Hierarchical Program Paths

CHUNBAI YANG, City University of Hong Kong
SHANGRU WU, City University of Hong Kong
W.K. CHAN†, City University of Hong Kong

Complete dynamic control flow is a fundamental kind of execution profile about program executions with a wide range of applications. Tracing the dynamic control flow of program executions for a brief period easily generates a trace consisting of billions of control flow events. The number of events in such a trace is large, making both path tracing and path querying incur significant slowdowns. A major class of path tracing techniques is to design novel trace representations that can be generated efficiently, and encode the inputted sequences of such events so that the inputted sequences are still derivable from the encoded and smaller representations. The control flow semantics in such representations have however become obscure, which makes implementing path queries on such a representation inefficient and the design of such queries complicated. We propose a novel two-phase path tracing framework — Hierarchical Program Path (HPP) — to model the complete dynamic control flow of an arbitrary number of executions of a program. In Phase 1, HPP monitors each execution, and efficiently generates a stream of events, namely HPPTree, representing a novel tree-based representation of control flow for each thread of control in the execution. In Phase 2, given a set of such event streams, HPP identifies all the equivalent instances of the same exercised inter-procedural path in all the corresponding HPPTree instances, and represents each such equivalent set of paths with a single subgraph, resulting in our compositional graph-based trace representation, namely HPPDAG. Thus, an HPPDAG instance has the potential to be significantly smaller in size than the corresponding HPPTree instances, and still completely preserves the control flow semantics of the traced executions. Control flow queries over all the traced executions can also be directly performed on the single HPPDAG instance instead of separately processing the trace representation of each execution followed by aggregating their results. We validate HPP using the SPLASH2 and SPECint 2006 benchmarks. Compared to the existing technique, named BLPT, HPP generates significantly smaller trace representations and incurs less slowdowns to the native executions in online tracing of Phase 1. The HPPDAG instances generated in Phase 2 are significantly smaller than their corresponding BLPT and HPPTree traces. We show that HPPDAG supports efficient backtrace querying, which is a representative path query based on complete control flow trace. Finally, we illustrate the ease of use of HPPDAG by building a novel and highly efficient path profiling technique to demonstrate the applicability of HPPDAG.

Categories and Subject Descriptors: • Software and its engineering ~ Runtime environments • Software and its engineering ~ Software testing and debugging

Additional Key Words and Phrases: Path tracing, inter-procedural path, hierarchical and compositional representation

ACM Reference Format:

DOI:http://dx.doi.org/10.1145/0000000.0000000

†All correspondences should be addressed to W.K. Chan.

This work is supported by the General Research Fund and Early Career Scheme of the Research Grants Council of Hong Kong (project numbers 11200015, 11201114, 111313, 125113, and 123512).

Author’s addresses: C. Yang, S. Wu and W.K. Chan, Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon Tong, Hong Kong; emails: chunbyang2@gapps.cityu.edu.hk, shangru.wu@my.cityu.edu.hk, wkchan@cityu.edu.hk.

Permission to make digital or hardcopies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credits permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2016 ACM 1539-9087/2016/0000000 $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

ACM Transactions on Software Engineering and Methodology, Vol. xx, No. x, Article xx, Publication date: Month YYYY
1. INTRODUCTION

Many program analysis and testing techniques require certain characterizations of the dynamic program behavior through various kinds of program profiles, such as control flow [Ball and Larus 1996; Zhang and Gupta 2001; Larus 1999; Renieris et al. 2005], data flow [Zhao et al. 2006; Tallam et al. 2005], and program states [Zhang and Gupta 2004] of program executions. Among them, program profiles on control flow capture the order of executed program instructions in each thread of control in a program execution. Such profiles have been widely utilized in compiler optimization [Ramamurthi et al. 2011], testing [Huang et al. 2013; Goel et al. 2003; Gligoric et al. 2013], debugging [Law and Rothermel 2003; Chilimbi et al. 2009; Xu et al. 2014] and execution replay [Altekar and Stoica 2009], for example. For the purpose of collecting the complete control flow trace [Larus 1999; Zhang and Gupta 2001; Tallam et al. 2005] of an execution, many path tracing techniques generate and collect additional control flow events, including basic block identifiers [Zhao et al. 2006], branches taken based on the conditionals [Renieris et al. 2005], and fine-grained acyclic path identifiers [Larus 1999]. They then compose the sequence of those collected events into a trace representation (or trace for short) of the program execution. Other techniques, for example, dynamic program slicing techniques [Agrawal and Horgan 1990], may then navigate on such a trace to extract their control flow facts of interest. However, as we present in Section 4 of this paper, tracing a native execution of a program for a brief period, such as for a period of 10 seconds using a commodity computer, easily generates billions of such control flow events. Representing a trace in a raw format such as representing it as a sequence of those events significantly slows down the execution under tracing, and generates a large trace.

