The Hot Path SSA Form: Extending the Static Single Assignment Form for Speculative Optimizations

Subhajit Roy and Y. N. Srikant

Computer Science and Automation Department,
Indian Institute of Science.
{subhajit, srikant}@csa.iisc.ernet.in

Abstract. The Static Single Assignment (SSA) form has been an eminent contribution towards analyzing programs for compiler optimizations. It has been affable to the design of simpler algorithms for existing optimizations, and has facilitated the development of new ones. However, speculative optimizations — optimizations targeted towards speeding-up the "common cases" of a program — have not been fortunate enough to savor an SSA-like intermediate form. We extend the SSA form for speculative analyses and optimizations by allowing only *hot reaching definitions* — definitions along frequent acyclic paths in the program profile — to reach its respective uses; we call this representation the *Hot Path SSA form*. We propose an algorithm for constructing such a form, and demonstrate its effectiveness by designing the analysis phase of a novel optimization — Speculative Sparse Conditional Constant Propagation: an almost obvious extension of Wegman and Zadeck's Sparse Conditional Constant Propagation algorithm. Our experiments on some SPEC2000 programs proves the potency of such an optimization.

1 Introduction

Program analyses and optimizations have benefited immensely from the SSA form as an intermediate representation. An extremely simple idea — allow only a single definition of a variable to reach the statements using it — prunes out false dependencies, and factors long use-def chains into a web of short, simple ones. A multitude of optimizations were either made possible, or were heavily empowered by the SSA form — sparse conditional constant propagation, global value numbering, and strength reduction to name a few.

However, speculative optimizations — optimizations biased towards frequently executed paths — have not been fortunate enough to enjoy an SSA-like intermediate representation. These optimizations have recently attracted a lot of attention, and are now recognised as a major vehicle towards improving program performance.

Modern compilation systems, acknowledging the importance of such unconventional optimizations, have started providing support for speculative analysis and transformation. However, in most of the intermediate representations,

the profiling information is not integrated into the static program representation. This makes implementing speculative optimizations cumbersome, having to handle too many data-structures. Additionally, the absence of an SSA-like sparse representation has hindered the development of efficient algorithms for speculative optimizations.

We propose to extend the power of the SSA form to 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 paper:

- We propose a novel program representation — the Hot Path SSA (HPSSA) form — that allows a use to witness only the "more-likely" reaching definitions (section 4);
- We present an algorithm for constructing the HPSSA form (section 5);
- We demonstrate the potency of the HPSSA form by designing the analysis phase of a novel speculative optimization — Speculative Sparse Conditional Constant Propagation (SSCP) — 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 SCP algorithm [21], SSCP exhibits the possibilities of developing new speculative optimizations using the HPSSA form by tailoring of existing SSA-based traditional optimizations (section 6).

2 Background

2.1 The Static Single Assignment Form

A program is said to be in Static Single Assignment (SSA) form if each use of a variable has *exactly* one reaching definition. A special operator, the ϕ-function, *merges* multiple definitions from different paths into a single definition, forcing any subsequent *use* to see exactly one definition.

Figure 1 shows the SSA form of a program. Notice how the definitions of 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-def structure of a program in SSA form is extremely simple — allowing the design of cleaner and faster algorithms.

2.2 Acyclic Path Profiling

Ball and Larus [2] proposed an efficient algorithm for profiling acyclic paths — paths that terminate either at loop-backedges or at procedure exits. Essentially, an acyclic path profiler "chops-off" paths at a backedge, erasing the sequence in which the acyclic paths in the loop were actually executed. The Ball-Larus algorithm is widely used for speculative analyses and optimizations. We use acyclic path profiles to expose frequent use-def chains in the HPSSA form.

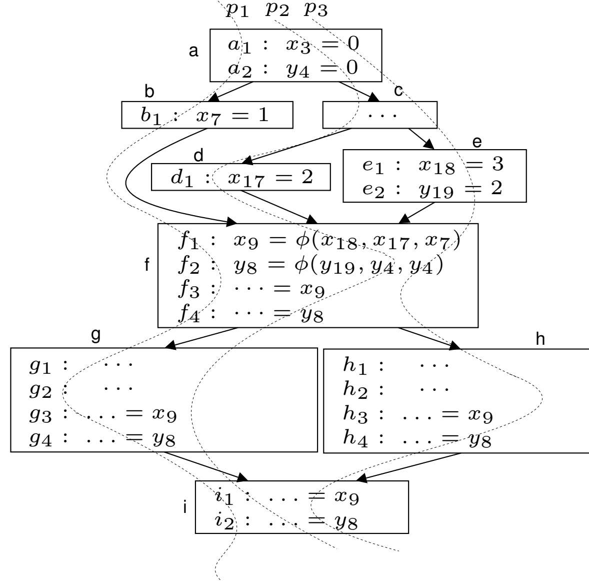


Fig. 1. A program in the SSA form. (***Hot acyclic paths***: p_1:`abfgi`; p_2:`acdfgi`; p_3:`acefhi`)

