. Nodes 1 and 6 are the pre-entry and post-exit nodes respectively.

In the following sub-sections, we discuss the path-numbering and path-identification algorithms. We assume a reducible graph with no nested loops. In our algorithms, we assume the absence of self-loops.

### 2.2.1 The loop-unrolled DAG

Before discussing the algorithms, let us look at a special graph — the loop-unrolled DAG — which will enable us to understand the following algorithms better. For k-iteration paths, the k-iteration loop-unrolled DAG, \( G_{lu(k)} \), can be formed from a graph \( G \) via the following steps (for each loop in the graph):

- unroll the loop, creating \( k \) copies of it: as we are interested in tracing paths that contain \( k \) iterations of the loop, unrolling the loop reveals all such paths; a node \( n^i \) in the DAG is called the \( i^{th} \) version of node \( n \) in the CFG if \( n^i \) corresponds to the node \( n \) in the \( i^{th} \) copy of the loop;

- add a dummy edge from the Start node to the first version of the loop-head; paths originating from this edge denote k-iteration paths in \( G \) that begin with the loop-head;

- add a dummy edge from the \( k^{th} \) version of the loop-tail to the End node; paths terminating with this edge denote k-iteration paths in \( G \) that end at the loop-tail;

- remove the backedges.

Similar to acyclic path-profiling, the dummy edges create new paths in \( G_{lu(k)} \) that either start at the loop-head or terminate at the loop-tail. As the loop needs to be traversed exactly \( k \)-times for such k-iteration paths, only the first version of the loop-head in \( G_{lu(k)} \) is a target of a dummy edge from the Start node, and only the \( k^{th} \) version of the tail-node is the source of a dummy edge to the End node — this disallows all paths that are traversed fewer than \( k \) times. As the loop is unrolled to contain exactly \( k \) copies and all the backedges are removed, all paths in \( G \) that traverse the loop more than \( k \) times are absent in \( G_{lu(k)} \).

**Example 2.4.** Figure 2.4 shows the two-iteration loop-unrolled DAG (\( G_{lu(2)} \)) for the graph in Figure 2.2.
2.2.2 The Path Numbering Algorithm

The goal of the path-numbering algorithm (Algorithm 2.1) is to assign unique identifiers to all the possible k-iteration paths in a given graph \( G \). Instead of unrolling the loops to create \( G_{lu(k)} \), we rather construct a new graph \( G_\delta \) from \( G \), simply by adding dummy edges from the Start node to each loop-head (we denote all such dummy edges by \( \delta^{start} \)) and dummy edges from the loop-tail to the End node (we denote all such dummy edges by \( \delta^{end} \)) to allow static paths beginning at a loop-head and ending at a loop-tail (for a loop \( L \), we denote the dummy edges for it by \( \delta_L^{start} \) and \( \delta_L^{end} \)). The \( k^{th} \) iteration through a loop \( L \) in \( G_\delta \) corresponds to the traversal through the \( k^{th} \) copy of \( L \) in \( G_{lu(k)} \).

The path-numbering algorithm simulates the traversal through \( G_{lu(k)} \) by iterating through loops in \( G_\delta \) \( k \)-times while ignoring appropriate edges in each iteration; the \texttt{Ignored} function (Algorithm 2.1) indicates if an edge is ignored in a particular iteration.

2.2.2.1 Numbering all the k-iteration static paths

The algorithm computes a set of weights for all edges in a graph \( G \) such that the sum of these weights along any path gives an unique identifier for the respective k-iteration path. For this purpose, the algorithm needs to compute \texttt{numPaths(n)} , the number of paths through each node \( n \). For example, in Figure 2.4, there exists six paths through \( 3' \) and seven paths through \( 4' \). Hence, \texttt{numPaths(2')}=13.

The \texttt{Path Numbering} algorithm (Algorithm 2.1) traverses the nodes in \( G_\delta \) in a reverse topological order, computing the number of paths originating from each node, and assigning edge-weights to its outgoing edges (similar to the Ball-Larus algorithm). On entering a loop \( L \), it starts off by computing
Algorithm 2.1 Path Numbering of k-iteration static paths; Ignored gives the set of all edges that need to be ignored in a particular iteration of the loop.

**Input**

- \( g \): graph with dummy edges \( G_d \)
- \( k \): the k-iteration profiling parameter

**Output**

- \( \text{val}, \text{cval} \): set of edge-weights and compensation edge-weights

begin

\[ \forall u \in g, \text{set } \text{itrNum}(u) := \begin{cases} \ k & \text{if } u \text{ is a loop node,} \\ \ 1 & \text{otherwise} \end{cases} \]

\[ \text{while } u \neq \text{Start do} \]

\[ u := \text{Get the next item from } \text{list} \]

\[ i := \text{itrNum}(u) \]

\[ \text{if } u \neq \text{End then} \]

\[ \text{numPaths}(u) := 0 \]

\[ \text{else} \]

\[ \text{numPaths}(u) := 1 \]

\[ \text{end if} \]

\[ \text{invalidPaths}(u) := 0 \]

\[ \text{for all edges } (e : u \rightarrow v) \in g \text{ do} \]

\[ \text{if } \neg(\text{Ignored}(e, i)) \text{ then} \]

\[ \text{val}(e, i) := \text{numPaths}(u) \]

\[ \text{numPaths}(u) += \text{numPaths}(v) \]

\[ \text{if } i < k \wedge u \in \{ \text{loop-nodes of } L \} \cup \{ \text{Start} \} \text{ then} \]

\[ \text{cval}(e, i) := \text{invalidPaths}(u) \]

\[ \text{if edge } e \in \text{loop-exit-edges of } L \text{ then} \]

\[ \text{invalidPaths}(u) += \text{numPaths}(v) \]

\[ \text{else} \]

\[ \text{invalidPaths}(u) += \text{invalidPaths}(v) \]

\[ \text{end if} \]

\[ \text{else} \]

\[ \text{cval}(e, i) := 0 \]

\[ \text{end if} \]

\[ \text{end for} \]

\[ \text{if } u = \text{head-node of some loop } L \wedge i > 1 \text{ then} \]

\[ \text{Set } \text{itrNum}(x) := (i - 1) \text{ for all nodes } x \in L \]

\[ \text{Set the list pointer to the tail node of the loop } L \]

\[ \text{end if} \]

end while

end

**Function:** Ignored \( u \rightarrow v : \text{edge, } i : \text{iteration number} \)

\[ \text{if } i > 1 \wedge (u \rightarrow v \text{ is a loop-entry edge } \lor u \rightarrow v \in \delta^{\text{start}}) \text{ then} \]

\[ \text{return } \text{true} \]

\[ \text{end if} \]

\[ \text{if } i < k \wedge u \rightarrow v \in \delta^{\text{end}} \text{ then} \]

\[ \text{return } \text{true} \]

\[ \text{end if} \]

\[ \text{if } i = k \wedge u \rightarrow v \text{ is the loop backedge} \text{ then} \]

\[ \text{return } \text{true} \]

\[ \text{end if} \]

\[ \text{return } \text{false} \]
the val and numPaths values for the last iteration of the loop (iteration k) — as the graph is traversed in a backward direction, the k-th iteration is seen first. However, on reaching the loop-head, it ignores the loop-entry edges and \( \delta_L^{\text{Start}} \) (as decided by the \texttt{Ignored} function); as the list pointer is set back to the loop-tail, the algorithm is forced to follow the backedge again, performing another backward traversal of L, now computing the val and numPaths values for iteration k-1. The algorithm continues traversing the loop for a previous iteration each time till the first iteration is reached. On reaching the loop-head in the first iteration, it is the backedge that is ignored, and the algorithm exits the loop via the loop-entry edges and \( \delta_L^{\text{Start}} \). The vector \( \texttt{itrNum(n)} \) keeps track of the iteration number for which the computation of \( \texttt{numPaths(n)} \) and \( \text{val(n, i)} \) still needs to be done. It is easy to see that the k-iteration paths in \( G_\delta \) — ignoring the edges discounted by the \texttt{Ignored} function — is exactly that in \( G_{lu(k)} \).

The computation of val and numPaths is similar to the Ball-Larus algorithm; however, instead of being simple integers, val for an edge \( e \) is now a vector indexed by the iteration number. Also, \( \texttt{numPaths(n)} \) gives the number of paths from node \( n \) corresponding to the last iteration for which this value was computed (as the order of visiting the nodes in \( G_\delta \) simulates a reverse topological traversal of \( G_{lu(k)} \), the value of \( \texttt{numPaths(n)} \) for the last iteration for which \( n \) was visited is sufficient to compute val for all remaining iterations).

2.2.2.2 Eliminating invalid k-iteration paths

The loop-unrolled DAG, \( G_{lu(k)} \), contains all the static k-iteration paths for a given graph \( G \); however, the valid k-iteration paths are still lesser. The DAG \( G_{lu(k)} \) also includes invalid k-iteration paths: paths in \( G \) that begin at a loop-head and iterate through the loop fewer than \( k \) times, exiting via a loop-exit edge in an iteration \( i < k \). In \( G_{lu(k)} \), these paths originate from a \( \delta^{\text{start}} \) edge, and later follow an exit-edge of an \( i \text{'th} \) copy of the loop, where \( i < k \).

For example, in Figure 2.4, for the paths from the \texttt{Start} node that reach \( 2' \), the ones beginning with the dummy edge \( \text{Start} \rightarrow 2' \) and passing via \( 4' \rightarrow 6 \) or \( 5' \rightarrow 6 \) would be invalid as these paths originate from a dummy edge, and exit the loop in the first iteration itself; having three such paths, invalidPaths(\( 2' \)) = 3.

Why are such paths deemed invalid? Observe that a path in \( G \) that begins at a loop-head and iterates through a loop \( i \) times, where \( i < k \), is essentially a valid \( i \)-iteration path. Every valid \( i \)-iteration path, where \( i < k \), is “contained” in some valid k-iteration path; In an actual execution of the program, a k-iteration path that begins at a loop-head is a path that got “chopped” at a backedge. Hence, for any \( i \)-iteration path (\( i < k \)) that begins at a loop-head, the corresponding k-iteration path in the execution trace can be constructed by prepending the following to this “short” \( i \)-iteration path:

- the previous \( k-i \) iterations of the loop in the execution trace (if the execution has traversed the loop at least \( k \) times);
path-identifier\( (p) = \sum_{e : u \to v \in p} val(e, i) - \begin{cases} 
  cval(e, i) & \text{if } \delta^\text{start}_L \in p \land (e = \delta^\text{start}_L \lor (u \in L \land i < k)) \\
  0 & \text{otherwise}
\end{cases}
\)

\( (2.1) \)

- the loop-entry path, followed by the iterations that leads to this \( i \)-iteration path in the execution trace (if the execution has iterated the loop less than \( k \) times).

An analogy to understanding is how a basic-block profile is “contained” in an edge profile.

If such invalid paths are not eliminated in the path-numbering phase, these “redundant” paths would waste precious slots in the path-frequency table. As the performance of the profiler depends on the size of the path-frequency table\(^3\), eliminating the invalid paths is of paramount importance.

In our algorithm, these paths are compensated via compensation weights (\( cval \)): an additional set of weights assigned to edges such that using \( \sum_{e \in \text{path}} val(e, i) - cval(e, i) \) as the edge-weight instead of \( \sum_{e \in \text{path}} val(e, i) \) for appropriate edges in an iteration \( i \), still numbers the valid paths uniquely. This compensation is applied for an edge \( e \) in \( G_\delta \) if:

- the edge \( e \) appears in a path \( p \) that commences with \( \delta^\text{start}_L \) (which implies that, in \( G \), the path starts with a loop-head) i.e. \( \delta^\text{start}_L \in p \); and

- either
  - \( e \) is the dummy edge \( \delta^\text{start}_L \) i.e. \( e = \delta^\text{start}_L \), or
  - \( e : u \to v \) is either a loop-edge or a loop-exit edge, and \( e \) is being traversed for the \( i^\text{th} \) iteration of the loop \( L \), where \( i < k \) i.e \( u \in L \land i < k \).

Hence, after compensation, the path-identifier for a valid \( k \)-iteration path is given by equation (2.1).

With these identifiers, not only are the paths numbered **uniquely**, but also **perfectly**: if there are \( n \) valid \( k \)-iteration paths through a node, the paths through it would be assigned path-identifiers from 0 to \( n-1 \), with no two valid paths having the same identifier.

Refer to the computation of invalidPaths and cval in Algorithm 2.1: invalidPaths\( (n) \) computes the number of invalid paths passing through the node \( n \) for the last iteration for which this value was computed if this node was reached via \( \delta^\text{start} \). Traversing the nodes of the loop in reverse topological order, the algorithm computes the number of such invalid paths through each node, and accordingly, assigns compensation weights to the outgoing edges (the number of invalid paths through a node is equal to the sum of the invalid paths through all its successors). The number of invalid paths via a loop-exit edge \( u \to v \) is accounted for by the number of possible paths through \( v \): for each path that originates

\(^3\)“small” tables can be implemented as an array; “big” tables need expensive hashtables.

25
from an edge $\delta^{\text{start}}$ and reaches $u$, all paths that exit via the edge $u \rightarrow v$ are invalid — and that is exactly equal to the number of paths through $v$. The values for $\text{cval}(n,i)$ is computed using the values for $\text{invalidPaths}(n)$; the computation is similar to how $\text{val}(n)$ is computed using $\text{numPaths}(n)$.

**Example 2.5.** Figure 2.5 shows the values for $\text{val}$ computed by the path-numbering algorithm. The weights for the two iterations are shown separated by '/'; the compensation weights for iteration number 1 are indicated in the brackets. The edges not having any weights (or having a weight zero) are left unmarked.

Table 2.1 shows the execution trace of how the number of paths(n), the number of invalid paths(in), edge-weights(v), and compensation edge-weights(cv) get computed for this graph.

### 2.2.3 The Path Identification Algorithm

Given a path-identifier $n$, the Path Identification algorithm (Algorithm 2.2) finds a valid path from the Start node to the End node, such that, the sum of the edge-weights is equal to $n$. Beginning at the Start node, for each node identified to be in the path, it greedily select an outgoing edge with the highest $\text{val}$, but not allowing the partial path-identifier $\text{pathWeight}$ (adjusted with the compensated edge-weight of the edge $e$) to exceed $n$. Once an edge $e$ is selected to be on the path, $\text{pathWeight}$ is updated by adding the compensated edge-weight of $e$ to it. The edge-weight val is compensated with the compensation weight $\text{cval}$ in the following cases:

- the current edge $u \rightarrow v \in \delta^{\text{start}}$; or

Table 2.1: An execution trace of the path-numbering algorithm. $S$ and $E$ represent the Start and End nodes respectively; '-' indicates the value 0.

<table>
<thead>
<tr>
<th>$u$</th>
<th>$v$</th>
<th>$s(u)$</th>
<th>$v(e, i(u))$</th>
<th>$n(u)$</th>
<th>$\text{cval}(e, i(u))$</th>
<th>$\text{in}(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>26</td>
<td>0</td>
</tr>
</tbody>
</table>

(u: selected node; e: selected edge; i: itrNum; v: val; n: numPaths; cv: cval; in: invalidPaths)
### Algorithm 2.2 The Path Identification Algorithm

**Input**
- $g$: graph with dummy edges ($G_δ$)
- $n$: path-identifier whose path is to be identified
- $k$: the $k$-iteration profiling parameter
- $val$, $cval$: edge-weights and compensation edge-weights

**Output**
- $path$: sequence of edges identifying the path for identifier $n$

**begin**

$u := \text{Start}$

$\text{pathWeight} := 0$

$\text{path} := [ ]$

$\forall x \in g, \text{set visitCount}(x) := 0$

**while** $u \neq \text{End} \text{ do}$

$i := \text{visitCount}(u)$

if $u = \text{loop-tail of loop } L$ then

Set $\text{visitCount}(x) := (i + 1)$ for all nodes $x \in L$

else

$\text{visitCount}(u) := (i + 1)$

end if

$i := \text{visitCount}(u)$

$maxVal := -1$

$maxWeight := 0$

for all edges $e : u \rightarrow v \wedge \neg \text{Ignored}(e, i)$ do

if $(e = \delta^i_{\text{start}}) \lor (\delta^i_{\text{start}} \cap \text{path} \neq \emptyset \land u \in L \land i \neq k)$ then

$cweight := val(e, i) - cval(e, i)$

else

$cweight := val(e, i)$

end if

if $val(e, i) \geq maxVal \land \text{pathWeight} + cweight \leq n$ then

$w := v$

$maxVal := val(e, i)$

$maxWeight := cweight$

end if

end for

$\text{pathWeight} := \text{pathWeight} + \text{maxWeight}$

Add $u \rightarrow w$ to $\text{path}$

$u := w$

end while

end

- the $\delta^i_{\text{start}}$ edge was included in the path to the current edge $u \rightarrow v$, $u \in L$; and the algorithm has not yet iterated $k$ times through the loop $L$.