Many prior works [Burtscher 2004; Kaplan et al. 2003; Ketterlin and Clauss 2008; Nevill-Manning and Witten 1997] study trace compression techniques to address the above problem. These techniques efficiently transform a sequence of control flow events into an encoded representation. Keeping the complete dynamic control flow using such an encoded representation can be up to two to three orders of magnitude smaller than using the raw format [Burtscher 2004]. However, to the best of our knowledge, the encoding schemes of almost all existing trace compression techniques aim at general-purpose compression, unaware of the control flow semantics of the trace to be encoded. For instance, a sequence of control flow events is simply considered as a sequence of unrelated symbols in the context-free grammar approach of Sequitur [Nevill-Manning and Witten 1997] and as a sequence of values in VPC [Burtscher et al. 2005] to be fed to the value predictors, which carries no specific meaning. Thus, after the encoding process, the control flow semantics may become obscure to subsequent path queries to be performed on the encoded traces. For example, consecutive trace events, modeling an inter-procedural path, may turn out to be encoded by several different nodes scattered across the context-free grammar graph of WPP [Larus 1999]. Querying on such encoded traces may then require non-trivial efforts to decode (and re-code) the representation and restore the original semantics of the relevant trace segments. On the other hand, some techniques [Zhang and Gupta 2001] try to alleviate the overhead on decoding the trace by optimizing on specific types of trace queries. However, when trace queries fall out of the targeted scope of these techniques, substantial decoding efforts are still required. For instance, TWPP [Zhang and Gupta 2001] groups procedure calls on the same procedure and each such group of traces is separately encoded by identifying repetitive patterns. Thus intra-procedural queries or queries on a limited set of procedures require only a subset of the whole trace to be decoded. However, for inter-procedural path queries, which may
involve a large number of different procedures, substantial decoding overhead is still unavoidable. In brief, these interesting compression techniques effectively reduce the size of the traces with their encoding schemes, but impose burdens on subsequent path queries over their encoded representations.

We observe that, both within a single control flow trace and across different control flow traces of a real-world program, there typically exists a large number of occurrences of the same inter-procedural paths (e.g., the so-called hot paths [Duesterwald and Bala 2000]), and inter-procedural paths are compositional, i.e., a longer path can be composed by several shorter ones. Our insight is that with a single representation of all the occurrences of the same inter-procedural path and the compositional reuse of such representations, the size of the whole control flow trace can be significantly reduced. At the same time, correctly ordered inter-procedural path representations can preserve the semantics of the original inter-procedural control flow.

In this paper, we propose a technique that innovatively transforms a set of complete control flow traces of a program into a single, efficient and compositional trace representation that preserves the semantics of the original control flow traces. As such, trace queries over the entire set of control flow traces can be directly and efficiently performed on this novel trace representation rather than separately performed on each control flow trace, which may contain a large amount of replicated instances of the same inter-procedural path among these traces. Specifically, we present a novel path tracing framework — Hierarchical Program Path (HPP). HPP captures the complete dynamic control flow of program executions, compositionally merge replications among instances of inter-procedural paths while maintaining the execution order of these subpath instances, and generates an efficient, directly queryable and graph-based trace representation. Our trace representation supports efficient and direct path queries over an arbitrary set of whole program execution traces of a program without any need of prior transformation or decoding of the representation.

The HPP framework adopts the path encoding scheme of the classic Ball and Larus (BL) algorithm [Ball and Larus 1996]. HPP consists of one online phase followed by an offline phase as depicted in Figure 1. In the trace collection phase (Phase 1), it models each program execution into a tree representation called HPPTree, and efficiently generates a stream of control flow events, as the post-order traversal of this HPPTree instance, online. Then, in the trace construction phase (Phase 2), HPP accepts a set of...

---

1 Without loss of generality, we assume the program is single-threaded. Later in Section 4.1, we will show how HPP can be easily extended to handle multi-threaded programs as we have done in our prototype implementation of the HPP framework.
event streams, and processes each event in each event stream one by one to iteratively identify and merge all the occurrences of the equivalent subtrees of the corresponding HPPTree instances into the same subgraph, resulting in our novel hierarchical trace representation — **HPPDAG**.