2.3 A peek at the Hot Path SSA Form

In this paper, we propose to tie the run-time behaviour of a program — as indicated by the frequently executed acyclic paths — directly to its static program representation, thus providing a convenient data-structure for the speculative optimizers. In the proposed representation, which we call the Hot Path SSA (HPSSA) form, an additional construct — the τ-function — is introduced to capture information relevant for speculative analyses and optimizations. The τ-functions act as "filters", separating the more-likely use-def chains from the lesser-likely ones. The first argument of the τ-function is the traditional meet-over-all-paths reaching definition; the rest of the arguments are the "hot" reaching definitions: definitions that are more-likely to reach the respective program point.

Figure 2 shows the HPSSA form of the program in Figure 1. Consider the basic-block g: the τ-function at g_1 indicates that x_9 is the "safe" meet-of-all-paths reaching definition, though the definitions of x_7 and x_{17} are more likely to reach this program point (via the ϕ-statement at f_1). Similarly, for g_4, h_3 and h_4, the hot reaching definitions are from definitions of y_4, x_{18} and x_{19} respectively — all of which are definitions to constants. Hence, the HPSSA form exposes 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 — enabling a speculative optimizer to *speculatively* "predict" the value of these variables.

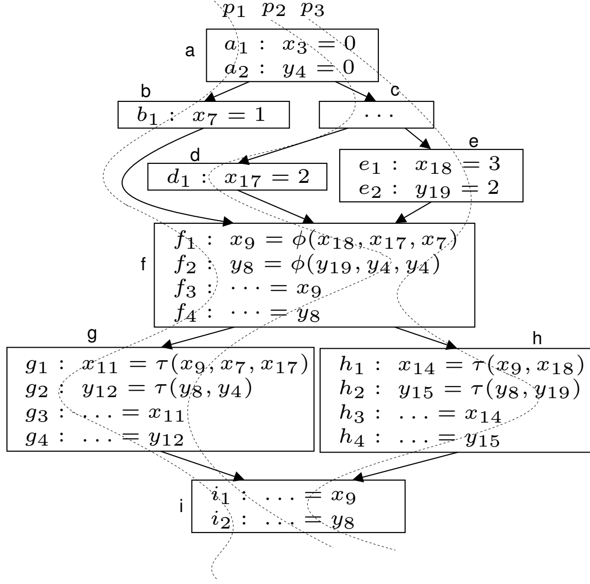


Fig. 2. The program in Figure 1 translated to the Hot Path SSA (HPSSA) form (***Hot paths***: p_1:`abfgi`; p_2:`acdfgi`; p_3:`acefhi`).

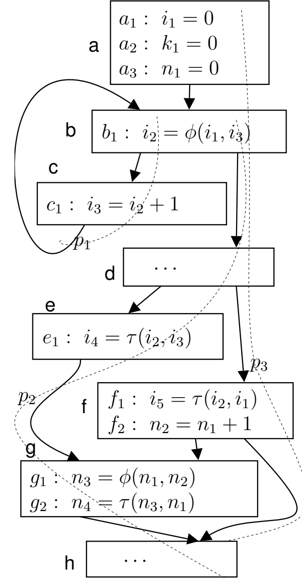


Fig. 3. HPSSA form for a program with loops (***Hot paths***: p_1:`bc`; p_2:`bdegh`; p_3:`abdfh`).

Though the HPSSA form uses acyclic path profiles, it is still adept at propagating hot reaching definitions across loop-boundaries. Figure 3 shows the HPSSA form of a program with a loop. Notice how the variable i_3 becomes the hot reaching definition at the basic block e, even though i_3 reaches the node e along a path that contains a backedge (as `c-b` is a backedge, `c-b-d-e` is not a segment of any acyclic path).

In this paper, we only assume reducible flow-graphs; we also assume the existence of a loop-preheader node (leading to the loop-header) for each loop in the program.

3 Thermal Properties of a Program

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

3.1 Thermal States of Program Entities

Definition 1. *Hot/Cold Paths: A program path $p : n_1 \rightsquigarrow n_2$ 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 left 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 [17, 20]. 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 2. *Temperature (θ) of a node (edge) is defined as:*

- *hot: if the node (edge) is present on a hot path;*
- *cold: if the node (edge) is not present on any hot path.*

A backedge b in a flow-graph is marked hot if, either of the dummy edges, δ_{start} to a loop-header h or δ_{end} from a loop-tail t, is hot[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 1, all the nodes and edges are hot; the path $c \rightarrow d \rightarrow f \rightarrow g$ is hot (through the path p_2) while the path $e \rightarrow f \rightarrow g$ is cold.