The algorithm accumulates edges till a whole path from the $\text{Start}$ node to the $\text{End}$ node is formed. The algorithm needs to be run on the graph $G_δ$, i.e. before the dummy edges are removed.
2.2.4 Correctness Results

Let $\text{numPaths}(n)$ denote the number of all paths through a node $n$; $\text{invalidPaths}(n)$ denotes the number of invalid paths through a node $n \in L$, when $n$ lies on a path originating from the edge $\delta^\text{start}_L$.

We prove the correctness results on the loop-unrolled DAG $G_{lu(k)}$ rather than the CFG $G$, as it is actually $G_{lu(k)}$ that is traversed by the path-numbering algorithm (the condition $\neg(Ignored(e, i))$ controls this traversal). Each pair $(v, i), 1 \leq i \leq k$, where $v$ represents a vertex and $i$ an iteration-number, forms an unique node in $G_{lu(k)}$ with $\text{itrNum}((v, i))$ $\neg-1$, indicating the copy of the loop that $v$ belongs to. Hence, in our proofs, $\text{val}(v)$ and $\text{cval}(v)$ take single arguments (instead of $\text{val}(v, i)$ and $\text{cval}(v, i)$ as used in our algorithm).

We often consider the following disjoint set of paths:

- $N$: A set of normal paths that begin with the edge $\text{start} \rightarrow \text{entry}$, where $\text{entry}$ is the actual entry-point to the procedure.

- $D_i$: A set of dummy paths that begin with the dummy edge $\delta^\text{start}_L$.

**Lemma 2.6.** The values $\text{numPaths}(v)$ and $\text{invalidPaths}(v)$ are computed correctly for each node $v$ in the $k$-iteration loop-unrolled DAG.

**Proof.** The proof is by induction on the height of a node in the DAG i.e. the length of its longest path to the $\text{End}$ node.

**Base Case:** It is trivially satisfied for height $H = 0$, i.e. for the $\text{End}$ node.

**Induction Step:** Consider a node $u$ at a height $H > 0$.

**Computation of $\text{numPaths}(v)$:** Surely all paths from its successors are numbered uniquely and perfectly (by inductive hypothesis) as they are at a height less than $H$ and the graph is a DAG. Also, the number of paths through a node is simply the sum of paths through all of its successors; thus, $\text{numPaths}(u) = \sum_{v \in S(u)} \text{numPaths}(v)$.

**Computation of $\text{invalidPaths}(v)$:** The number of invalid paths through a node is equal to the sum of the invalid paths via all its outgoing edges: $\text{invalidPaths}(u) = \sum_{v \in S(u)} \text{invalidPathsVia}(u \rightarrow v)$.

**Case I:** If $u$ $\text{Start}$ and $u \rightarrow v \in \delta^\text{start}_L$, then $\text{invalidPathsVia}(u \rightarrow v) = \text{invalidPaths}(v)$.

**Case II:** If $u \neq \text{Start}$ and $u$ does not belong to a loop, $\text{invalidPathsVia}(u \rightarrow v) = 0$ as there cannot be such invalid paths outside loops.

**Case III:** Let $u \in L$, where $L$ is a natural loop:

- If $v \notin L$ and $\text{itrNum}(u) = k$, then $u \rightarrow v$ is a loop-edge for $L$. Also, as $\text{itrNum}(u) = k$, all paths originating from $\delta^\text{Start}_L$ that reach $u$ have already seen $k$ iterations of $L$; hence, $\text{invalidPathsVia}(u \rightarrow v) = 0$. 

28
invalidPaths(u) = \sum_{v \in succ(u)} \begin{cases} 
  \text{numPaths(v)} & \text{if } u \in L, \text{itrNum}(u) < k, \text{ } u \rightarrow v \text{ is a loop-exit edge of } L \\
  \text{invalidPaths(v)} & \text{if } u, v \in L \text{ or } u = \text{Start} \\
  0 & \text{otherwise} 
\end{cases} 

(2.2)

- If \( v \notin L \) and \( \text{itrNum}(u) < k \); then \( u \rightarrow v \) is a loop-exit edge for \( L \). Also, as \( \text{itrNum}(u) < k \), all paths originating from \( \delta^\text{Start}_L \) that reach \( u \) have not yet seen \( k \) iterations of \( L \). Hence, \( \text{invalidPathsVia}(u \rightarrow v) = |\delta^\text{Start}_L| \times \text{numPaths}(v) \) (where \( |\delta^\text{Start}_L| \) denotes the number of \( \delta^\text{Start}_L \) edges in the graph); a path originating from \( \delta^\text{Start}_L \) that passes through the loop-exit edge \( u \rightarrow v \) is an invalid \( k \)-iteration path (as the path began with the loop-head for \( L \), but exited the loop via \( u \rightarrow v \) before iterating through the loop \( k \)-times). Also, for a loop \( X \), there can be only one \( \delta^\text{Start}_X \) edge, so \( |\delta^\text{Start}_X| = 1 \).

- If \( v \in L \), then \( u \rightarrow v \) is a loop edge for \( L \): \( \text{invalidPathsVia}(u \rightarrow v) = \text{invalidPaths}(v) \).

To summarize, the invalid paths for a node \( u \) is given by equation (2.2).

\[ \square \]

**Lemma 2.7.** For all paths \( p \in D_n \), the valid sub-paths from a node \( u \) are assigned consecutive path-identifiers (from 0 to \( \text{numPaths}(u) - \text{invalidPaths}(u) - 1 \)) by the path-numbering algorithm (while the last \( \text{invalidPaths}(u) \) path-identifiers remain unassigned).

**Proof.** We will prove the lemma by induction on the height of the DAG.

**Base Case:** The theorem is trivially satisfied for height \( h = 0 \).

**Inductive Step:** Consider a node \( u \) at a height \( h > 0 \). Now, if \( u \notin \{\text{Start}\} \cup L_n \), or \( u \in L_n \land \text{itrNum}(u) = k \); then \( \text{invalidPaths}(u) = 0 \)— as there are no invalid paths, in these cases the lemma is trivially satisfied.

Otherwise, the edge \( u \rightarrow v_i \) may be:

- a loop-exit edge for the loop \( L_n \) with \( \text{itrNum}(u) < k \); in this case, \( \text{invalidPathsVia}(u \rightarrow v_i) = \text{numPaths}(v_i) \), i.e. all the paths through this edge are invalid. So, all the \( \text{numPaths}(v_i) \) path-identifiers through \( u \rightarrow v_i \) can be considered unassigned;

- either \( \delta^\text{Start}_L \), or an edge in \( L_n \) with \( \text{itrNum}(u) < k \); in this case, \( \text{invalidPathsVia}(u \rightarrow v_i) = \text{invalidPaths}(v_i) \), and all the valid paths through \( v_i \) are consecutively numbered from 0 to \( \text{numPaths}(v_i) - \text{invalidPaths}(v_i) - 1 \) by the induction hypothesis.

Thus, the valid paths through \( u \rightarrow v_1 \) (where \( v_1 \) is the first successor of \( u \)) are numbered from 0 to \( (\text{numPaths}(v_1) - \text{invalidPathsVia}(u \rightarrow v)) - 1 \). In general, the algorithm numbers all the valid paths till the edge \( u \rightarrow v_i \) from 0 to \( \sum_{1 \leq j \leq i} (\text{numPaths}(v_j) - \text{invalidPathsVia}(u \rightarrow v_j)) - 1 \). As the valid
paths through \( u \rightarrow v_{i+1} \) are also consecutively numbered, the paths till \( u \rightarrow v_{i+1} \) get consecutively numbered from 0 to \( \sum_{1 \leq j \leq i+1} (\text{numPaths}(v_j) - \text{invalidPathsVia}(u \rightarrow v_j)) - 1 \).

Hence, all valid sub-paths for the paths \( p \in D_n \) through the node \( u \) get consecutively numbered from 0 to \( \sum_{1 \leq j \leq |\text{succ}(u)|} (\text{numPaths}(v_j) - \text{invalidPathsVia}(u \rightarrow v_j)) - 1 \) which is exactly same as \( \text{numPaths}(u) - \text{invalidPaths}(u) - 1 \). The last \( \text{invalidPaths}(u) \) path-identifiers, thus, remain unassigned.

\[ \square \]

**Theorem 2.8.** The path numbering algorithm uniquely and perfectly numbers all the valid paths.

**Proof.** We again prove the same by induction on the height \( h \) of the DAG. Let the successors of \( u \) be \( v_1, v_2, \ldots, v_n \). The edge weights are set as, \( \text{val}(u \rightarrow v_i) = \sum_{1 \leq j < i} \text{numPaths}(v_j) \) and \( \text{eval}(u \rightarrow v_i) = \sum_{1 \leq j < i} \text{invalidPaths}(v_j) \).

We first prove the theorem for the sub-paths of each of the following disjoint set of paths separately:

1. If \( p \in N \), all the paths are valid and hence no compensation weights are used.

   **Base Case:** The theorem holds trivially for height \( h = 0 \).

   **Induction Step:** Consider a node \( u \) at a height \( h > 0 \). By induction hypothesis, the sub-paths of \( p \) from each of its successor node \( v_i \) are numbered uniquely from 0 to \( \text{numPaths}(v_i) \). As, the number of the sub-paths from \( u \) to all its successor from \( v_1 \) to \( v_{i-1} \) is \( \sum_{1 \leq j < i} \text{numPaths}(u \rightarrow v_j) \), the algorithm numbers the sub-paths through \( u \rightarrow v_i \) from \( \sum_{1 \leq j < i} \text{numPaths}(u \rightarrow v_j) \) to \( (\sum_{1 \leq j < i} \text{numPaths}(u \rightarrow v_j)) + \text{numPaths}(v_i) - 1 \) — these are clearly unique and perfect.

2. If \( p \in D_i \), all paths through the loop-exit paths of the loop \( L_i \) are invalid for all copies of the loop till \( k-1 \).

   **Base Case:** The theorem holds trivially for height \( h=0 \).

   **Induction Step:** Consider a node \( u \) at a height \( h > 0 \). By induction hypothesis, the valid sub-paths of \( p \) from each of its successor node \( v_i \) are numbered uniquely from 0 to \( \text{numPaths}(v_i) - \text{invalidPaths}(v_i) - 1 \), while the last \( \text{invalidPaths}(v_i) \) identifiers are unassigned by the lemma 2.7. We may assign these unassigned path identifiers to other valid sub-paths, i.e. ones that pass through the next successor node. So, ignoring all the invalid sub-paths, the valid sub-paths through \( u \rightarrow v_i \) are numbered from \( \sum_{1 \leq j < i} (\text{numPaths}(v_j) - \text{invalidPaths}(v_j)) \) to \( \sum_{1 \leq j < i} (\text{numPaths}(v_j) - \text{invalidPaths}(v_j)) + \text{numPaths}(v_i) - \text{invalidPaths}(v_i) - 1 \). Note that numbering of sub-paths through each edge \( u \rightarrow v_i \) leaves \( \sum_{1 \leq j < i} \text{invalidPaths}(v_j) \) path-identifiers of the \( \sum_{1 \leq j < i} \text{numPaths}(v_j) \) paths unassigned. Conceptually, to ignore the invalid paths through each of the edges \( u \rightarrow v_i \), the path-identifiers for edges \( u \rightarrow v_j, j > i \) are “shifted” by \( \text{invalidPaths}(v_i) \). The total compensation (or the shift) that needs to be assigned to any edge \( u \rightarrow v_i \) is \( \text{eval}(u \rightarrow v_i) = \sum_{1 \leq j < i} \text{invalidPaths}(v_j) \).
\[ \sum_{j < i} invalidPathsVia(v_j). \] These path identifiers for all the valid paths are:

- **unique** as subtracting \( \text{eval}(v_i) \) only allows assignment of the unassigned path-identifiers to other valid sub-paths, but never overlaps it with already assigned path-identifiers of valid paths;

- **perfect** as path-identifiers of all the invalid sub-paths are now assigned to valid sub-paths (or remain unassigned with values greater than the path-identifier of the last valid sub-path).

Now, consider the Start node: the paths through its outgoing edges are numbered uniquely and perfectly as proved above. The algorithm also numbers the paths through an edge \( \text{start} \rightarrow v_i \) in exactly the same manner; from \( \sum_{1 \leq j < i} (\text{numPaths}(v_j) - \text{invalidPaths}(v_j)) \) to \( \sum_{1 \leq j < i} (\text{numPaths}(v_j) - \text{invalidPaths}(v_j)) + \text{numPaths}(v_i) - \text{invalidPaths}(v_i) - 1 \). By exactly the same argument as above, the path-identifiers for all the paths from the Start node are also unique and perfect.

\[ \square \]

### 2.3 Program Instrumentation

#### 2.3.1 Removal of the dummy edges

As the dummy edges do not represent real control-flow, for instrumentation, their effect needs to be “simulated” by adjusting their weights on the actual control-flow edges appropriately. The dummy edges can be removed in a manner similar to that described by Ball and Larus [1]. To remove a dummy edge \( \text{Start} \rightarrow v \), we subtract the edge's weight from all incident edges of \( v \) and add it to all outgoing edges of \( v \).

For k-iteration profiling, we also need to be careful about updating the correct iteration edge-weights for a loop \( L \):

- on removing \( \delta_L^{\text{start}} \), the edge weights of only the first iteration of the loop \( L \) are affected;

- on removing \( \delta_L^{\text{end}} \), the edge weights of only the \( k \)th iteration of the loop \( L \) are affected.

However, for the exit-edge of a loop \( L \), the edge-weights for all the iterations of \( L \) may be affected on removal of a dummy edge of another loop \( L' \).

For example, consider figure 2.8: the loop-unrolled DAG shows that on removing \( \delta_{L_3}^{\text{start}} \), the edge-weight on it needs to be added to the edge-weights of both its incoming edges \( 3' \rightarrow 4' \) and \( 3'' \rightarrow 4' \); this indicates that the edge-weights for both the iterations of the edge \( 3 \rightarrow 4 \) in the original graph needs adjustment. Similarly, on removing \( \delta_{L_3}^{\text{end}} \), the edge-weights of both the iterations of \( 5 \rightarrow 6 \) get affected.

**Example 2.9.** The adjusted graph weights on removing the dummy edges can be seen in Figure 2.6.
2.3.2 The Instrumentation Algorithm

The instrumentation algorithm (Algorithm 2.3) needs \( k \) accumulators \((c[0], c[1], \ldots, c[k-1])\), where \( c[1] \) accumulates the partial \((i+1)\)-iteration path-identifier of a loop in execution; as \( k \)-iteration paths can at most be \( k \) iterations long, \( k \) such counters are sufficient. For example, say a program is executing the 20\(^{th}\) iteration of a loop — for two-iteration path profiling, the counter \( c[0] \) will accumulate the partial path-identifier for the acyclic path executed in the 20\(^{th}\) iteration while \( c[1] \) will accumulate the two-iteration path-identifier spanning the 19\(^{th}\) and the 20\(^{th}\) iteration. The algorithm also needs a flag \( f_\delta \) to remember if or not the \( k \)-iteration path starts with the loop-head (i.e. the path in the loop-unrolled DAG starts with \( \delta^{\text{start}} \)). As, at least \( k \) iterations through a loop is needed to form a \( k \)-iteration path, \( f_\delta \) also indicates if the loop has already seen \( k \) iterations. The saturating counter \( (\text{itr}) \) identifies the initiating \( k-1 \) iterations of the loop. It is important to identify the iterations \( i < k \), as, till then, the path-counters \( c[j], j \geq i \) contain garbage values. After \( k \) iterations of the loop, all the path counters contain valid values.

The path-frequency table is maintained either as an array or a hashtable; the macro \texttt{SelectImplementation} selects one of them depending on the number of static paths in the current procedure. The function \texttt{IncrPathFreq(p)} increments the frequency count of the path with identifier \( p \) while profiling. Pointers to the path-frequency tables for each procedure are maintained in a program-wide global table. The initialization code, inserted in the respective procedure’s \texttt{Start} node, initializes the local variables and selects an appropriate implementation of the path-frequency table.

To accumulate the path-identifiers, as in the Ball-Larus scheme, our algorithm splits appropriate edges and adds instrumentation code. For an edge \( e \) in the CFG, the \( i^{th} \) counter in the counter array, \( c[i-1] \), is incremented with the edge weight \( \text{val}(e, i) \), adjusted with the compensation weight \( c\text{val}(e, i) \), if any. As all paths except the loop-entry path need compensation, till \( f_\delta \) turns \texttt{true}, only the counter \( c[\text{itr+1}] \) among the valid counters is excluded from compensation by the condition “\((\text{id} < \text{itr})\)” (the counters \( c[i] \), where \( i > \text{itr+1} \), do not carry valid values at this stage). After \( k \) iterations of the loop, when the loop-entry path retires and the flag \( f_\delta \) turns \texttt{true}, \( f_\delta \) simply “short-circuits” the check.

For the backedge of loops profiled using \( k \)-iteration profiling, the path-frequency table can be updated only after \( k \) iterations, as a \( k \)-iteration path is not created until the end of the \( k \)-th iteration. Each path-accumulator \( c[i] \) loads the partial path-identifier currently held in \( c[i-1] \) and \( c[0] \) is cleared. The iteration counter \( \text{itr} \) is incremented (if not saturated); if \( k \) iterations of the loop have been seen, the flag \( f_\delta \) is set.

\textsuperscript{3} \texttt{SelectImplementation} expands at compile time to the appropriate implementation of the path-frequency table.
Algorithm 2.3 The Instrumentation Algorithm; the notation $x << Y$ implies that the entity $x$, which can either be a node or an edge, is instrumented with profiling code $Y$.

**Input**

- $g$: control-flow graph
- $k$: the $k$-iteration profiling parameter
- val, evak: edge-weights and compensation edge-weights after removing dummy edges

**Output**

- the instrumented program

begin

Start

```text
\$\$c[0] := c[1] := \ldots := c[k-1] := 0$
\$\$f_s := \text{false}$
\$\$itr := 0$
\$\$\text{SelectImplementation(numPaths(g))}$
```

for all edges $e \in \text{cfg}$ do

for all $i \in 1, \ldots, k$ do

if $\text{val}(e,i) \neq 0 \land i \neq k$ then

\$\$c[i-1] += \begin{cases} \text{val}(e,i) - \text{val}(e,i) \text{ if } f_s \lor (it < itr) = \text{true} \\ \text{val}(e,i) \text{ otherwise} \end{cases}$

else if $\text{val}(e,i) \neq 0$ then

\$\$c[i-1] += \text{val}(e,i)$

end if

end if

end for

end for

for all $k$-iteration profiled loop-backedges $e \in \text{cfg}$ do

if $(f_s)$ then $\text{IncrPathFreq}(c[itr])$

for $i := k-1, k-2, \ldots, 1$: $c[i] := c[i-1]$

$c[0] := 0$

\$\$e \equiv$

if $(itr < k-1)$ then {

\$\$itr += 1$

if $(itr = k-1)$ then $f_s := \text{true}$

}\$\$

end for

end for

for all $k$-iteration profiled loop-exit edges $e \in \text{cfg}$ do

\$\$c[0] := c[itr]$

for $i := 1, 2, \ldots, k-1$: $c[i] := 0$

$f_s := \text{false}$

\$\$itr := 0$

\$\$e \equiv$

end for

for all acyclically profiled loop-backedges $e \in \text{cfg}$ do

\$\$\text{IncrPathFreq}(c[0])$

\$\$c[0] := 0$

\$\$e \equiv$

end for

for all edges $e; [u \rightarrow \text{End}] \in \text{cfg}$ do

\$\$\text{IncrPathFreq}(c[0])$

\$\$e \equiv$

end for

end
For exit-edges of all k-iteration profiled loops, the partial path-identifier of the last valid path accumulator is transferred to c[0] — if the loop has been iterated i times, where i<k, then c[i+1] is loaded into c[0]; otherwise, c[k-1] is loaded into c[0] (this is because when the program is not executing in a loop, only c[0] is needed to accumulate the path identifier). The path accumulators c[1..k-1], the iteration counter, and the flag are cleared to make them ready for the next iteration.

The paths leading to the End node of the procedure is updated along the edges leading to the End node. For mixed-mode profiling, all acyclically profiled loop backedges have instrumentation code exactly as for the Ball-Larue scheme.

**Example 2.10.** Figure 2.7 shows the instrumented CFG for Figure 2.2 for two-iteration path profiling. Table 2.2 shows how some of the two-iteration paths are identified for the trace 1-(2-3-5-2-4-5)100-6.

The notation used in the figure is explained below:

- the flag \( f_i \) is shown simply as \( f \);
- \( c\leftarrow[a,b] \) implies \( c[0]+a; c[1]+b; \);
- \((a:b)\) refers to the selection: \( f_i \lor (id < itr) \ ? \ a : b \);
- the function \( \text{setf}() \) updates the values of both \( f_i \) and \( itr \) according to the following code segment:
  ```
  if (itr < k-1) then { itr += 1; if (itr = k-1) then f := true; }.
  ```
Table 2.2: Execution trace of the instrumented code.

<table>
<thead>
<tr>
<th>Edge</th>
<th>c</th>
<th>f</th>
<th>itr</th>
<th>Incr</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 → 2</td>
<td>[−13, 0]</td>
<td>false</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 → 3</td>
<td>[0, 0]</td>
<td>false</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 → 5</td>
<td>[0, 1]</td>
<td>false</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → 2</td>
<td>[0, 0]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 → 4</td>
<td>[18, 2]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 → 5</td>
<td>[18, 3]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → 2</td>
<td>[0, 18]</td>
<td>true</td>
<td>1</td>
<td>3</td>
<td>1-2-3-5-2-4-5</td>
</tr>
<tr>
<td>2 → 3</td>
<td>[13, 18]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 → 5</td>
<td>[13, 19]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → 2</td>
<td>[0, 13]</td>
<td>true</td>
<td>1</td>
<td>19</td>
<td>2-4-5-2-3-5</td>
</tr>
<tr>
<td>2 → 4</td>
<td>[18, 19]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 → 5</td>
<td>[18, 16]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → 2</td>
<td>[0, 18]</td>
<td>true</td>
<td>1</td>
<td>16</td>
<td>2-3-5-2-4-5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>4 → 5</td>
<td>[18, 16]</td>
<td>true</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 → 6</td>
<td>[15, 0]</td>
<td>false</td>
<td>0</td>
<td>15</td>
<td>2-3-5-2-4-5-6</td>
</tr>
</tbody>
</table>

For example, when the edge 5 → 2 is seen for the first time, no edge is recorded as the value of f was false (see the previous row of the table entry in Table 2.2). Hence, the algorithm was able to identify that the path seen till then, 1-2-3-5-2-4-5, is not a two-iteration path. The next time the edge is seen, the two-iteration path identifier 3 is identified and its frequency count incremented, which corresponds to the two-iteration path 1-2-3-5-2-4-5.

2.4 Mixed-Mode Profiling: Profiling different regions with varied length paths

The k-iteration profiling algorithm can be used in mixed-mode: different regions of the same procedure can be profiled with different values of k, thereby keeping the number of static paths in check — the “interesting” regions may be profiled using larger values of k while using acyclic profiling for the remaining procedure.

Mixed-mode profiling is also useful for nested loops: k-iteration profiling of nested loops needs multiple sets of edge weights for the inner loop — one of each iteration of the outer loop. Instead, one may use k-iteration profiling for the innermost loop and acyclic profiling for the outer loops. If the profiling information for an outer loop is desired, all the contained inner loops can be collapsed into single node — effectively reducing the outer loop into an innermost loop. Such schemes for profiling programs at various granularity is suggested in [27, 28].

Figure 2.8 illustrates mixed-mode profiling: it shows an example graph, its mixed-mode loop-unrolled DAG and the static paths recognised by the mixed-mode profiler.
Figure 2.8: Mixed-mode profiling: The loop $L_3$ is profiled with $k=1$ (acyclically), while loops $L_1$ and $L_2$ with $k=2$ (the backedge $6 \rightarrow 4$ occurs at most once in the paths while the backedges $3 \rightarrow 2$ and $5 \rightarrow 4$ occur twice in some of the paths).

It is easy to see why the path-numbering and path-identification algorithms work for mixed-mode profiling; we provide an informal argument using Figure 2.8(a):

- Consider disjoint loops that use different values of the parameter $k$: Assume the absence of the edge $6 \rightarrow 4$. Let $L_1$ use $k=k_1$ while $L_2$ use $k=k_2$. As the path-numbering algorithm numbers the paths in a bottom-up fashion, the paths from node 4 are still numbered correctly as the value $k_1$ does not influence it in any way. Similarly, numbering paths for the nodes of $L_1$ is only dependent on the number of paths through the node 4 (which is computed correctly by the argument above). Hence, all the nodes in $L_1$ are also numbered correctly.

- Consider nested-loops where the outer loop is acyclically profiled; Let $L_2$ be $k$-iteration profiled while $L_3$ is acyclically profiled. Figure 2.8(a) shows the loop-unrolled DAG for this case; the dummy edges $Start \rightarrow 4'$ and $6 \rightarrow End$ are added to take care of acyclic paths due to backedge $6 \rightarrow 4$. It is easy to see that the path-numbering algorithm would still be able to number all the paths in this DAG uniquely and perfectly.

The key idea in mixed-mode path-numbering is: while numbering a node $n$, we just need to know the number of paths through its successor nodes and not how the paths in the successor nodes were numbered.
2.5 Experimental Results

We have implemented acyclic and k-interation profiling for C source code using Lance [29]. Our implementation profiles each function in a given program, using separate tables to keep the path frequency counts for each procedure. The profiler attempts to use an array-based implementation for a path-frequency table if the number of static paths is low, but switches to a hashtable-based implementation if a procedure has more than 6000 static paths. It completely aborts profiling a procedure if the number of static paths in it exceeds 60,000. For nested loops it uses mixed-mode profiling: only the innermost loop is k-interation profiled while the outer loops are acyclically profiled. Functions with a solitary basic block are not profiled. Our implementation, both of the acyclic and k-interation path profilers, currently lacks the optimizations described by Ball and Larus [1], like optimizing the number of edges to be instrumented [30] and replacing the first increment of a counter by a load instruction (which also removes the initialization of counters). Our implementation also misses features like checking for overflow in the path-frequency table counters.

The cost of path profiling can be classified into two categories: computational cost (the overhead of the instrumentation code that computes the path-identifiers for each k-interation path traversed), and the table-update cost (the cost of updating the path-frequency table using the path-identifiers). Path profilers, generally, use an array-based implementation of the path-frequency table if the number of static paths is small; they switch to an expensive hashtable-based implementation whenever the number of static paths crosses a certain threshold. With increasing values of \( k \), the computational cost increases as a k-interation profiler has to maintain and update \( k \) counters to compute the path-identifiers. At the same time, as the first k-interation path is not formed till the \( k^{th} \) iteration of a loop (assuming that the loop executes for more than \( k \)-iterations), all iteration before the \( k^{th} \) iteration do not require a frequency-table update; this slight advantage, however, would be easily overwhelmed if the much larger number of “longer” static paths for a higher value of \( k \) forces the profiler to select a hashtable-based path-frequency table, while the instrumented programs for the lower values of \( k \) were enjoying an array-based path-frequency table. We use small programs [31] as benchmarks so that the frequency counts for all the three runs — for acyclic, two- and three-interation profiling — can be kept in an array (for most cases) for a comparison of the approximate computational overhead; we also show how fast the number of static paths grow in some of these programs, and instances where the use of hashtable slows down the programs significantly — to indicate the table-update cost.

Table 2.3 shows the program slowdown factors; the ratio of the instrumented program’s running time \( (ExcTime_{k-itr}) \) for collecting acyclic \((k-1)\) and k-interation \((k = 2, 3)\) profiles over that of the uninstrumented version \( (ExcTime_{original}) \)\(^4\). For most of the benchmarks, the slowdown from acyclic to

\(^4\)We report the results for an improved version of the tool over that reported in [25].
Table 2.3: Slowdown factor for k-iteration path profiling \((\text{ExecTime}_{k-\text{itr}}/\text{ExecTime}_{\text{original}})\).

<table>
<thead>
<tr>
<th>Program</th>
<th>(k = 1)</th>
<th>(k = 2)</th>
<th>(k = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>1.23</td>
<td>1.40</td>
<td>7.65</td>
</tr>
<tr>
<td>crc</td>
<td>1.15</td>
<td>1.36</td>
<td>1.60</td>
</tr>
<tr>
<td>edn</td>
<td>1.83</td>
<td>2.61</td>
<td>3.19</td>
</tr>
<tr>
<td>fdct</td>
<td>1.10</td>
<td>1.09</td>
<td>1.19</td>
</tr>
<tr>
<td>fibcall</td>
<td>1.06</td>
<td>1.79</td>
<td>3.75</td>
</tr>
<tr>
<td>fir</td>
<td>1.75</td>
<td>3.28</td>
<td>3.80</td>
</tr>
<tr>
<td>jfdctint</td>
<td>1.09</td>
<td>1.14</td>
<td>1.20</td>
</tr>
<tr>
<td>ludempp</td>
<td>1.65</td>
<td>2.16</td>
<td>9.57</td>
</tr>
<tr>
<td>ndes</td>
<td>1.45</td>
<td>11.96</td>
<td>---</td>
</tr>
</tbody>
</table>

Table 2.4: The table shows the number of static acyclic paths and k-Iteration paths for varying values of \(k\). \((\times\) indicates that path-numbering for the procedure was aborted as the number of static paths in the procedure exceeded 60,000).

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>BL</th>
<th>(k=2)</th>
<th>(k=3)</th>
<th>(k=4)</th>
<th>(k=5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>cl_hash</td>
<td>713</td>
<td>5887</td>
<td>44063</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td></td>
<td>writebytes</td>
<td>10</td>
<td>16</td>
<td>24</td>
<td>34</td>
<td>46</td>
</tr>
<tr>
<td>output</td>
<td>27</td>
<td>84</td>
<td>255</td>
<td>768</td>
<td>2307</td>
<td></td>
</tr>
<tr>
<td>ndes</td>
<td>des</td>
<td>416</td>
<td>1328</td>
<td>4064</td>
<td>12272</td>
<td>36896</td>
</tr>
<tr>
<td></td>
<td>ks</td>
<td></td>
<td></td>
<td></td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
<tr>
<td></td>
<td>cyfun</td>
<td>568</td>
<td>5318</td>
<td>35800</td>
<td>(\times)</td>
<td>(\times)</td>
</tr>
</tbody>
</table>

Two- and three-iteration profiling is small — indicating that the computational cost does not increase significantly from k-iteration to (k+1)-iteration profiling.

A few programs, however, show comparatively large slowdowns: \texttt{compress} and \texttt{ludempp} for three-iteration profiling, and \texttt{nades} for two-iteration path profiling. Table 2.4 gives a comparison of how the number of static paths increase for two of these programs. The program \texttt{nades} slows down significantly because the profiler switches to a hashable-based implementation of the path-frequency table for the function \texttt{cyfun} (14294 two-iteration static paths). As this function has more than 60,000 three-iteration static paths, the three-iteration profiler aborts profiling this function; hence, we do not provide any data for this case. For \texttt{compress}, the slowdown is very small for two-iteration path profiling but increases significantly for three-iteration profiling due to the same reason — the number of static paths in the function \texttt{compress} increases from 5887 two-iteration paths to 44063 three-iteration paths; thus requiring the hashable-based implementation for three-iteration profiles. The reason for the slowdown of \texttt{ludempp} is also same; the function \texttt{ludempp} in this program has 4144 two-iteration paths and 9110 three-iteration paths.