Definition 3. *Hot/Cold Reaching Definitions and Definition Chains*

A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 <u>hot</u> if there exists a hot path from n_1 to n_2, and δ is not killed along that path. A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 <u>cold</u> if there does not exist a hot path from n_1 to n_2, and δ is not killed at least along one cold path from n_1 to n_2.

Consider Figure 1: treating a ϕ-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 it via 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 via the paths p_1 and p_2). In the SSA form, the ϕ-functions can be seen as creating a *definition chain*, that is broken only by a non-ϕ definition: $x_7 \rightarrow x_9$ and $x_{17} \rightarrow x_9$ are the *hot reaching definition chains* at g_3 , while $x_{18} \rightarrow x_9$ is a *cold reaching definition chain.* In the HPSSA form, the τ-functions "kill" the cold definition chains: for example, in Figure 2, $x_{18} \rightarrow x_9$ no longer reaches g_3 as it is killed by g_1.

---

[1] The Ball-Larus profiler converts a flow-graph with cycles into a directed acyclic graph (DAG) by adding dummy edges, $\delta_{start}/\delta_{end}$, to and from the backedge source/target (respectively) for each loop in the program [2].

3.2 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, 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 are said to form the *BuddySet* at a node. Consider Figure 1 with the following set of hot paths:

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.

$BuddySet_a(f) = \{\{p_1, p_5\}, \{p_2\}, \{p_3, p_4\}\}$; i.e. p_1 and p_5 are buddies, so are p_3 and p_4, while p_2 has no buddy at f.

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.
- $Paths_s(u \to v)$: The set of all profiled "hot" acyclic paths reaching the node u that initiate from the incubation node s and progress along the edge $u \to v$ from u; without the subscript s, it denotes paths from all incubation nodes that progress along $u \to v$.
- $S(u)$: 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 α.

4 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 τ-function — to "filter" definitions along cold paths, thus allowing only *hot* definitions to propagate further. The form of a τ-statement is shown below:

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

The τ-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 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 τ-function can be seen as a conditional ϕ-function:

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

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 τ-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 τ-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 "killed" by τ-functions on their way to the cold node.

Each speculative argument x_i in a τ-function is mapped to the set of hot profile paths along which the definition corresponding to x_i is reached. In Figure 2, for the variable x in g_1, the τ-function allocates the parameter x_7 corresponding to the path p_1, and the parameter x_{17} for the path p_2. However, for the variable y 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 honours the constraint imposed by the SSA form: each use is reachable by a single definition — encouraging the development of speculative extensions of existing SSA-based algorithms on the HPSSA form.

Exiting the HPSSA form

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

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

This puts the program in the SSA form; one can then use a standard out-of-SSA algorithm to exit the SSA form.

5 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:

- Insert ϕ-statements: The classic algorithm for construction of the minimal SSA form [8] places ϕ-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 iff u does not dominate v while a predecessor of v is dominated by u.

- Insert τ-statements: For each variable x, we identify program points that necessitate a τ-function, and, at all such points, insert a definition of the form $x = \tau(x)$ (discussed in detail in section 5.1).
- Variable renaming: The definitive variable renaming algorithm [8] uses a variable 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 τ-functions to the variable version corresponding to the meet-over-all-paths "safe" reaching definition.
- Allocation of the τ-function arguments: Finally, we allocate the speculative arguments to the τ-functions in correspondence to the hot reaching definition chains (discussed in detail in section 5.2).

Note that after step 3, the program is in SSA form, and 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.

5.1 Thermal Frontiers: Placing τ-functions

We call definitions due to ϕ and τ- functions as *pseudo* definitions, differentiating them from other *concrete* definitions; the corresponding statements are called pseudo/concrete statements. We define the set of *visible* definitions in the basic-block u as the last definition of 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 ϕ- or a τ-function.

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

The ϕ-functions act as *definition mergers* — "merging" multiple definitions into a single one. Comparably, the τ-functions act as *definition filters* — separating hot definitions from cold ones, which were merged by previously occurring ϕ-functions. *Hence, a node n will need a τ-function for a variable v if, and only if, both a hot and a cold reaching definition for the variable v arrive at n.*

The minimal SSA construction algorithm uses an exquisite structure — the Dominance Frontier — to insert the ϕ-statements. To build the HPSSA form, we identified a similar structure to place the τ-statements: the Thermal Frontier.

Definition 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, ($v \in TF(u, d)$), iff the node v is also exposed to a reaching definition d', defined at a node w (w not dominated by u), such that $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$. Also, v must be the first node in the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.*

Stated informally, a node v is in the thermal frontier of a hot/cold reaching definition d (defined at u), if v is also reachable by a different cold/hot (respectively) definition d' (defined at w), while being the first node along $u \rightsquigarrow v$ and $w \rightsquigarrow v$ to satisfy the conditions.

Unlike Dominance Frontiers, Thermal Frontiers need not be join nodes. For example, in Figure 2, node $g \in TF(b, x_7)$ as x_7 is a hot reaching definition (along p_1) and g is also reachable by the cold reaching definition x_{18}.

It is apparent that τ-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.[8].

Definition 5. *Let $\gamma_x(u)$ return the visible definition of the variable x in the basic-block u; then, for a set of nodes κ, the Iterated Thermal Frontier (ITF) is the limit of the increasing sequence of sets of basic-blocks:*
$TF^x(\kappa) = \bigcup_{u\in\kappa} TF(u, \gamma_x(u))$
$TF^x_1 = TF^x(\kappa)$
$TF^x_{i+1} = TF^x(\kappa \cup TF^x_i)$
$ITF^x = TF^x_\infty$, *where* TF^x_∞ refers to the fixpoint, i.e. when $TF^x_i = TF^x_{i+1}$

However, as the ϕ-statements are inserted by a prior phase, placing the τ-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 τ-functions, if a τ-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 ϕ-statement (as no node in the loop-body can dominate h). Hence, if the CFG nodes are processed in the topologial order, insertion of τ-functions at the required nodes due to the definition of the variable x at h would have already happened.

Theorem 1. For a set of visible definitions of a variable x at a set of nodes κ, τ-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 1. *A node $n \in TF(u, d^x)$ for a definition d^x (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 τ-function is unnecessary as n can then be reachable by only hot or only cold definitions of x. If condition II fails, a τ-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 $v \notin TF(u, d^x)$ if the same definition d^x 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 τ-function templates at all points identified by the lemma, leaving the task of weeding out unnecessary τ-statements to the τ-argument allocation phase (section 5.2). In the rest of the discussion, we denote the set of Thermal Frontiers computed according to Lemma 1 as $TF(u, d)$, and denote the ideal set of Thermal Frontiers (as defined in Definition 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 1; we then identify a scheme for satisfying the second condition.

Caloric Connector

Definition 6. *Caloric Connector (CC): A node $n_{cc} \in CC$ if, for distinct nodes n and n' ($n \neq n'$), there exist paths $n \leadsto n_{cc}$, $n' \leadsto n_{cc}$ such that $\theta(n \leadsto n_{cc}) \neq \theta(n' \leadsto n_{cc})$, and for all nodes $n'' \in (N(n \leadsto n_{cc}) \cap N(n' \leadsto n_{cc})) - \{n_{cc}\}$, $n'' \notin CC$.*

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 $n \leadsto n_{cc}$ and $n' \leadsto n_{cc}$ satisfying these properties.

Consider Figure 1: 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.

Lemma 2. *A hot acyclic path $t \leadsto u$ extended by a forward edge $u \rightarrow v$ forms a cold path $t \leadsto u \rightarrow v$ if, for some incubation node s, there exists a set of buddy paths $B \in BuddySet_s(u)$ among the paths at u, such that none of the buddies $\sigma \in B$ traverse the edge $u \rightarrow v$.*

Lemma 3. *If an acyclic path $t \leadsto u \rightarrow v$ is cold, then, either*

- *$t \leadsto u$ is cold, or*
- *$s \leadsto t \leadsto u$ is hot, and $\exists B \in BuddySet_s(u)$, such that none of the buddies $\sigma \in B$ traverses $u \rightarrow v$ (where s is the incubation node for $s \leadsto t \leadsto u$).*

The intuition for the above lemmas is as follows: Each set of buddies at u, $B_i \in BuddySet_s(u)$, correspond to a unique sequence of edges $(s \leadsto u)_i$ from s to u, distinct from that of any other buddy set $B_j \in BuddySet_s(u)$, $B_i \neq B_j$. If no hot path $p \in B_i$ selects the edge $u \rightarrow v$, that particular sequence of edges $(s \leadsto u)_i \rightarrow v$ is surely missing among the hot paths reaching v. This implies that the path $(s \leadsto u)_i \rightarrow v$ is cold. We omit the formal proofs for want of space.

Algorithm 1 Computing the set of Caloric Connectors

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

1. Initialize $hasAColdPath$ and $hasAHotPath$ to $false$.
2. For all edges $e : u \rightarrow v$,
   - if $\theta_c(u \rightarrow v)$, set $hasAColdPath = true$;
   - if $\theta_h(u \rightarrow v)$,
     (a) Set $hasAHotPath = true$;
     (b) If e is not a backedge, and if, $\exists B \in BuddySets_s(u)$ (for some incubation node s) such that B does not intersect $Paths(u \rightarrow v)$, set $hasAColdPath = true$.
3. If both $hasAColdPath$ and $hasAHotPath$ are $true$, add v to the set of Caloric Connectors.