We have not spent much effort in optimizing the added instrumentation. Our implementation of the path-frequency tables, especially that of the hashable-based path-frequency table, is simplistic; we believe that it is possible to improve its performance. Due to the similarity in the operation of
acyclic and k-iteration profiling, most of the optimizations applicable to our implementation of acyclic path-profiling (for instance, an improved path-frequency table) are also valid for k-iteration profiling.

2.6 Conclusions

Ball and Larus proposed an algorithm to profile acyclic paths in a program efficiently; k-iteration path profiling is a generalization of the Ball-Larus algorithm, allowing paths spanning multiple loop iterations. Such longer paths would be useful in program understanding and profile-directed compiler optimizations. The k-iteration profiling algorithm allows mixed-mode profiling: profiling different parts of the same procedure with different path lengths, allowing the user to focus on the interesting regions, while keeping the profiling cost low. A base profiler can be used to indicate hot regions of a procedure; k-iteration profiling can then be used on these hot regions to generate better information.
Chapter 3

The Hot Path SSA Form

3.1 Introduction

The Static Single Assignment (SSA) form has been one of the most authoritative contributions to program compilation. Acknowledged as one of the most successful program representations, the SSA form has been a stable foundation for a couple of decades of work on program analyses and optimizations. With a recently surge of interest on speculative optimizations, one would be enticed to maneuver this huge number of efficient SSA-based algorithms for the speculative world — only if such an SSA-like form was available for writing such optimizations as well.

We propose to extend the power of the SSA form to path-profile guided speculative optimizations by separating the hot use-def chains from the cold ones, thus allowing a speculative optimizer to “see” only the most-likely dataflow facts. However, the ‘non-speculative’ SSA form is not lost: a traditional optimizer can still choose to constrain itself to the non-speculative form by ignoring the speculative information. The SSA form is not erased — just suitably extended with speculative information — obviating the necessity of constructing and maintaining the non-speculative SSA form separately; at the same time, this SSA-like intermediate form is much more amenable to speculative analyses and optimizations.

We call this extension to the SSA form as the “Hot Path SSA (HPSSA) form”. As the HPSSA form honours the constraint imposed by the SSA form (that of a single reaching definition for every use), many of the SSA-based algorithms for traditional optimizations developed over the last couple of decades (almost) immediately become available to speculative optimizers.

Following are our contributions in this chapter\(^1\):

\begin{itemize}
  \item We propose a novel program representation — the Hot Path SSA (HPSSA) form — that allows a
\end{itemize}

\(^1\)An earlier version of this work appeared in [26].
use to witness only the “more-likely” reaching definitions (section 3.3);

- We present an algorithm for constructing the HPSSA form (section 3.4);

- We demonstrate the potency of the HPSSA form by designing the analysis phase of a novel speculative optimization — Speculative Sparse Conditional Constant Propagation (SSCC) — that identifies both “safe” (expressions that are sure to be constants) and “speculative” (expressions that are more-likely to be constants) constants in a given program. An almost trivial extension of Wegman and Zadeck’s SCC algorithm [32], SCC exhibits the possibilities of developing new speculative optimizations using the HPSSA form by tailoring of existing SSA-based traditional optimizations (section 3.5).

Figure 3.1 and 3.2 compare the various representations of a program: the control-flow graph (not in SSA form), the SSA form and the HPSSA form.

3.2 Preliminaries

In this section, we establish a few terms and notations that we use in the rest of the paper.

3.2.1 Thermal States of Program Entities

Definition 3.1. Hot/Cold Paths

A program path $p : n_1 \leadsto n_2$ is said to be hot (cold) if the sequence of edges from node $n_1$ to $n_2$ appears (does not appear) in any profiled path that occurs frequently in the program profile.

The above definition has been intentionally slightly ambiguous to make it general enough to encompass various profiling and hot path selection schemes. The phrase “profiled path” implies any sequence of basic-blocks that is collected by a control-flow profiler; for instance, the “profiled path” is an edge for an edge profiler, an acyclic path for a Ball-Larus path profiler, and a path spanning multiple loop iterations for a k-iteration profiler [3, 25]. In this paper (and our implementation), a “profiled path” refers to intraprocedural acyclic paths, profiled using a Ball-Larus profiler. The qualifier “frequently” in the above definition depends on the hot path selection scheme: we may select hot paths by a threshold frequency, or pick a finite number of the most commonly executed paths from each procedure.

Definition 3.2. Temperature ($\theta$) of a node (edge) is said to be:

- hot: if the node (edge) is present on a hot path;

- cold: if the node (edge) is not present on any hot path.
Figure 3.1: The Static Single Assignment Form (Hot acyclic paths: $p_1$:abfgi; $p_2$:acdfgi; $p_3$:acefhi).

A backedge $b$ in a flow-graph is marked hot if, either of the dummy edges, $\delta_{\text{start}}$ to a loop-header $h$ or $\delta_{\text{end}}$ from a loop-tail $t$, is hot\footnote{The Ball-Larus profiler converts a flow-graph with cycles into a directed acyclic graph (DAG) by adding dummy edges, $\delta_{\text{start}}/\delta_{\text{end}}$, to and from the backedge source/target (respectively) for each loop in the program [1].}; this is understandable, as any control-flow through a dummy edge
reported by the Ball-Larus profiler indicates a control-flow through the corresponding backedge in the program flow-graph.

We will use the notation $\theta(n)$ to denote the temperature (hot/cold) of a program entity (nodes, edges or paths). The predicates $\theta_h(n) / \theta_c(n)$ denote that the entity $n$ is hot/cold.

For example, in Figure 3.1, all the nodes and edges are hot; the path $c \rightarrow d \rightarrow f \rightarrow g$ is hot (due to the path $p_2$) while the path $e \rightarrow f \rightarrow g$ is cold.

**Definition 3.3.** Hot/Cold Reaching Definitions and Definition Chains

A definition $\delta$ at a basic-block $n_1$ is said to reach a respective use at a basic-block $n_2$ **hot** if there exists a hot path from $n_1$ to $n_2$, and $\delta$ is not killed along that path. A definition $\delta$ at a basic-block $n_1$ is said to reach a respective use at a basic-block $n_2$ **cold** if there does not exist a hot path from $n_1$ to $n_2$, and $\delta$ is not killed at least along one cold path from $n_1$ to $n_2$.

Consider Figure 3.1(b): treating a $\phi$-function not as a definition, but as a label to the set of definitions in its argument set, we can see that though the meet-over-all-paths reaching definition set at $g_3$ is $\{x_{18}, x_{17}, x_7\}$, the definition $x_{18}$ does not reach $g_3$ along any hot path. So, $x_{18}$ is a cold reaching definition at $g_3$, while $x_7$ and $x_{17}$ are the hot reaching definitions (reaching the node $g$ along the paths $p_1$ and $p_2$). In the SSA form, the $\phi$-functions can be seen as creating a **definition chain**, that is broken only by a non-$\phi$ definition: $x_7 \rightarrow x_9$ and $x_{17} \rightarrow x_9$ are the **hot reaching definition chains** at $g_3$.
Figure 3.3: The HPSSA form of a program with loops (Hot acyclic paths: \(p_1:bc; p_2:bdgh; p_3:abdh\)).

\(x_{18} \rightarrow x_9\) is a cold reaching definition chain. In the HPSSA form, the \(\tau\)-functions “block” the cold reaching definition chains: for example, in Figure 3.1(b), \(x_{18} \rightarrow x_9\) no longer reaches \(g_3\) as it is killed by the \(\tau\) statement at \(g_1\).

### 3.2.2 The structure of profiled acyclic paths

The set of acyclic paths can be grouped by the node they initiate from — the program entry or a loop header; we refer to this node as the incubation node for the acyclic paths originating from it. In Figure 3.3, node \(a\) is the incubation node for \(p_3\), while \(b\) is the incubation node for \(p_1\) and \(p_2\).

A set of profiled acyclic paths \(\{p_1, p_2, \ldots, p_n\}\) entering a node \(u\) are said to be buddies at \(u\), if the paths \(p_1, p_2, \ldots, p_n\) have seen exactly the same sequence of edges from their incubation node; the group of all buddies at a node \(u\), from an incubation node \(s\), is referred to as BuddySet\(_s\)(\(u\)). Consider Figure 3.1(b) with the following set of hot paths:

\[
\begin{align*}
  p_1: & \ a-b-f-g-i; & p_2: & \ a-c-d-f-g-i; & p_3: & \ a-c-e-f-h-i; & p_4: & \ a-c-e-f-g-i; & p_5: & \ a-b-f-h-i.
\end{align*}
\]

\(\text{BuddySet}_s(f) = \{\{p_1, p_3\}, \{p_2\}, \{p_3, p_4\}\}\); i.e. \(p_1\) and \(p_5\) are buddies, so are \(p_3\) and \(p_4\), while \(p_2\) has no buddy at \(f\).
Chapter 3. The Hot Path SSA Form

Notations

Let us define a few notations to ease the following discussion:

- $Paths(u)$: The set of all profiled “hot” acyclic paths reaching the node $u$.
- $Paths_s(u)$: The set of all profiled “hot” acyclic paths reaching the node $u$ that initiate from the incubation node $s$.
- $Paths_s(u \rightarrow v)$: The set of all profiled “hot” acyclic paths reaching the node $u$ that initiate from the incubation node $s$ and progress along the edge $u \rightarrow v$ from $u$; without the subscript $s$, it denotes paths from all incubation nodes that progress along $u \rightarrow v$.
- $S(u)$: Set of all incubation nodes in the set of all profiled “hot” acyclic paths reaching node $u$.
- $N(\alpha)/E(\alpha)$: Set of all nodes/edges in the path $\alpha$.
- There can possibly be multiple paths between two nodes $u$ and $v$. We refer to the set of all such sub-paths initiating at $u$ and terminating at $v$ as $SubPaths(u, v)$. The notation $u \leadsto v$ is used to denote one specific path from $u$ to $v$. The term path is used to denote any sequence of basic blocks, as well as, acyclic hot paths identified using the Ball-Larus profiler; whenever it is not clear from the context, we refer to the latter as profiled paths. Note that a sub-path of a profiled path is always a hot path.

3.3 The Hot Path SSA (HPSSA) form

A speculative optimizer needs to identify “highly likely facts” — facts propagated along frequently executed paths — to perform optimizations that, though not legal on all static paths, “mostly” benefits the program. The HPSSA form uses a novel construct — the $\tau$-function — to “filter” definitions along cold paths, allowing only hot definitions to propagate further. The form of a $\tau$-statement is shown below:

$$x_{out} = \tau(x_0, x_1, \ldots, x_n)$$

The $\tau$-function argument list contains two types of arguments:

- **Safe (or non-speculative) argument**: The first argument, $x_0$, is the safe argument. It carries the variable version that needs to be assigned to $x_{out}$ to perform safe analyses and optimizations over the program.

- **Speculative arguments**: The rest of the arguments, $x_1 \ldots x_n$, are the speculative arguments, carrying the variable versions that reach the current node along the frequently executed paths; a
Chapter 3. The Hot Path SSA Form

speculative optimizer can treat the definition of $x_{out}$ as the union of these speculative arguments to perform speculative analyses and optimizations over the heavily executed paths.

The $\tau$-function can be seen as a conditional $\phi$-function:

$$
\tau(x_0, x_1, \ldots, x_n) = \begin{cases} 
\phi(x_0) & \text{safe interpretation} \\
\phi(x_1, \ldots, x_n) & \text{speculative interpretation}
\end{cases}
$$

If a program is in the Hot Path SSA form, then,

- each use of a variable is reachable by a single definition;
- if the safe interpretation of the $\tau$-function is used, each use of a variable is reachable by the meet-over-all-paths reaching definition chains;
- if the speculative interpretation of the $\tau$-function is used, each use of a variable in a hot basic-block is reachable only by the meet-over-hot-paths reaching definition chains (or the meet-over-all-paths reaching definition chains, if the use is not reachable from any meet-over-hot-paths reaching definition chain).

With the speculative interpretation, the set of reaching definition chains at even a cold basic-block might be smaller than that corresponding to the meet-over-all-paths, as some of the definition chains may be "blocked" by $\tau$-functions on their way to the cold node.

Each speculative argument $x_i$ in a $\tau$-function is mapped to the set of hot profile paths along which the definition corresponding to $x_i$ is reached. Consider Figure 3.2: at the statement at $g_1$, the $\tau$-function allocates the parameter $x_7$ corresponding to the path $p_1$, and the parameter $x_{17}$ for the path $p_2$; at the statement at $g_2$, it allocates only one parameter ($y_4$) corresponding to both $p_1$ and $p_2$ as the same definition (from statement $a_2$) reaches it along both the paths.

The HPSSA form savors the primary attribute of the SSA: each use is reachable by a single definition — encouraging the development of speculative extensions of existing SSA-based algorithms on the HPSSA form.

3.4 Constructing the HPSSA Form

In this this section, we discuss the construction of the HPSSA form. The original program (not in SSA form) is transformed into HPSSA form in four steps:

- **Insertion of $\phi$-statements**: The classic algorithm on constructing the minimal SSA form [2] places $\phi$-statements at the iterated dominance frontier of each definition in the program. A node
\( v \) is said to be in the dominance frontier of another node \( u \), if, and only if, \( u \) does not dominate \( v \) while a predecessor of \( v \) is dominated by \( u \).

- **Insertion of \( \tau \)-statements:** For each variable \( x \), we identify program points that necessitate a \( \tau \)-function, and, at all such points, insert a definition of the form \( x = \tau(x) \) (discussed in detail in section 3.4.1).

- **Variable renaming:** The definitive variable renaming algorithm [2] uses a variable renaming stack to propagate reaching definitions by traversing the basic-blocks over the dominator tree. The correctness of our algorithm requires a depth-first traversal over the dominator tree. Note that this phase also renames the sole argument in the inserted \( \tau \)-functions to the variable version corresponding to the meet-over-all-paths “safe” reaching definition.

- **Allocation of arguments to the \( \tau \)-functions:** Finally, we allocate the speculative arguments to the \( \tau \)-functions in correspondence to the hot reaching definition chains (discussed in detail in section 3.4.2).

Note that after step 3, the program is in SSA form; after step 4, it is in HPSSA form. We have intentionally kept the phases for building the SSA form (steps 1 and 3) clearly distinct from the steps required for constructing the HPSSA form (steps 2 and 4) to apprise the essentials of the HPSSA construction algorithm. It will be apparent that the phases need not be separate — some of them can be combined in an efficient implementation.

Figure 3.4 exemplifies the construction of the HPSSA form.

### 3.4.1 Thermal Frontiers: Placing \( \tau \)-functions

We call definitions due to \( \phi \) and \( \tau \)-functions as pseudo definitions, differentiating them from other concrete definitions; the corresponding statements are called pseudo/concrete statements. The visible definitions for a basic-block \( u \) comprises of the last definition for each variable in the block; these definitions are the only ones that are “seen” by the basic-blocks reachable from \( u \). In the following discussion, a reaching definition would refer to only concrete definitions; pseudo reaching definitions can be seen as the set of concrete definitions that were “merged” due to a \( \phi \)- or a \( \tau \)-function.

Each definition \( x := \ldots \) in the program can potentially lead to the insertion of a \( \tau \)-statement for variable \( x \). In a basic-block, a \( \tau \)-statement is inserted after all the \( \phi \)-statements (if any), before any of the concrete statements.

The \( \phi \)-functions act as definition mergers — “merging” multiple definitions into a single one. Comparably, the \( \tau \)-functions act as definition filters — separating hot definitions from cold ones, which were merged by previously occurring \( \phi \)-functions. Hence, a node \( n \) will need a \( \tau \)-function for a variable \( v \) if, and only if, both a hot and a cold reaching definition for the variable \( v \) arrive at \( n \).
(a) The original program that is intended to be cast into the HPSSA form.