The algorithm for computing the set of Caloric Connectors (Algorithm 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 in accordance to Lemma 2 and 3, and sets the flags $hasHotPath$ and $hasColdPath$ accordingly. A node v is marked as a Caloric Connector if it has both a hot and a cold path reaching it.

Computing Thermal Frontiers For a concrete definition d and a basic-block $v \in TF(u, d)$, the second condition of Lemma 1 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).

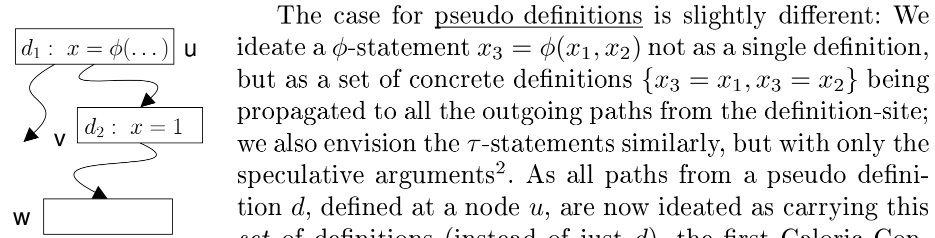


Fig. 4. Violation of condition II of Lemma 1.

The case for pseudo definitions is slightly different: We ideate a ϕ-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 all the outgoing paths from the definition-site; we also envision the τ-statements similarly, but with only the speculative arguments[2]. As all paths from a pseudo definition d, defined at a node u, are now ideated 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 1 — provided the pseudo-definition d actually reaches n_{cc}. Figure 4 illustrates this 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 1.

Algorithm 2 outlines our algorithm for inserting τ-nodes.

[2] In the HPSSA construction algorithm, the hot definitions are "percolated" through the ϕ and τ statements as the percolated definitions may appear as arguments to future τ-statements.

Algorithm 2 Inserting τ-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: if the pseudo definition d is a reaching definition at v (d is not killed by concrete definitions along some path to v), add the set of the Closest Caloric Connectors for v to $TF(v, d)$;
   - if d is a concrete definition: $TF(v, d) = DF(v) \cap CC$.
2. For all $u \in TF(v, d)$, for all visible definitions "$d: \ x = \ldots$" in the basic-block v: if u does not already have a τ-function for x, insert a τ-statement: $x = \tau(x)$ just after all ϕ-statements (if any) at u, before any concrete statement.

5.2 Allocating τ-function arguments

Before delving into the details of the algorithm, we take a slight digression into a deeper understanding of the ϕ and τ statements. 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 would 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 2, we 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.

The algorithm, in essence, computes the path-sensitive *active* reaching definitions at each node u containing a τ-function. The hot reaching definitions (variable versions) stand as arguments in the τ-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 ξ_i can be used as a parameter for a τ-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., it does reach u along a 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*[3] in x_i may be reaching u: In Figure 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 τ-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 ξ if, and only if, *all the concrete reaching definitions that it merges reaches* u along ξ. This condition can be ensured by checking if all

[3] A definition for x_i is *contained* in a ϕ-definition if the ϕ-function argument-list either includes x_i, or includes a variable-version x_j such that x_i is *contained* in the definition for x_j; for the τ-functions, we only consider the speculative argument-list.

the *contained* concrete definitions for x_i are available as active definitions at u for the set of paths ξ.

Allowing definitions corresponding to pseudo-definitions in the τ-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 abandon any further discussion on the same: in the following discussion, 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). As pseudo-definition "labels" to a set of merged definitions can no longer appear in the τ-function argument lists, the implication of ignoring the pseudo definitions is a larger argument list for the τ-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 [8] — using a variable 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 stack maintains the set of active reaching definitions (x_i), along with the set of hot paths (ξ_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 $DefPaths$ be the set of P. A *frame* in the variable stack is a map $[DefPaths \mapsto Version]$, where $Version$ 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 $\xi_i \in DefPaths$. A variable stack $VarStack_x$ is a stack of frames for the base variable x.

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

A $Frame$ in $VarStack$ supports the following operations: `get(`ξ`:DefPaths)` returns the version associated with ξ in the map; `accumulate(`ξ_i`:DefPaths,` x_i`:Version)` accumulates definitions: if a pair $[\xi_j, x_i] \in Frame$, 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.