(b) **Step 1:** The $\phi$-function is inserted at node $e$, which happens to be the dominance frontier of nodes $b$, $d$ and $c$.

(c) **Step 2:** The nodes $f$ and $g$ are identified as necessary points that need $\tau$-functions.

(d) **Step 3:** Variable renaming. Note that the first (safe) argument in the $\tau$-function gets renamed appropriately.

(e) **Step 4:** Speculative arguments are allocated to the $\tau$-functions.

**Figure 3.4:** Constructing the HPSSA Form.
The minimal SSA construction algorithm uses an exquisite structure — the Dominance Frontier — to insert the \( \phi \)-statements. To build the HPSSA form, we identified a similar structure to place the \( \tau \)-statements: the Thermal Frontier.

**Definition 3.4. Thermal Frontier**

A node \( v \) is said to be in the Thermal Frontier (TF) of a reaching definition \( d \), where \( d \) is defined at a node \( u \), \( (v \in TF(u, d)) \), if, and only if, the node \( v \) is also exposed to a reaching definition \( d' \), defined at a node \( w \) (\( w \) not dominated by \( u \)) such that \( \theta(d) \neq \theta(d') \), and none of the predecessors of \( v \) satisfy the above properties for \( d \) and \( d' \).

Stated informally, a node \( v \) is in the thermal frontier of a hot (or cold) reaching definition \( d \) (defined at \( u \)), if a different cold (or hot, respectively) definition \( d' \) meets \( d \) at the node \( v \).

Unlike Dominance Frontiers, Thermal Frontiers need not be join nodes. For example, in Figure 3.1(a), node \( g \in TF(b, b_1 : x = 1) \) as \( b_1 \) is a hot reaching definition (along \( p_1 \)), and \( g \) is also reachable by the cold reaching definition \( e_1 \).

It is apparent that \( \tau \)-functions for a definition \( d \) at a node \( u \) will be needed at the iterated \( TF(u, d) \). We define the **Iterated Thermal Frontier** in exactly the same way as iterated join and iterated dominance frontier were defined by Cytron et al. [2].

**Definition 3.5. Iterated Thermal Frontier (ITF)**

Let \( \gamma_x(u) \) return the visible definition of the variable \( x \) in the basic-block \( u \); then, for a set of nodes \( \kappa \), the Iterated Thermal Frontier (ITF) is the limit of the increasing sequence of sets of basic-blocks:

\[
TF^x(\kappa) = \bigcup_{u \in \kappa} TF(u, \gamma_x(u)) \\
TF_i^x = TF^x(\kappa) \\
TF_{i+1}^x = TF^x(\kappa \cup TF_i^x) \\
ITF^x = TF^x_\infty, \text{ where } TF^x_\infty \text{ refers to the fixpoint, i.e. when } TF_i^x = TF_{i+1}^x
\]

However, as the \( \phi \)-statements are inserted by a prior phase, placing the \( \tau \)-functions does not require fixpoint computation: a simple topological traversal over the CFG nodes suffices. Fixpoint computation is generally required if dataflow information can change after propagating through a backedge. While placing the \( \tau \)-functions, if a \( \tau \)-statement for a variable \( x \) is inserted in the header \( h \) of a loop due to a definition in the loop body (the only case that requires fixpoint computation), then, the loop-header \( h \) is sure to contain a \( \phi \)-statement (as no node in the loop-body can dominate \( h \)). Hence, if the CFG nodes are processed in the topological order, insertion of \( \tau \)-functions at the required nodes due to the definition of the variable \( x \) at \( h \) would have already happened.

For a set of visible definitions of a variable \( x \) at a set of nodes \( \kappa \), \( \tau \)-statements would be required at the Iterated Thermal Frontier \( ITF^x \) for variable \( x \). The following lemma states the necessary condition for computing the set of Thermal Frontiers.
Lemma 3.6. A node \( n \in TF(u, d^z) \) for a definition \( d^z \) (of a variable \( x \)) if

- Condition I: \( n \) is the junction of a hot and a cold path, i.e., paths at different temperatures meet at this node;
- Condition II: \( n \) is reachable by at least two different definitions of the variable \( x \).

Proof. If condition I fails, a \( \tau \)-function is unnecessary as \( n \) can then be reachable by only hot or only cold definitions of \( x \). If condition II fails, a \( \tau \)-function is again unnecessary as the node is then dominated by a definition of \( x \).

However, note that the above lemma is not a sufficient condition: a node \( v \notin TF(u, d^z) \) if the same definition \( d^z \) reaches \( v \) via both a hot and cold path (satisfying condition I), while \( v \) is also reachable by a different hot definition (of \( x \)), \( d' \), along a separate hot path (satisfying condition II). Hence, the above lemma may identify spurious Thermal Frontiers: our HPSSA algorithm inserts \( \tau \)-function templates at all points identified by the lemma, leaving the task of weeding out unnecessary \( \tau \)-statements to the \( \tau \)-argument allocation phase (section 3.4.2). In the rest of the discussion, we denote the set of Thermal Frontiers computed according to Lemma 3.6 as \( TF(u, d) \), and denote the ideal set of Thermal Frontiers (as defined in Definition 3.4) as \( TF_{ideal}(u, d) \).

Let us now sketch an algorithm for computing the Thermal Frontier of a node: we first identify certain nodes that are “junctions” of hot and cold paths (we call them Caloric Connectors), and thus, satisfy the first condition of Lemma 3.6; we then identify a scheme for satisfying the second condition.

3.4.1.1 Caloric Connector

Definition 3.7. Caloric Connector (CC)

A node \( n_{cc} \in CC \) if, for distinct nodes \( n \) and \( n' (n \neq n') \), there exist paths \( n \rightsquigarrow n_{cc} \) \( n' \rightsquigarrow n_{cc} \) such that \( \theta(n \rightsquigarrow n_{cc}) \neq \theta(n' \rightsquigarrow n_{cc}) \), and for all nodes \( n'' \in (N(n \rightsquigarrow n_{cc}) \cap N(n' \rightsquigarrow n_{cc})) \setminus \{n_{cc}\}, n'' \notin CC \).

In other words, a node \( n_{cc} \) is a Caloric Connector in a given graph (for a given set of hot paths) if there exist distinct nodes \( n \) and \( n' \), such that \( n \) and \( n' \) can reach \( n_{cc} \) through paths having different temperatures, and \( n_{cc} \) is the first common node in \( n \rightsquigarrow n_{cc} \) and \( n' \rightsquigarrow n_{cc} \) satisfying these properties. Simply put, caloric connectors are join points of hot and a cold paths.

Consider Figure 3.1(b): the node \( g \) is a Caloric Connector as the path \( d \rightarrow f \rightarrow g \) is hot while \( e \rightarrow f \rightarrow g \) is cold, while both the “predecessor” paths \( d \rightarrow f \) and \( e \rightarrow f \) are hot.

We make the following observations:

Lemma 3.8. For some incubation node \( s \), a hot acyclic path \( s \rightsquigarrow u \) extended by a forward edge \( u \rightarrow v \) forms a cold path \( s \rightsquigarrow u \rightarrow v \) if, there exists a set of buddy paths \( B \in BuddySets(u) \), such that, the buddies \( p \in B \) contain the path \( s \rightsquigarrow u \) but none of \( p \in B \) traverse the edge \( u \rightarrow v \).
Proof. Consider the set of buddy paths \( B \in \text{BuddySet}_s(u) \) such that the buddies \( p \in B \) contain the path \( s \rightarrow u \), but no path \( p \in B \) selects the edge \( u \rightarrow v \). Each set of buddy paths at \( u \) represents a unique sequence of edges from \( s \) to \( u \), distinct from any other set of buddies. Hence, as no path \( p \in B \) selects the edge \( u \rightarrow v \), the particular sequence of edges \( s \rightarrow u \rightarrow v \) is surely missing among the hot paths reaching \( v \). This implies that the path \( s \rightarrow u \rightarrow v \) is cold. \hfill \Box

Lemma 3.9. If there exists an cold acyclic path \( s \rightarrow u \rightarrow v \), \( s \rightarrow u \in \text{SubPaths}(s, u) \) (where \( s \) is an incubation node), then, either

- \( s \rightarrow u \) is cold, or
- \( s \rightarrow u \) is hot, and \( \exists B \in \text{BuddySet}_s(u) \), such that none of the buddies \( p \in B \) traverses \( u \rightarrow v \).

Proof. Let us consider the cases separately:

Case I: \( \theta_c(s \rightarrow u) \): As a path \( s \rightarrow u \) is already cold, extending it by an edge is still going to keep it at the same temperature.

Case II: \( \theta_h(s \rightarrow u) \): Attempting a proof by contradiction, let us assume that \( \forall B \in \text{BuddySet}_s(u) \), at least one of the buddies \( p \in B \) traverses \( u \rightarrow v \).

Let \( s \rightarrow u \rightarrow v \) be the cold path that "materializes" at \( v \), proceeding from the hot path \( s \rightarrow u \) along the edge \( u \rightarrow v \).

As \( s \rightarrow u \) is hot, there surely exists a set \( B \in \text{BuddySet}_s(u) \), such that, the buddies \( p \in B \) contain the path \( s \rightarrow u \). Our assumption (that at least one profiled path from each element of \( \text{BuddySet}_s(u) \) is assumed to traverse \( u \rightarrow v \)) confirms the existence of at least one profiled path \( p \in B \) that reaches \( v \) along the edge \( u \rightarrow v \). Now, because \( p \) contains the sequence of edges \( s \rightarrow u \rightarrow v \), the path \( s \rightarrow u \rightarrow v \) cannot be cold. Hence, our assumption must be incorrect. \hfill \Box

Figure 3.5 illustrates the above lemmas: Let \( a \) be the incubation node for the set of paths \( p_1, p_2 \) and \( p_3 \); consider the temperature of the path \( p_f = abcd\). In Figure 3.5(a), As none of the buddies at node \( d \) (paths \( \{p_1, p_2, p_3\} \)) select the edge \( d \rightarrow f \), the path \( p_f \) materializes as a cold path. On the other hand in Figure 3.5(b), even though a single path \( (p_1) \) among the buddy paths selects node \( f \), the path \( p_f \) remains hot.

The algorithm for computing the set of Caloric Connectors (Algorithm 3.1) is targeted at identifying if both a hot and a cold path can reach a node. Iterating through all nodes in the CFG in topological order, for each node \( u \), the algorithm examines the temperature of each outgoing edge \( u \rightarrow v \). It decides on the existence of a hot and/or a cold path at \( v \), and sets the flags \( \text{hasHotPath} \) and \( \text{hasColdPath} \) accordingly. A node \( v \) is marked as a Caloric Connector if it has both a hot and a cold path reaching it.
Algorithm 3.1 Computing the set of Caloric Connectors.

Traverse each node \( v \) in the graph (in the topological order) in the following manner:

1. Initialize \( \text{hasAColdPath} \) and \( \text{hasAHotPath} \) to \( false \).

2. For all edges \( e : u \rightarrow v \),
   - if \( \theta_c(u \rightarrow v) \), set \( \text{hasAColdPath} = true \);
   - if \( \theta_h(u \rightarrow v) \),
     - (a) Set \( \text{hasAHotPath} = true \);
     - (b) If \( e \) is not a backedge, and if, \( \exists B \in \text{BuddySet}_s(u) \) (for some incubation node \( s \)) such that \( B \) does not intersect \( \text{Paths}(u \rightarrow v) \), set \( \text{hasAColdPath} = true \).

3. If both \( \text{hasAColdPath} \) and \( \text{hasAHotPath} \) are \( true \), add \( v \) to the set of Caloric Connectors.

3.4.1.2 Computing Thermal Frontiers

For a concrete definition \( d \) and a basic-block \( v \in TF(u, d) \), the second condition of Lemma 3.6 is satisfied if \( v \) is in the dominance frontier of \( u \) (the node \( v \) is then also exposed to a different definition \( d' \) at a node \( w \) that is not dominated by \( u \)).

The case for pseudo definitions is slightly different: We identify a \( \phi \)-statement \( x_3 = \phi(x_1, x_2) \) not as a single definition, but as a set of concrete definitions \( \{ x_3 = x_1, x_3 = x_2 \} \) being propagated to
Algorithm 3.2 Inserting $\tau$-statements.

Process each control-flow graph node $v$ in the topological order as follows:

1. For all visible definitions $"d : x = \ldots"$ in the basic-block $v$,
   - if $d$ is a pseudo definition: for all $u \in CCC(v)$, add $u$ to $TF(v, d)$ if the pseudo definition $d$ is a reaching definition at $u$ ($d$ is not killed by concrete definitions at least along one path to $u$)
   - if $d$ is a concrete definition: $TF(v, d) = DF(v) \cap CC$.

2. For all $u \in TF(v, d)$, and for all visible definitions $"d : x = \ldots"$ in the basic-block $v$: if $u$ does not already have a $\tau$-function for $x$, insert a $\tau$-statement: $x = \tau(x)$ just after all $\phi$-statements (if any) at $u$, before any concrete statement.

all the outgoing paths from the definition-site; we also envision the $\tau$-statements similarly, but with only the speculative arguments\(^3\). As all paths from a pseudo definition $d$ defined at a node $u$ are now seen as carrying this set of definitions (instead of just $d$), the first Caloric Connector ($n_{cc}$) on each outgoing path from $u$, called the Closest Caloric Connectors of $u$ ($CCC(u)$), satisfies Lemma 3.6 — provided the pseudo-definition $d$ actually reaches $n_{cc}$.

Figure 3.6 illustrates the case when $d$ does not reach $n_{cc}$: Let $w \in CCC(u)$; however, $w \notin TF(u, d_1)$ as the pseudo-definition $d_1$ is “killed” by the concrete definition $d_2$ at $v$, making $d_2$ the dominating definition for $w$ — violating condition II of Lemma 3.6.

Algorithm 3.2 outlines our solution for inserting $\tau$-nodes.

3.4.2 Allocating $\tau$-function arguments

Before delving into the details of the algorithm, we take a slight digression into a deeper understanding of the $\phi$ and $\tau$ statements.

3.4.2.1 Understanding pseudo definitions

We view a pseudo definition — not as a new definition — but as a label to an existing set of definitions, namely, the definitions corresponding to its argument set. So, when we talk of reaching definitions in this section, we refer to all definitions (pseudo and concrete) that are not killed by a concrete definition; we do not allow pseudo definitions to kill an existing set of definitions. For example, in Figure 3.2, we

\(^{3}\)In the HPSSA construction algorithm, the hot definitions are “percolated” through the $\phi$ and $\tau$ statements as the percolated definitions may appear as arguments to future $\tau$-statements.
would say that the definitions for \( x_9, x_{17}, \) and \( x_7 \) are the set of hot definitions that reach \( g \); we call this set as the set of active definitions at \( g \). In the SSA form, as each definition corresponds to a unique version of the variable, we use the terms definition and variable version interchangeably.

A set of concrete definitions are merged at a pseudo definition. Let us define a set, \( \Upsilon(x_k) \), that represents the set of concrete definitions “contained” by the definition of a variable \( x_k \):

\[
\Upsilon(x_k) = \begin{cases} 
\{x_k\} & \text{if } x_k \text{ refers to a concrete definition} \\
\bigcup_{x_i \in \text{Arg}} \Upsilon(x_i) & \text{if } x_k \text{ refers to a pseudo definition}
\end{cases}
\]

where \( \text{Arg} \) is the set of all arguments for a \( \phi \)-definition, and the set of speculative arguments for a \( \tau \)-definition. For example, in Figure 3.2, \( \Upsilon(x_9) = \{x_7, x_{17}, x_{18}\} \) and \( \Upsilon(x_{11}) = \{x_7, x_{17}\} \).

We say that a set of definitions \( \{x_1 \ldots x_n\} \) covers a pseudo definition \( x_i \), if, and only if, \( \Upsilon(x_i) = \bigcup_{x_j \in \{x_1 \ldots x_n\}} \Upsilon(x_j) \). For example, in Figure 3.2, the definitions \( \{x_7, x_{17}\} \) cover \( x_{11} \).

3.4.2.2 The \( \tau \)-function Argument Allocation Algorithm

The algorithm, in essence, computes the path-sensitive active reaching definitions at each node \( u \) containing a \( \tau \)-function. The hot reaching definitions (variable versions) stand as arguments in the \( \tau \)-functions at \( u \), each definition mapped to the set of hot paths along which it reaches \( u \). A definition \( x_i \) that reaches \( u \) along the set of hot paths \( \xi_i \), can be used as a parameter for a \( \tau \)-function only if the following conditions are satisfied:

- if \( x_i \) is a concrete reaching definition: \( x_i \) can only be used as a parameter if \( \xi_i \neq \emptyset \) (i.e., \( x_i \) does reach \( u \) along some hot path);

- if \( x_i \) is a pseudo reaching definition: As discussed above, pseudo definitions are just labels to a set of concrete definitions. Even if \( \xi_i \neq \emptyset \), not all concrete definitions contained in \( x_i \) may be reaching \( u \): In Figure 3.2, the pseudo-definition \( x_9 \) reaches \( g_1 \) along the hot paths \( \xi_i = \{p_1, p_2\} \), i.e. \( \xi_i \neq \emptyset \). However, if \( x_9 \) is used as parameter for the \( \tau \)-function at \( g_1 \), it would invariably mean the inclusion of the definition \( x_{18} \), which is not a hot reaching definition at \( g \). Hence, a pseudo-definition can be used as an argument for some set of hot paths \( \xi_i \) if, and only if, all the concrete reaching definitions that it merges reaches \( u \) along \( \xi_i \). This condition can be ensured by checking if all the contained concrete definitions for \( x_i \) are available as active definitions at \( u \) for the set of paths \( \xi_i \).

Allowing definitions corresponding to pseudo-definitions in the \( \tau \)-function argument list requires tracking of both pseudo and concrete definitions (which might appear along intersecting set of paths), while ensuring that a pseudo definition never kills a concrete definition, even along the same path. For the sake of simplicity, we ignore all pseudo definitions and maintain only the concrete definitions as active
definitions (except if a pseudo-definition occurs as the only available reaching definition, or if a pseudo-definition is propagated along a backedge). So, pseudo-definition “labels” to a set of merged definitions can no longer appear in the \( \tau \)-function argument lists. The implication of ignoring the pseudo definitions is a larger argument list for the \( \tau \)-functions.

Instead of performing an expensive classical path-sensitive dataflow analysis, we designed an algorithm very similar to the variable renaming phase of SSA construction \cite{2} — using a variable renaming stack to maintain the active definitions (or renamed variables) reaching each node. Our algorithm is defined as a recursive procedure running over the dominator tree of the control-flow graph. The variable renaming stack maintains the set of active reaching definitions \( (x_i) \), along with the set of hot paths \( (\xi_i) \) that carry the definitions to the current node\(^4\). Our algorithm is more efficient than context-tupled classical path-sensitive dataflow analysis as it does not require storing of path-sensitive dataflow information at each basic-block.

Let \( P \) be the set of profiled acyclic path identifiers, and \( \text{DefPaths} \) be the set of \( P \). A frame in the variable renaming stack is a map \( \text{DefPaths} \rightarrow \text{Version} \), where \( \text{Version} \) is the renamed version of a variable; a frame can be seen as a set containing pairs \( \{[\xi_1, x_1], [\xi_2, x_2], \ldots, [\xi_n, x_n]\} \), where \( \xi_i \in \text{DefPaths} \). A variable renaming stack \( \text{VarStack}_x \) is a stack of frames for the base variable \( x \).

\( \text{VarStack} \) supports the following operations: \text{push}(\xi_i: \text{DefPaths}, x_i: \text{Version}, u: \text{Basic-block}) \) pushes a new frame with the association \( [\xi_i, x_i] \) on \( \text{VarStack}_x \); \text{pop}(u: \text{Basic-block}) \) pops off all frames that were pushed in the basic-block \( u \); and \text{top()} \) returns the topmost frame on the stack.

A Frame in \( \text{VarStack} \) supports the following operations: \text{get}(\xi: \text{DefPaths}) \) returns the version associated with \( \xi \) in the map; \text{accumulate}(\xi: \text{DefPaths}, x_i: \text{Version}) \) accumulates definitions: if a pair \( [\xi_j, x_i] \in \text{Frame} \), replace \( [\xi_j, x_i] \) by \( [\xi_j \cup \xi_i, x_i] \), else add a new association \( [\xi_i, x_i] \) to the frame.

The top of the variable renaming stack contains the set of active definitions — definitions that can be used to allocate arguments to the \( \tau \)-functions in the current basic-block. The algorithm traverses the control-flow graph recursively in a depth-first order over the dominator tree (as does the variable renaming phase for SSA construction); the set of dominators\(^5\) are traversed in the topological order of the nodes in the control-flow graph: the order is important to ensure that when a basic-block is processed, the definitions from all its incoming paths reach it. The active definitions are propagated via \( \text{VarStack} \) from a parent node to its children in the dominator tree; for a join node \( u \), the active definitions are accumulated (by a similar operation as \text{accumulate}(\xi: \text{DefPaths}, x_i: \text{Version}) \) for a frame) in a Definition Accumulator \( \Omega_x(u) \) from its predecessors in the CFG — it is loaded up on \( \text{VarStack} \) when the node \( u \) is processed.

The \( \tau \) argument-allocation algorithm is sketched in Algorithm 3.3.

\(^4\)The updates to \( \xi_i \) is done lazily; so a certain points, they may contain more paths than the actual set of hot reaching paths.

\(^5\)The children of a node \( n \) in the dominator tree are the dominators of \( n \).
Algorithm 3.3 A sketch of the $\tau$-function argument allocation algorithm.

Process a basic-block $u$ in the following manner:

1. Push the Definition Accumulator $\Omega(u)$ on VarStack (if $\Omega(u)$ exists).

2. If $u$ is the incubation node for a set of hot paths: for all base-variables $x$ that do not have a $\phi$-definition appearing in the basic-block $u$, push a frame $\langle \xi, x_i \rangle$, where $\xi$ is the set of all paths that incubate from $u$, and $x_i$ is the meet-over-all paths reaching definition (variable-version) for $x$ at $u$.

3. Process each statement $stm$ in the basic-block:

   (a) If $stm$ is a $\phi$-statement: if $u$ is a loop-header and the dummy profile edge $t \rightarrow \delta_{\text{end}}$ is hot (where $\delta_{\text{end}}$ denotes the dummy-end node for a Ball-Larus profiler, and $t$ is the corresponding loop-tail), accumulate $\langle \xi, x_i \rangle$ at the topmost frame of VarStack$_x$, where

      i. $\xi$ is the set of all paths that incubate from $u$, and
      ii. $x_i$ is the $\phi$-statement argument corresponding to the backedge $t \rightarrow u$.

   (b) If $stm$ is a $\tau$-statement:

      i. Create a set $C$ of candidate definitions from the definitions in VarStack$_x$.top(): for all $\langle \xi, x_i \rangle \in$ VarStack$_x$.top(), accumulate $\langle \xi \cap \text{Paths}(u), x_i \rangle$ in $C$, if, and only if, $\langle \xi \cap \text{Paths}(u) \rangle \neq \emptyset$;

      ii. If the definitions in the set $C$ do not cover the safe argument, i.e. $\text{defSet}(C) \not\subseteq \Upsilon(x_0)^u$, allocate arguments to the $\tau$-function in the following manner: for each $\langle \xi, x_i \rangle \in C$, add $x_i$ as an argument, mapping it to $\xi_i$; otherwise, replace the $\tau$-function $x_{\text{out}} = \tau(x_{\text{safe}}, \ldots)$ by a simple copy statement: $x_{\text{out}} = x_{\text{safe}}$.

   (c) Update VarStack to include new definitions in the basic-block $u$:

      • Concrete definition: Push the definition as a new frame associating it with Paths($u$);
      • Pseudo definition: Ignore.

4. Save the active definitions in $\Omega$ of the (forward) successors (if successor is a join node): for each forward (ignore backedges) successor edge $u \rightarrow v$, if $v$ is a join node, for each $\langle \xi, x_i \rangle \in$ VarStack$_x$.top() such that $\langle \xi \cap \text{Paths}(u \rightarrow v) \rangle \neq \emptyset$, accumulate $\langle \xi \cap \text{Paths}(u \rightarrow v), x_i \rangle$ in $\Omega_x$.

5. Recurse on the children of $u$ in the dominator tree in accordance to their topological order in the control flow graph.

6. Pop off all frames pushed by $u$ from VarStack.

---

*defSet($C$) returns the set of definitions, i.e., the first element of the pairs contained in $C$.

Let us understand the algorithm in three stages by increasingly making the control-flow graph more involved:

For an acyclic control-flow graph without control-flow joins (applicable algorithm steps 2, 3(b), 3(c), 5, 6)

For a tree-like control-flow graph (i.e. without control-flow joins and backedges), just the steps 2, 3(b), 3(c), 5 and 6 of the algorithm are enough to allocate arguments to the $\tau$-functions:
Step 2:

This step handles the case where a definition appearing outside a hot path \( p \) reaches a use in \( p \). Consider Figure 3.3: the definition to \( n_1 \) appears outside the hot path \( p_2 \). The algorithm, while processing the basic-block \( b \), does not find a φ-statement for the variable \( n \), and thus “inserts” a definition-pair \( \langle p_1, p_2, n_1 \rangle \) at the top of \( \text{VarStack}_n \), “allowing” it to now follow the path \( p_2 \). Hence, when the basic-block \( g \) is processed, the definition \( n_1 \) appears as an active definition at the top of the variable renaming stack associated with the path \( p_2 \), permitting its assignment as an argument to the \( \tau \)-function in the basic-block \( g \).

Step 3(b):

For each \( \tau \)-statement of a variable \( x \) at a basic-block \( u \), the algorithm examines the set of active definitions — contained in the topmost frame \( T \) in \( \text{VarStack}_x \) — to build up a set \( C \) of candidate variable versions to be used in the arguments: for each definition pair \( \langle \xi_i, x_i \rangle \in T \) with \( \xi_i \cap \text{Paths}(u) \neq \emptyset \), \( \langle \xi_i \cap \text{Paths}(u), x_i \rangle \) is added to \( C \). Note that \( \sum_{\langle \xi'_i, x_i \rangle \in C} \xi'_i = \text{Paths}(u) \).

If the definitions in \( C \) do not cover the safe argument, then, suitable arguments are allocated to the \( \tau \)-function in the following manner. The variable versions in \( C \) turn out to be the set of speculative arguments needed by the \( \tau \)-function under consideration: each \( x_i \) in \( \langle \xi'_i, x_i \rangle \in C \) is added as an argument to the \( \tau \)-function, mapped to \( \xi'_i \), the set of hot paths along which \( x_i \) reaches the current node.

If the definitions in \( C \) do cover the safe argument, the \( \tau \)-function at this point is extraneous, and can be pruned. Hence, the \( \tau \)-function \( x_{\text{out}} = \tau(x_{\text{safe}}, \ldots) \) is replaced by a simple copy statement: \( x_{\text{out}} = x_{\text{safe}} \).

Step 3(c):

The variable renaming stack is updated to include new active definitions: for each concrete definition, it pushes a new frame associating it with \( \text{Paths}(u) \). The intuition is that a new concrete definition effectively “kills” all active definitions on all basic-blocks dominated by the current block. Thus, the top of the stack now contains this new definition as the lone active definition for the given variable.

- If \( \text{stm} \) is a concrete statement, the algorithm pushes a new frame \( [\text{Paths}(u) \rightarrow x_i] \) onto \( \text{VarStack}_x \), thus making it the “active” definition.
- If \( \text{stm} \) is a φ or \( \tau \)-statement, ignore the definition as we do not allow pseudo definitions to enter the variable renaming stack (and thus prevent them from being allocated as parameters to \( \tau \)-functions).

Step 5:

The \( \tau \)-argument allocation algorithm processes the basic-blocks in a depth-first order over the dominator tree, with the set of dominatees at each node in the dominator tree traversed in the topological order of the nodes in the control-flow graph: this order is important for graphs with control-flow joins (discussed later).
Step 6:

The set of active definitions pushed by the current basic-block are popped (after its dominatees are processed), thus exposing the active definitions in the frame below: this step is exactly similar to that used by the SSA construction algorithm. After all the dominatees of the basic-block \( u \) have been processed, all the frames that were pushed in \( VarStack \) by \( u \) are popped off.

For an acyclic control-flow graph with control-flow joins (applicable algorithm steps 1, 4 in addition to those for an acyclic control-flow graph without control-flow joins i.e. steps 2, 3(b), 3(c), 5, 6)

For a control-flow graph with control-flow joins, the Step 5 of the algorithm ensures that the basic-blocks are traversed in a depth-first order over the dominator tree, the set of dominatees traversed in the topological order of the nodes in the control-flow graph. The order is important as a node must “accumulate” all the possible variable versions reaching it from all possible paths that reach it before it can attempt to rename the arguments of its \( \tau \)-functions.

Consider Figure 3.2: after the node \( b \) is processed, the top of \( VarStack_x \) contains all the variable versions that are “active” at this point, which are required to be propagated to the node \( f \). However, as \( f \) is not the immediate dominatee of \( b \), the algorithm will unwind the stack to \( a \) (due to step 6), thus losing all the computed “active” definitions at the end of \( b \). So, after processing a basic-block \( u_i \) for all the \( \phi \) or \( \tau \)-functions in \( v \) along edges \( u \rightarrow v \), Step 4 “accumulates” the current set of active definitions in a Definition Accumulator \( \Omega_x(v) : DefPath \rightarrow Version \) (for all variables \( x \)), using a primitive function \( accumulate() \) (exactly similar to Frame :: accumulate()) for each definition the frame (intersecting the respective path-components with \( Paths(u \rightarrow v) \)). Thus, for a control-flow join \( v \), \( \Omega_x(v) \) accumulates the set of all active definitions from each of its control-flow predecessor for use in its \( \tau \)-statements. When the basic-block \( v \) is processed, the algorithm pushes \( \Omega_x(v) \) as a new frame onto \( VarStack_x \), readying itself to use this set of gathered definitions for use in \( v \) (Step 1).

For an control-flow graph with control-flow joins and backedges (applicable algorithm steps 3(a) in addition to those for an acyclic control-flow graph with control-flow joins steps 1, 2, 3(b), 3(c), 4, 5, 6)

Though we use acyclic path-profiling, the algorithm is sensitive to reaching definitions propagating along backedges. Consider Figure 3.3: the HPSSA form identifies that \( i_3 \) is a hot reaching definition at the block \( e \) (which is reaching via the backedge), while \( i_1 \) is a hot reaching definition at the basic-block \( f \). This is achieved by the Step 3(a) of the algorithm: If a node \( u \) is a loop-header for some backedge \( t \rightarrow u \), and \( stm \) is a \( \phi \)-statement in \( u \), the algorithm accumulates the \( \phi \)-function argument corresponding to the backedge, associating it with all paths incubating from \( u \), on the topmost frame of the variable renaming
stack. This step is required as, when \( u \) is processed, the “active” definitions along the edge \( t \rightarrow u \) is not available to \( u \) as \( u \in Dominator(t) \), and a dominator is always processed before a dominatee.

We illustrate the algorithm via an example (Figure 3.8) for the flow-graph in Figure 3.2.

Let the basic-blocks be processed in the order \( a, b, c, d, e, f, g, h, i \).

The basic-block \( a \) is processed foremost: the algorithm (Step 3(c)) pushes the definition \( x_3 \) on \( VarStack_x \) (Figure 3.8(a)), and then recurses on the children of \( a \) in the dominator tree (Figure 3.7), namely \( b, c \) and \( f \) (Step 5). At the node \( b \), the algorithm (Step 3(c)) pushes the definition \( x_7 \) on the stack; its successor node, \( f \), turns out to be a join node: hence, the algorithm (Step 4) accumulates the definitions in the topmost frame of the stack into the (currently empty) definition accumulator \( \Omega_x(f) \) (Figure 3.8(b)). As \( b \) has no children in the dominator tree, the algorithm (Step 6) retraces the recursive path to node \( a \), popping off the definition pushed by \( b \) in the process. The variable renaming stack and the recursion stack (\( RecurStack \)) now again resemble that in Figure 3.8(a).