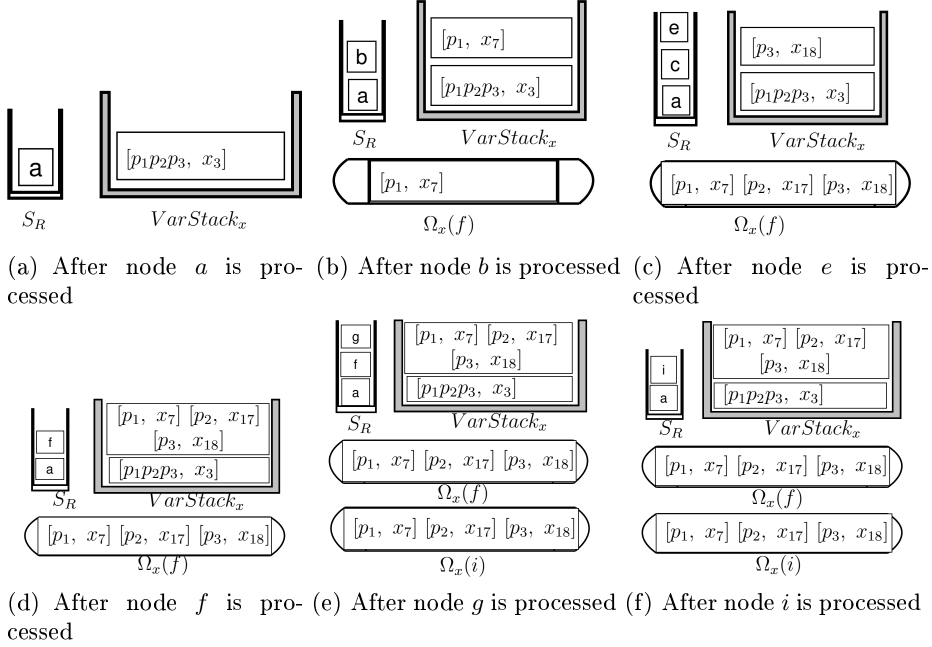
The top of the variable stack contains the set of active definitions — definitions that can be used to allocate arguments to the τ-functions in the current basic-block. The algorithm traverses the control-flow graph recursively in a depth-first order over the dominator tree (as does the variable renaming phase for SSA construction); the set of dominatees[5] are traversed in the topological order of the nodes in the control-flow graph: the order is important to ensure that

[4] The updates to ξ_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 dominatees of n.

when a basic-block is processed, the definitions from all its incoming paths reach it. The active definitions are propagated via $VarStack$ 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 `accumulate(`ξ_i`:DefPaths, `x_i`:Version)` for a frame) in a *Definition Accumulator* $\Omega_x(u)$ from its predecessors in the CFG — it is loaded up on $VarStack$ when the node u is processed.

The τ-allocation algorithm is sketched in Algorithm 3. Let us describe the algorithm via an example (Figure 5) for the flow-graph in Figure 2:



(a) After node a is processed (b) After node b is processed (c) After node e is processed

(d) After node f is processed (e) After node g is processed (f) After node i is processed

Fig. 5. Steps in the execution of the τ-argument allocation algorithm.

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 (a)), and then recurses on the children of a in the dominator tree, 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 (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 stack and the recursion stack (S_R) now again resemble that in Figure (a).

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

Algorithm 3 A sketch of the τ-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 which do not have a ϕ-definition appearing in the basic-block u, push a frame $\langle \xi_i, x_i \rangle$, where ξ_i 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 ϕ-statement: if u is a loop-header and the dummy profile edge $t \to \delta_{end}$ is hot (where δ_{end} denotes the dummy-end node for a Ball-Larus profiler, and t is the corresponding loop-tail), accumulate $\langle \xi_i, x_i \rangle$ at the topmost frame of $VarStack_x$, where
      i. ξ_i is the set of all paths that incubate from u, and
      ii. x_i is the ϕ-statement argument corresponding to the backedge $t \to u$.
   (b) If stm is a τ-statement:
      i. Create a set C of candidate definitions from the definitions in $VarStack.top()$ for each incubation node s: add $\langle \xi_i, x_i \rangle$ to C iff $(Paths_s(u) \cap \xi_i) \neq \emptyset$;
      ii. If there exists at least one $x_i \in C$ such that its variable-version differs from the safe argument x_0, add arguments to the τ-function for each x_i, mapping the respective variable position to ξ_i; otherwise, replace the τ-function with a simple copy statement: $x_{out} = x_0$.
   (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 Ω of the (forward) successors (if successor is a join node): for each forward (ignore backedges) successor edge $u \to v$, if v is a join node, for each $\langle \xi_i, x_i \rangle \in VarStack.top()$ such that $(\xi_i \cap Paths(u \to v)) \neq \emptyset$, accumulate $\langle \xi_i \cap Paths(u \to v), x_i \rangle$ in Ω_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.

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

The node g is processed next: on encountering the τ-definition for x_{11}, the algorithm (Step 3(b)) attempts to allocate arguments for the same: Examining the active definitions (top of the variable 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 are from the possible set of definitions to be added as arguments to the τ-function.