The nodes \( c, d, \) and \( e \) are processed similarly; Figure 3.8(c) shows the state of the data-structures just after node \( e \) is processed. After handling \( e \), the recursion is unwound to node \( a \).

The algorithm then picks the node \( f \): it first pushes the definition accumulator of \( f, \Omega_x(f) \), on the variable renaming stack (Step 1); on encountering the \( \phi \)-definition for \( x_9 \), it simply ignores the same (Step 3(c)). Finally, it recurses on the immediate dominates of \( f \), viz. \( g \) and \( h \) (Step 5).

The node \( g \) is processed next: on encountering the \( \tau \)-definition for \( x_{11} \), the algorithm (Step 3(b)) attempts to allocate arguments for the same: Examining the active definitions (top of the variable renaming stack), the algorithm attempts to assemble the candidate set \( C \) — a subset of definitions from the topmost frame of \( VarStack_x \) that, together, can map to all the hot paths passing through \( g \). The set of active definitions at \( g \) turn out to be \( \{[p_1, x_7], [p_2, x_{17}], [p_3, x_{18}]\} \). To be added to \( C \), the path-component in the definition pairs must intersect with \( Paths(g) = \{p_1, p_2\} \); \([p_1, x_7] \) and \([p_2, x_{17}] \) satisfy the condition, while \([p_3, x_{18}] \) does not. Notice how the cold definitions are pruned out from the possible set of definitions to be added as arguments to the \( \tau \)-function.

![Figure 3.7: The dominator tree of the CFG in Figure 3.1.](image-url)
Figure 3.8: Steps in the execution of the $\tau$-argument allocation algorithm.
As $\Upsilon(x_g)$ contains the definition $x_{18} \notin C$, we allocate arguments to the $\tau$-function from $C$.

\[
x_{11} = \tau(x_g, x_{17}(p_1), x_{17}(p_2))
\]

The algorithm then accumulates the active definitions in $\Omega_x(i)$ (Figure 3.8(e)). The nodes $h$, and then $i$, are processed in a similar manner.

Note that the set of candidate definitions $C$ for a $\tau$-function at a node $v$ contains the exact set of hot definitions that reach $v$. Additionally, for each pair $\langle \xi, x_i \rangle \in C$, $x_i$ reaches $u$ along the paths in $\xi$, and along no other hot path.

Now consider the control-flow graph with loops (Figure 3.3): Let us explicate as to how the definition of $i_3$ in the block $c$ is identified as a hot reaching definition at the $\tau$-function in the node $e$ even though we use acyclic path-profiles. As the loop-path $p_1$ is hot, when the node $b$ is processed, the definition-pair $\langle p_1, p_2, i_3 \rangle$ is added to the top of the variable renaming stack (being the parameter to the $\phi$-function corresponding to the backedge) by Step 3(a). When the algorithm recurses on the children of $b$ in the dominator tree, the variable renaming stack carries the definition to the basic-block $e$ (via the node $d$), where it is recognised as an argument for the $\tau$-function along the path $p_2$. The Step 2 in the algorithm is required to carry the meet-over-all-paths definition $n_1$ from the node $a$ to the node $g$, as there does not exist any acyclic hot path from $a$ to $g$.

### 3.5 Exiting the HPSSA form

Exiting the HPSSA form is extremely simple — a $\tau$-statement is replaced by a copy statement from the safe-argument to the defined variable:

\[x_{\text{out}} = \tau(x_0, x_1, \ldots, x_n) \quad \rightarrow \quad x_{\text{out}} = x_0\]

This puts the program in the SSA form; one can then use an out-of-SSA algorithm [2, 33, 34] to exit the SSA form.

### 3.6 Speculative Sparse Conditional Constant Propagation

We have implemented the analysis phase of a novel optimization algorithm, the *Speculative Sparse Conditional Constant Propagation (SSCC)*, on the HPSSA form. This optimization expands the scope of the SCC [32] algorithm — allowing it to identify speculative constants (expressions that are highly likely to be constants) — along with the conventional “safe” constants (expressions that are guaranteed to be constants).

This section is more than a description of a new analysis algorithm: through this algorithm, we essentially aim to demonstrate how new speculative optimizations can be developed on the HPSSA form by simple extensions of existing “safe” SSA-based optimizations.
The SSCC algorithm operates on a four level lattice (Figure 3.9 shows the SSCC lattice for integers): the conventional constant propagation lattice is extended by another layer — that of speculative constants (indicated by the constants superscripted with ‘s’). The speculative constants can be seen as constant values with exactly the same properties as that of ordinary constants — just marked “speculative” — indicating that they are predicted values, not guaranteed to hold under all executions.

The transfer functions of all existing operations (including that of the $\phi$-function) hold as in SCC, except for the fact that if any operand in an expression turns out to be a speculative constant, the result of the operation, if a constant, would be a speculative constant carrying the respective constant value. For example, $2 + 3^s$ would render the speculative constant $5^s$.

The transfer function for the $\tau$-functions is defined as follows (where $\sqcap$ is the meet operator): If the meet of all the arguments does not produce $\bot$ (not-constant), the transfer function resembles the transfer function for the $\phi$-functions. Even if the meet of all the arguments turns out to be $\bot$, there might still be the chance of the expression being identified as a speculative constant; let $\beta = x_1 \sqcap x_2 \cdots \sqcap x_n$. The transfer function attempts to return $\beta$, if $\beta$ is $\bot$, $\top$ or a speculative constant; if $\beta$ is a “safe” constant, $\beta$ moves in the lattice to $(\beta)^s$, the corresponding speculative constant. Formally, the transfer function for $\tau(x_0, x_1, \ldots, x_n)$ is given by the following (where each expression refers its abstract value in the lattice, and $\beta = x_1 \sqcap x_2 \cdots \sqcap x_n$):

$$
\tau(x_0, x_1, \ldots, x_n) \sqcap \left\{ \begin{array}{ll}
x_0 \sqcap \beta & \text{if } x_0 \sqcap \beta \neq \bot \\
\beta & \text{if } x_0 \sqcap \beta = \bot \text{ and } \beta \text{ is a safe constant} \\
(\beta)^s & \text{otherwise}
\end{array} \right.
$$

The meet with the current value of the $\tau$-function is added to ensure termination by enforcing monotonicity; otherwise, code fragments resembling that in Figure 3.10 will never reach a fixpoint due to $i_3$ increasing its value in the speculative domain, and the $\tau$-function feeding the same value back to it.
\[i_0 = 0;\]
while(...) {
    \[i_1 = \phi(i_0, i_3);\]
    \[i_2 = \tau(i_1, i_3);\]
    \[i_3 = i_2 + 1;\]
}

**Figure 3.10:** A case that requires meet with its old value in the \(\tau\)-statement transfer function for SSCC.

(a) The definitions of \(x_3\) and \(x_5\) form a dependency cycle. (b) The definition of \(x_5\) is self-referential.

**Figure 3.11:** Cases in the HPSSA form that need special handling.

The HPSSA form, however, allows certain cases that not possible in the SSA form; these possibilities need to be taken under consideration while designing speculative analyses over this form.

- **Definition dependency cycles**

Consider Figure 3.11(a): As the loop-entry path is not contained in the list of hot acyclic paths, the \(\tau\)-statement in the loop-header node \(b\) blocks the definition of \(x_1\). This causes an definition dependency cycle: the definition of \(x_3\) is dependent on the definition of \(x_5\), while the definitions of \(x_5\) is dependent on \(x_3\). If \(x_1\) turns out to be not-constant, \(x_3\) solely depends on \(x_5\), which then solely depends on \(x_3\) (as \(x_4\) also just depends on \(x_3\)).

In our implementation of SSCC, we adopt a simplistic approach: if the speculative argument list of an \(\tau\)-statement in a loop-header only contains definitions from inside the loop\(^6\), we consider the safe interpretation of that \(\tau\)-statement.

---

\(^6\)This can be ascertained from the temperature of the loop-entry edge: the edge from the loop-preheader to the loop header.
• **Self-referential \( \tau \)-definitions**

Consider Figure 3.11(b): The value of the variable \( x \) is modified inside the loop (in block \( d \)), causing a \( \phi \)-definition at nodes \( b \) and \( e \). However, the hot acyclic path through the loop happens to be the route that leaves \( x \) unmodified, causing a self-referential \( \tau \)-definition at node \( e \). The definition \( x_5 = \tau(x_4, x_1, x_5) \) implies that the frequent reaching definitions at this node is either from the loop-preheader \((x_1)\) or whatever was set by this statement in the previous iteration of the loop \((x_5)\).

In our implementation of SSCC, we do not propagate any modification to \( x_5 \) due to this statement to the “pending to be modified” list of statements. Simply put, after processing a statement \( x_1 = \tau(x_0, x_1, x_2, \ldots, x_n) \), we add all statements that involve \( x_1 \) as an operand to the SSA-worklist\(^7\), except the current (self-referential definition) statement.

### 3.7 Implementation and Experiments

We implemented our HPSSA construction algorithm, as well as the analysis phase of the SSCC algorithm on the Scale compiler [35]; we were also aided by the CIL [36] tool. We only cast scalar variables whose address has not been taken in the HPSSA form; \( \tau \)-functions are not introduced for the remaining variables. Also, we do not check for the containment relation to prune out extraneous \( \tau \)-functions. The SSCC algorithm implementation handles only integer variables; the implementation is interprocedural but context-insensitive; function pointers are ignored (it flags a warning, computing a possibly unsafe solution).

We tested our implementation on some programs from the SPEC CPU2000 benchmark suite [37]. We used a naive hot path selection criteria: all the acyclic paths executed on the \texttt{train} input set was considered “hot” for building the HPSSA form. Table 3.1 exhibits our findings for programs run on the \texttt{ref} input set. The programs were run with the default parameters, i.e., no parameters were set on the command line, either for training, or for the actual run (on the \texttt{ref} set). We collected statistics for \textit{dynamic uses} (use of a variable/expression during the actual run) for variables (\textit{Variable Uses}), and for sub-expressions that could be constant-folded speculatively (\textit{Expression Uses}). The uses are tabulated only for the speculative constants — uses that are likely (but not guaranteed) to be constants. We have not shown the number of “sure” constants as it would be the same as that for the original SCC algorithm. We also indicate the \textit{Hit Rate} (\textit{HitRt}): the percentage of uses where the use of variable/expression actually agrees with the “predicted” speculative constant value.

\(^7\)The SCC algorithm [32] maintains two worklists, the CFG-worklist and the SSA-worklist, to capture all statements that await processing.
Table 3.1: Speculative constants discovered by the SSCC algorithm (‘−’ indicates almost; \text{grp}, \text{prg}, & \text{src} refer to the graph, program & source inputs respectively).

<table>
<thead>
<tr>
<th>Program</th>
<th>Inpt</th>
<th>Variable Uses</th>
<th>Expression Uses</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Uses</td>
<td>HitRt</td>
<td>Uses</td>
</tr>
<tr>
<td>181.mcf</td>
<td>-</td>
<td>33110</td>
<td>100.00</td>
<td>49665</td>
</tr>
<tr>
<td>175.vpr</td>
<td>-</td>
<td>6938074</td>
<td>100.00</td>
<td>8110837</td>
</tr>
<tr>
<td>164.gzip</td>
<td>grp</td>
<td>26592</td>
<td>100.00</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>prg</td>
<td>17412</td>
<td>100.00</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>src</td>
<td>4721</td>
<td>99.98</td>
<td>5</td>
</tr>
<tr>
<td>197.parser</td>
<td>-</td>
<td>165970964</td>
<td>100.00</td>
<td>340</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>grp</td>
<td>132100650</td>
<td>100.00</td>
<td>938</td>
</tr>
<tr>
<td></td>
<td>prg</td>
<td>100819492</td>
<td>100.00</td>
<td>6576416</td>
</tr>
<tr>
<td></td>
<td>src</td>
<td>108134316</td>
<td>100.00</td>
<td>5256006</td>
</tr>
</tbody>
</table>

The programs seem to enshroud plenitude opportunities for an optimizer adept at performing speculative program transformations. Most of the programs show a large number of dynamic speculative usages with good hit rates (except 256.bzip2 for the sub-expression uses; still the overall hit rate turns out high, courtesy the variable usages). A more intelligent hot path selection scheme may be able to reap more constants, though it may also have an effect on the hit-rate; we are interested in experimenting with alternative schemes in the future.

### 3.8 The \( \tau \)-function as a generalized \( \phi \)-function

Though we have presented the \( \tau \)-operator as an additional operator for a cleaner presentation, the \( \tau \)-operator can essentially be seen as a generalized \( \phi \)-function. We show the generalized form of the \( \tau \)-operator below:

\[
x_{\text{out}} = \tau(\frac{x_0, x_1, \ldots, x_n}{\text{safe}}; \frac{x_{n+1}, \ldots, x_{n+m}}{\text{speculative}})
\]

The extended \( \tau \)-function allows for a set for the safe arguments; notationally, we separate the safe and the speculative argument set by a semi-colon. The \( \phi \)-function can be seen as a \( \tau \)-function for which the speculative argument set is empty. Such an extension is useful as it effectuates excision of instructions when a basic-block contains both a \( \phi \)- and a \( \tau \)- statement.

Figure 3.12 contrasts the basic and generalized HPSSA form: the \( \phi \)-statement in basic-block \( d \) is eliminated in the generalized HPSSA form.
3.9 Conclusions

We propose a novel extension to the highly successful SSA form, and demonstrate — by an analysis algorithm for Speculative Sparse Conditional Constant Propagation — that novel path-profile guided speculative optimizations can be enabled on the HPSSA form by almost obvious modifications of existing SSA-based traditional optimizations. We are pondering over the design of speculative versions of other existing SSA-based traditional optimizations — Global Value Numbering [38] and Partial Redundancy Elimination [39] being our foremost targets. We are also interested in extending the HPSSA form for richer profiles like the k-iteration path [Chapter 2] profiles.
Chapter 4

Related Work

4.1 Control-Flow Profiling

Ball and Larus [40] established that programs follow a very few of all possible static paths. In [1], they provide an efficient algorithm for capturing *acyclic* paths — paths “chopped” at backedges. Whole Program Paths [12] is an attempt to capture the whole program trace; as the full program traces tend to be enormous in size, they proposed the use of the SEQUITUR [41] algorithm to compress the trace. Various other algorithms to compress program traces has been proposed; for instance, Renieris et. al [42] use arithmetic coding for trace compression. However, despite the disadvantage of “forgetting” the correlations across loop-iterations, acyclic path profiling has remained popular due to its low-cost.

Ball et al. [43] provide an indicator as to when mere edge-profiles can serve as a good estimate for path-profiles. They provide algorithms for identifying *definitely* hot paths (a path *p* is deemed definitely hot if its frequency crosses a certain threshold for the flow configuration that allows the minimum flow through *p*) and *potential* hot path (a path *p* is deemed potentially hot if its frequency crosses a certain threshold for the flow configuration that allows the maximum flow through *p*) through a graph annotated with edge-profiles. They discover that the various hot-edges based hot path selection schemes generally yield good results, though there exist cases that necessitate path profiling.

Ball [30] suggested techniques for efficiently counting events in a program’s execution. The BL path-profiling [1] uses some of the techniques to drive a set of optimizations to reduce the profiler overheads. We intend to use these optimizations to improve the efficiency of our k-iteration profiling tool in the future.

Tallam et al. [3] used Ball-Larus paths to record “slightly” longer overlapping paths, and proposed an instrumentation algorithm to collect their frequencies; these overlapping paths were then used to estimate the frequencies of longer paths crossing loop-backedges and procedure boundaries — paths that
Figure 4.1: Overlapping Paths (OL) for estimating longer paths spanning loop iterations [3].

were more adept at identifying opportunities for optimizations. Their basic algorithm for estimating flow for paths spanning across loop-iterations is as follows:

- **Collect frequencies of overlapping paths**: Overlapping paths (OL) were created by extending the Ball-Larus paths. A \(k\)-overlapping path (OL-\(k\)) path would terminate at the \((k+1)\)\(^{th}\) predicate block following the backedge. Consider Figure 4.1: we only show predicate blocks, i.e. blocks making control-flow decisions (the curved arrows indicate paths comprising of only non-predicate blocks). The path \(B_1B_2B_4B_1\) is a OL-0 path, as it contains only one predicate block after the backedge; similarly, \(B_1B_2B_4B_1B_2\) and \(B_1B_2B_4B_1B_3\) are OL-1 paths, as they terminate at the second predicate block after the backedge.