As the variable versions in the set C differ from that of the safe argument, we allocate arguments to the τ-function from C.

$$x_{11} = \tau(x_9,\ \ x_7\,\langle p_1\rangle\,,\ \ x_{17}\,\langle p_2\rangle)$$

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

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

Now consider the control-flow graph with loops (Figure 3): Let us illustrate as to how the the hot reaching definition of i_3 in the block c is identified as a hot reaching definition at the τ-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 $\langle p_1p_2, i_3\rangle$ is added to the top of the variable stack (being the parameter to the ϕ-function corresponding to the backedge) by Step 3(a). When the algorithm recurses on the children of d in the dominator tree, the variable stack carries the definition to the basic-block e where it is recognised as an argument for the τ-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.

6 Speculative Sparse Conditional Constant Propagation

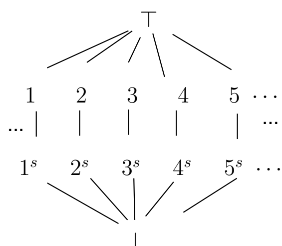


Fig. 6. The SSCP Lattice (the constants superscripted by 's' are the speculative constants).

```
i0=0;
while(...) {
    i1=φ(i0, i3);
    i2=τ(i1, i3);
    i3=i2 + 1;
}
```

Fig. 7. A case that requires meet with its old value in the τ-statement transfer function for SSCP.

We have implemented the analysis phase of a novel optimization — the Speculative Sparse Conditional Constant Propagation (SSCP) on the HPSSA form. This optimization expands the scope of the SCP [21] 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 — through this novel analysis, 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 SSCP algorithm operates on a four level lattice (Figure 6 shows the SSCP 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 ϕ-function) hold as in SCP, 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 τ-functions is defined as follows (where $\sqcap$ is the meet operator): If the meet of all the arguments does not produce $\perp$ (not-constant), the transfer function resembles the transfer function for the ϕ-functions. Even if the meet of all the arguments turns out to be $\perp$, there might still be the chance of the expression being identified as a speculative constant: let $\beta = x_1 \sqcap x_2 \ldots \sqcap x_n$. The transfer function attempts to return β, if β is $\perp$, $\top$ or a speculative constant; if β is a "safe" constant, β moves in the lattice to $(\beta)^s$, the corresponding speculative constant. Formally, the transfer function for $\tau(x_0, x1, \ldots, x_n)$ is given by the following (where each expression refers its abstract value in the lattice, and $\beta = x_1 \sqcap x_2 \ldots \sqcap x_n$):

$$\tau(x_0, x1, \ldots, x_n) \sqcap \begin{cases} x_0 \sqcap \beta & \text{if } x_0 \sqcap \beta \neq \perp \\ \beta & \text{if } x_0 \sqcap \beta = \perp \text{ and } \beta \text{ is not a safe constant} \\ (\beta)^s & \text{otherwise} \end{cases}$$

The meet with the current value of the τ-function is added to ensure termination by ensuring monotonicity; otherwise, code fragments resembling that in Figure 7 will never reach a fixpoint due to i_3 increasing its value in the speculative domain, and the τ-function feeding the same value back to it. We omit detailed discussions on this analysis for want of space.

7 Implementation and Experiments

We implemented our HPSSA construction algorithm, as well as the analysis phase of the SSCP algorithm on the Scale compiler [18]; we were also aided by the CIL [7] tool. We only cast scaler variables whose address has not been taken in the HPSSA form; τ-functions are not introduced for the remaining variables. The SSCP 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 SPEC2000 benchmark suite. We used a naive hot path selection criteria: all the acyclic paths executed on the *train* input set was considered "hot" for building the HPSSA form. Table 1 exhibits our findings for programs run on the *ref* input set. The programs were run with the default parameters, i.e., no parameters were set on

Table 1. Speculative Constants discovered by the SSCP algorithm. ('~' indicates *almost*; grp, prg, & src refer to inputs graphic, program & source respectively).

Program	Inpt	Variable Uses		Expression Uses		Total		
		Uses	**HitRt**	**Uses**	**HitRt**	**Hits**	**Misses**	**HitRt**
181.mcf	-	33110	100.00	49665	100.00	82775	0	100.00
175.vpr	-	6938074	100.00	8110837	100.00	15048911	0	100.00
164.gzip	grp	26592	100.00	5	100.00	26597	0	100.00
	prg	17412	100.00	5	100.00	17417	0	100.00
	src	4721	99.98	5	100.00	4725	1	99.98
197.parser	-	165970964	~100.00	340	97.94	165970861	443	~100.00
256.bzip2	grp	132106650	~100.00	938	76.97	132107372	216	~100.00
	prg	100819492	~100.00	6576416	15.67	101849942	5545966	94.84
	src	108134316	~100.00	5256006	17.94	109077366	4312956	96.20

the command line, either for training, or for the actual run (on the *ref* set). We collected statistics for *dynamic uses* (use of a variable/expression during the actual run) for variables (*Variable Uses*), and for sub-expressions that could be constant-folded speculatively (*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 same as that for the original SCP algorithm. We also indicate the *Hit Rate* (*HitRt*): the percentage of uses where the use of variable/expression actually agrees with the "predicted" speculative constant value.