- **Estimate Flow**: Using the frequency of the overlapping paths, the upper and lower bound on the frequencies of the interesting paths — the paths spanning two iterations of the loop — was estimated.

The cost of collecting the path frequencies was found to be around 4.2 times that of the Ball-Larus paths. The average imprecision in estimating the flows derived from overlapping path frequencies was found be to low — from -4% to +8%. Tallam et al. [3] also proposed an algorithm to collect frequencies of paths that cross procedure boundaries.

Understandably, path-profiling has evoked a lot of interest for JIT compilers. Targeted Path Profiling (TPP) [44], an attempt to reduce the overhead of collecting path-profiles, uses edge-profiles to reduce the number of potential paths by not enlisting paths with cold edges (allowing array accesses rather than expensive hashtable lookups), and skipping instrumentation of program regions wherever it is possible to unambiguously construct paths from an edge profile. Practical Path Profiling [45] improves upon

68
TPP, firstly, by cutting down on profiling instrumentation on cold paths and paths that are predicted well by an edge profile, and further, by additional techniques to push down the expense of the remaining instrumentation.

Structural Path Profiling (SPP) [28] is an interesting idea that drives a loop-structure based partitioning of a procedure into a hierarchy of the nested graphs; each nested graph is, then, profiled independently. The method has been proposed as an efficient online path profiling technique for JIT compilers. The k-iteration mixed-mode path profiling algorithm can be combined with SPP: the different partitions may be profiled with different path lengths (by varying k) depending on the hotness of the respective code section.

Hierarchical Path Profiling [27] uses the concept of program regions to drive profiling. The inner regions are collapsed to single nodes in the outer regions, and path profiles are collected separately for the different regions, which can be used by region-based compilers to drive local optimizations. Hierarchical \textit{k-iteration} profiling seems to be an interesting possibility that we intend to investigate.

There have been research solutions towards profiling a smaller, selective subset of the paths. Selective Path Profiling (SPP) [46] computes the values for probes in a manner, such that, that the sum of these values for any acyclic path in a selected subset is unique, but possibly with non-unique path sums for the remaining paths. Preferential Path Profiling (PPP) [47] is a recent attempt at profiling selective paths in a procedure, thereby reducing the profiling overheads; the authors prove that though perfect path numbering is not achievable for this problem, their solution performs well. Preferential \textit{k-iteration} profiling, i.e. profiling a subset of all the possible multi-iteration path, seems to be an useful option.

Path profiling has found multitude of applications from compiler optimizations to program testing. Young et al. [48] used path-profiles to improve Superblock scheduling [49], that previously relied on edge-profiles. Gupta et al. [7, 50] used path-profiles to devise algorithms for partial redundancy elimination (PRE) using speculation and partial dead code elimination (PDE) using predication. Ammons and Larus [51] proposed a novel analysis and optimization technique that benefits hot paths. Reps et al. [52] used path-profiles to locate program errors. Statistical Bug Isolation [53] uses statistical techniques for isolating bugs; Holmes [54], uses acyclic path profiles to drive statistical debugging. Richer control-flow profiles in the form of \textit{k-iteration} paths, by the virtue of capturing more correlation between executing entities, should be beneficial to the above applications.

4.2 The SSA Form: Extensions and Applications

The \textit{Hashed SSA form (HSSA)} [4] extends the SSA form, introducing new operators — the $\chi$ and the $\mu$ operators — to capture the \textit{may modify (MayDef)} and the \textit{may reference (MayUse)} behaviours due to indirect memory references. Consider Figure 4.2: the variable $p_3$ may be pointing to either $x_1$ or $y_1$;
the \( \chi \) annotations suggest that \( x_1 \) and \( y_1 \) may get modified due to the assignment to the dereference to \( p_3 \), while the \( \mu \) annotations indicate that the use of the dereference of \( p_3 \) may use \( x_2 \) and \( y_2 \). The HSSA form creates virtual variables to represent indirect memory references, such that, unique virtual variables are assigned to indirect memory references having similar alias behaviour.

Arrays (and other structures that can be modelled as arrays) vexed SSA-based optimizers till the Array SSA [55], with its ability to capture element-level data flow information of array variables, arrived as a major advance. In the Array SSA form, an initial placement for \( \phi \)-functions follow the following rules (though an optimized semantically alternate placement might also be possible):

- define \( \phi \): A definition of an array element is followed by a \( \phi \)-function to merge the values of the modified element(s) with that of the of the (unmodified) array.
- merge \( \phi \): Inserted at the dominance frontiers, these \( \phi \)-functions serve to merge values flowing along different control-flow paths.

Some optimizations (like an application to parallelization described in [55]) need runtime evaluation of the \( \phi \)-functions. For such applications, the semantics of the \( \phi \)-function is defined using an @ array: for a definition \( X_k[i] \), the corresponding element \( @X_k[i] \) records its last modified time-stamp; at runtime, the \( \phi \)-function “selects” the array element with the most recent time-stamp.

For predicated code, as different predicated definitions of the same variable may reach an use, the traditional SSA form falls short; the \( \psi \) -SSA form [5] simplifies the use of SSA-based optimizations on predicated code. Figure 4.3 shows an example adapted from [5]: In the \( \psi \) -SSA form, the \( \psi \)-functions merges reaching definitions that can reach a (possibly guarded) use. The purpose of \( \psi \)-function can be thought to be one that is similar to our \( \tau \)-function, albeit in a different world, i.e. to “filter” reaching definitions. If each guard is considered opaque (i.e. we only consider a control-flow split at the point, but

\[
\begin{align*}
1 & \quad x_1 := \ldots \\
2 & \quad y_1 := \ldots \\
3 & \quad \text{if } (x_1 = x_2) \text{ then} \\
4 & \quad \quad p_1 := \&x_1 \\
5 & \quad \text{else} \\
6 & \quad \quad p_2 := \&y_1 \\
7 & \quad p_3 := \phi(p_1, p_2) \\
8 & \quad \ldots \\
9 & \quad *p_3 := \ldots \\
10 & \quad x_2 := \chi(x_1) \\
11 & \quad y_2 := \chi(y_1) \\
12 & \quad \ldots \\
13 & \quad \mu(x_2), \mu(y_2) \\
14 & \quad \ldots := *p_3
\end{align*}
\]

Figure 4.2: The \( \chi \) and \( \mu \) functions in the Hashed SSA Form [4].
Figure 4.3: The $\psi$-SSA form [5].

not the predicate/guard that causes the split), the (concrete) definitions reaching the program point 7 are the definitions $x1$, $x3$ and $x5$ (ignoring the PSI assignments). However, examining the guards reveals that statement 5 kills the definition at statement 2 (both are executed under the same guard predicate); also, because the use is executed under the same predicate ($g1$), the definition $x1$ is also unable to reach statement 7. The $\psi$-SSA form models this scenario by introducing a PSI statement at line 6.

Lin et al. [15] propose a speculative SSA form by creating speculative versions of the HSSA operators — speculative update ($\chi$), and speculative use ($\mu$). The speculation flag, either by use of profiling information and/or a set of heuristic rules, is turned on these operators if it is highly likely that the update or reference would be realized at runtime. Lin et al.’s work is orthogonal to our work as we target exposing the hot use-def chains rather than likely alias relations. Both these techniques can be seamlessly combined for a more powerful speculative optimization framework.

The Gated Single Assignment Form (GSA) [56] introduces multiple pseudo-functions to aid symbolic analysis. SSA forms for parallel programs have also been proposed [57, 58, 59].

Most of the above mentioned extensions are orthogonal to the HPSSA form. We may attempt to amalgamate some of the above proposals into the HPSSA form via concrete implementations in the future.

4.3 Profile-Guided Analysis and Optimizations

Ammons and Larus [51] propose an interesting technique towards analyzing and optimizing programs by constructing a hot path graph — the control-flow graph with hot paths duplicated. As the dataflow facts on the isolated hot paths are not polluted by those generated by the rest of the program, flow-analysis on the hot path graph yields better precision on these hot paths. The analysis algorithm utilizes data-flow tracing, one of the two algorithms proposed by Holley and Rosen [60] for solving qualified flow-analysis problems.

Traditionally, superblock scheduling [49] techniques relied on basic-block or edge profiles to identify
the set of frequently-executed program paths; Young et al. [48] used path-profiles to improve superblock
scheduling. Tallam et al. [3] identified the need for longer paths for aggressive optimizations. They
pointed out that trace scheduling of a loop, that is unrolled once, would require profiles capturing longer
paths spanning two loop-iterations. They also mentioned that redundancy may occur in loops, when
the same instruction is executed in different iterations, and across procedure calls, when an instruction
in the called functions deems an instruction in the caller redundant. Motivated by the above issues,
they proposed an algorithm for estimating the frequency of paths that spans across loop and procedure
boundaries.

Speculative and Probabilistic Dataflow Analysis techniques to infer highly-likely facts have also been
proposed to enable speculative optimizations. Ramalingam [61] used edge-profiles to infer the probability
with which a fact holds true for the class of finite bi-distributive subset problems. Mehrota and Solz
[62] improved on Ramalingam’s framework by taking execution history into account.

Pointer analysis aims to infer the possible locations a pointer variable might point to; however, if
certain paths in the program are more likely to be traversed than others, certain points-to facts are more
likely than others. Probabilistic pointer analyses [63, 64, 65] assign probabilities with which a points-to
relation might hold at a program point.

Partial Redundancy Elimination (PRE) [6] is recognized as one of the important optimizations for
a compiler. Capable of subsuming global common subexpression elimination and loop-invariant code
motion, PRE aims to eliminate partially redundant computation — computation that is redundant
along some, but not all paths. Figure 4.4(a) illustrates a possible scenario: the computation in the node
d is partially redundant as it is redundant along path acd but not along bce; Figure 4.4(b) shows how
the partially redundant computation can be avoided by hoisting the computation to node b. Figure
4.5(a) illustrates a case when conventional PRE is not possible as the path bce does not contain the
computation; however, if the frequency of the path acd is much higher than that of bce, it might still
be profitable to hoist the computation of the expression to node b (Figure 4.5(b)). Note, however, that
it introduces a new computation in the path bce: in case the computation in question can raise an
exception, this approach would require hardware support for suppressing/delaying any such exception
raised. PRE has been a target of multiple speculative algorithms: Gupta et al. [7] presented a path
profile based speculative PRE that uses cost-benefit analysis to judge the profitability of speculative
code-motion; Xue and Cai [8] propose an optimal algorithm for edge-profile directed speculative PRE.
Horspool et al. [9] sacrifice optimality for efficiency on the way to a fast speculative PRE algorithm —
an algorithm that can be particularly useful in JIT compilers. Gupta et al. have also demonstrated the
applicability of a cost-based analysis for eliminating Partial Dead Code [50].

The HPSSA form provides a common ground for writing efficient speculative optimizations on a
sparse program representation; it scores over flow-based speculative optimizations due to the exact
(a) The computation in node $d$ is partially redundant.

(b) The partial redundancy is eliminated by "hoisting" the computation of $u + v$ to node $b$.

Figure 4.4: Partial Redundancy Elimination [6].

(a) The computation of $u + v$ is redundant along the path $acd$. However, PRE is not applicable as the computation does not appear along the path $bce$.

(b) If the path $acd$ is executed more often than $bde$, it might still be beneficial to eliminate the redundancy along $acd$ by hoisting the computation to node $b$, even though it introduces an additional (unnecessary) computation along the path $bce$.

Figure 4.5: Speculative Partial Redundancy Elimination [7, 8, 9].

reason that the SSA-based algorithms score over the flow-based safe optimizations. We are interested in designing efficient sparse algorithms based on the HPSSA for some of the above mentioned speculative analyses and optimizations in the future.
Chapter 5

Conclusions

In this thesis, we present novel tools aiming to improve speculative analyses and optimizations: a multi-iteration control-flow path profiling algorithm, and an SSA form to guide path profile directed speculative optimizations.

The k-Iteration path profiling algorithm can endow the design of speculative optimizations directed towards a program’s frequently occurring loop-iteration patterns. In Chapter 2, we provide an instance where k-iteration profiling discovers an optimization opportunity that would have been unsighted by acyclic path profiles. We are interested in understanding the run-time characteristics of program loops and designing relevant loop-directed speculative optimizations in the future.

Program profiling has discovered applications beyond optimization. The modern software engineering arena has witnessed the employment of path-profiling in bug isolation [53] and measuring coverage of test-suites [66]. We envisage that k-Iteration profiles, being more informative than acyclic path profiles, could advance the effectiveness of these applications as well. We intend to study the sensitivity of these applications to the k-iteration profiles.

The Hot Path SSA (HPSSA) form fills a profound void for an SSA-like form in the domain of path-profile guided speculative optimizations. We demonstrate, using Wegman and Zadeck’s Sparse Conditional Constant Propagation, as to how effective sparse algorithms for speculative optimizations can be designed on the HPSSA form by modest tailoring of existing SSA-based algorithms. Now, SSA-based analysis and optimization algorithms that evolved over the last couple of decades, can utilize the HPSSA form to “cross-over” to the speculative kingdom. Speculative avatars of Partial Redundancy Elimination and the Global Value Numbering algorithm on the HPSSA form are our primary targets at this point of time.

The rich k-Iteration profiles can enhance the potency of the HPSSA form as well, allowing the HPSSA-based optimizations to exploit more speculation opportunities. We illustrate a small example to that effect: Figure 5.1(a) presents a control-flow graph that contains a loop-carried dependency: the
(a) A program with loop-carried dependency with its hot acyclic paths shown.

(b) HPSSA on acyclic path profile.

(c) HPSSA on two-iteration path profile.

Figure 5.1: The Hot Path SSA form on multi-iteration profiles.

definition of variable y in basic-block 7 feeds from the value assigned to x in the previous iteration of the loop. Consider a hot trace:

\[
\cdots - \{2 - 5 - 6 - 8 - 9\} \cdots
\]

which can also be represented by the following sequence of acyclic paths:
\[
\cdots - (p_3 - p_2 - p_1)^{50} \cdots
\]

Figure 5.1(b) shows the HPSSA form for the program that is built using the acyclic path profile corresponding to the above execution trace. As the acyclic path profile identifies the acyclic paths \( p_1 \), \( p_2 \) and \( p_3 \) as hot, the \( \tau \)-function in basic-block 8 would ensure the SCC algorithm that the definition of \( x \) can be assigned a speculative value 52\(^{\circ}\). However, with both the paths \( p_2 \) and \( p_3 \) graded hot, HPSSA construction algorithm withdraws from inserting a \( \tau \)-function in basic-block 7 — a prudent decision considering that only the acyclic path profile is available, and hence hot definitions reaching this point can either be from \( x_2 \) or \( x_3 \) (assigned in the previous iteration).

Figure 5.1(c) presents the HPSSA form built using the \textit{two-iteration} path profile on the same trace. The two-iteration path profile identifies the two-iteration paths \( p_3 p_2 \), \( p_2 p_1 \) and \( p_1 p_3 \) as hot. Now, armed with a better representation of the program execution, the HPSSA construction algorithm can plant a \( \tau \)-function in basic-block 7, which fortifies that the definition of \( y \) is \textit{mostly} assigned the constant 19 (SSCC would detect it as the speculative constant 19\(^{\circ}\)). This was possible as the richer two-iteration profile exposes the fact that \( p_2 p_1 \) is hot and, also that \( p_3 p_1 \) is \textit{not} hot — a dilemma that the acyclic path profile could not resolve.

To conclude, we believe that the opportunities offered by the tools proposed in this thesis are enormous. We envisage a long road ahead, beckoning exciting applications of these tools at a better understanding and exploitation of an executing program’s runtime demeanor.
Bibliography