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.

8 Related Work

Multitude of interesting extensions and modifications have been proposed on the SSA form. The Hashed SSA (HSSA) form [6] extends the traditional SSA form to accommodate pointer variables by introducing an explicit may modify operator (χ) and may reference operator (μ). The Array SSA [13] form captures element-level data flow information of array variables. The ψ-SSA form [19] simplifies the use of SSA-based optimizations on predicated code. Though we have not addressed aliasing and arrays in this paper, it does not seem difficult to address these issues in the HPSSA form; we may investigate such extensions via concrete implementations in the future.

Lin et al. [14] proposed a speculative SSA form by extending speculative versions of the HSSA operators — speculative update (χ_s) and speculative use (μ_s). The speculative 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 an update

or reference will be substantiated 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.

Towards path-sensitive program optimizations, Ammons and Larus [1] proposed performing flow-analysis on a *hot path graph* that isolates the frequent paths. Das et al. [10] proposed a polynomial-time path-sensitive algorithm for verifying a given temporal safety property, and proved it effective by verifying the file I/O behaviour of a version of the GNU C Compiler.

Researchers have also been interested in inferring *likely* data-flow facts, computed over control-flow profiles. Ramalingam [16] used edge-profiles to infer the probability with which a fact holds true for the class of finite bi-distributive subset problems. Probabilistic pointer analyses [4, 9] assign probabilities with which a points-to relation might hold at a program point. Contributions to speculative partial redundancy elimination have been made by [15, 22]. Path profile based speculative PRE and PDE have been proposed by [11, 12]. Most of these techniques use edge and node profiles which are much weaker than path-profiles used by HPSSA. Also, the HPSSA form provides a common ground for writing efficient optimizations on a sparse program representation; it scores over flow-based speculative optimizations due to the exact reason that the SSA-based algorithms score over the flow-based safe optimizations.

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 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 [3] and Partial Redundancy Elimination [5] being our foremost targets. We are also interested in extending the HPSSA form for richer profiles like the k-iteration [17] profiles.

Acknowledgements. Subhajit Roy was supported by Doctoral Fellowship from Philips Research, India.

References

1. Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. *SIGPLAN Not.*, 39(4):568–582, 2004.
2. Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture (MICRO)*, pages 46–57, 1996.
3. Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value Numbering. *Software: Practice and Experience*, 1997.

4. Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Interprocedural Probabilistic Pointer Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 15(10):893–907, 2004.
5. Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Programming Language Design and Implementation (PLDI)*, pages 273–286, 1997.
6. Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *International Conference on Compiler Construction (CC)*, pages 253–267, 1996.
7. CIL - Infrastructure for C Program Analysis and Transformation. `http://hal.cs.berkeley.edu/cil/`.
8. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
9. Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. *SIGARCH Comput. Archit. News*, 34(5):416–425, 2006.
10. Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.
11. Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Dead Code Elimination Using Predication. In *Parallel Architectures and Compilation Techniques (PACT)*, page 102, 1997.
12. Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *International Conference on Computer Languages (ICCL)*, page 230, 1998.
13. Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Principles of Programming Languages (POPL)*, pages 107–120, 1998.
14. Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Programming Language Design and Implementation (PLDI)*, pages 289–299, 2003.
15. David J. Pereira R. Nigel Horspool and Bernhard Scholz. Fast Profile-Based Partial Redundancy Elimination. *Modular Programming Languages*, 2006.
16. G. Ramalingam. Data flow frequency analysis. In *Programming Language Design and Implementation (PLDI)*, pages 267–277, 1996.
17. Subhajit Roy and Y. N. Srikant. Profiling k-Iteration Paths: A Generalization of the Ball-Larus Profiling Algorithm. In *International Symposium on Code Generation and Optimization (CGO)*, pages 70–80. IEEE Computer Society, 2009.
18. Scale: A Scalable Compiler for Analytical Experiments. `http://www-ali.cs.umass.edu/Scale/`.
19. Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *International Symposium on Microarchitecture (MICRO)*, pages 172–181, 2001.
20. Sriraman Tallam, Xiangyu Zhang, and Rajiv Gupta. Extending path profiling across loop backedges and procedure boundaries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 251–264, 2004.
21. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.
22. Jingling Xue and Qiong Cai. A lifetime optimal algorithm for speculative PRE. *ACM Trans. Archit. Code Optim.*, 3(2):115–155, 2006.