Our HPPTree model treats an entire program execution as an ordered tree, which is a representation of inter-procedural control flow. We show by experiment that this representation is efficient in representing the inputted sequence of control flow events. For example, for each *intra*-procedural acyclic path instance (namely a BL path instance [Ball and Larus 1996], and see the definition in Section 2) that contains at least one *call site*, i.e., instructions performing procedure calls, the HPPTree model represents this path instance by a single path event. In contrast, the same path instance is represented by multiple path events by the de facto path tracing technique for capturing the complete dynamic control flow of program executions [Larus 1999], named BLPT (Ball-Larus-based Path Tracing) in this paper. Nonetheless, modeling an execution with HPPTree also raises a new challenge of potential loss of records on path information. We address it by regaining the execution order among path instances through the notions of *watch point* and *call site alignment* (see Section 3.1).

In Phase 2, HPP accepts the event streams produced by Phase 1, identifies each set of equivalent subtrees in the corresponding HPPTree instances, and models each such set by a single subgraph in an HPPDAG instance, which is a directed acyclic graph representation (see Section 3.2). Moreover, in each subgraph of the HPPDAG instance, the caller and the corresponding callee procedures are explicitly inter-connected by the hierarchical structure of HPPDAG. This novel trace representation makes, for instance, control flow walkthrough [Ayers et al. 2005], calling context computation [Sumner et al. 2010], control dependency computation [Wang and Roychoudhury 2007], easy to access their control flow facts of interest. Furthermore, as all the occurrences of the same inter-procedural path represented by equivalent subtrees in HPPTree instances have been merged into the same (and single) representation, the computation or transformation on these occurrences, e.g., dependency chain generation, only requires a one-off effort rather than repetitive efforts on each occurrence to make the subsequent path queries efficient to be performed.

To evaluate the feasibility of HPP, we have implemented HPP on top of the LLVM framework [Lattner and Adve 2004] and evaluated both phases of HPP as well as the feasibility of performing path queries using HPPDAG on the SPLASH2 benchmark suite [Woo et al. 1995] and the SPECint 2006 benchmark suite [Henning 2006]. The experimental results show that, on average, BLPT generates 42.1% more BL path events and 28.2% larger trace logs, and runs 21% slower than Phase 1 of HPP. The BLPT trace logs are on average 72.19 folds the size of the corresponding HPPDAG trace logs. After compression with *gzip* [GZIP 2010], the BLPT traces are on average 44.29 folds the size of their HPPDAG counterparts. At the same time, the compression and decompression time of the BLPT traces are 59.92 and 65.25 folds that of the HPPDAG counterparts, respectively. This suggests HPP can be a good complement of existing compression techniques. We also assess the efficiency on performing backtrace querying on HPPDAG traces, and the empirical results show that such querying is significantly more efficient than performing the same querying on the corresponding BLPT traces by 75.61 folds in total on the two benchmark suites. Finally, we build a novel application of HPPDAG traces, namely a path profiling technique. The results show that this new technique can be straightforward in design, and can efficiently generate the path profile of a set of inter-procedural paths over an arbitrary set of executions. Besides, the paths in such a profile have no constraints on their lengths,
i.e., the number of comprising intra-procedural path segments and the number of involved procedure calls. To the best of our knowledge, this is intractable by prior path profiling techniques. It demonstrates the potentials of HPP to improve the productivity of tool developers and the quality of the tool developed.

The main contribution of this paper is threefold. (1) It presents a novel framework HPP to generate a novel, highly efficient and hierarchical representation of complete dynamic control flow, namely HPPDAG. (2) It reports an experiment that validates the effectiveness and efficiency of HPP. (3) It shows the feasibility to build a novel and efficient path profiling technique on top of HPPDAG. The path profiling technique can be both intuitive in design and efficient in identifying the execution frequencies of inter-procedural paths, which bear no length constraints, in a set of execution traces of the same program.

The rest of this paper is structured as follows. Section 2 presents the preliminaries relevant to our work. Section 3 presents the HPP framework and its two kinds of trace representations: HPPTree and HPPDAG. The evaluation of HPP is shown in Section 4. Section 5 presents a novel path profiling technique built on top of HPP. Section 6 discusses related work. Section 7 concludes this paper.

2. PRELIMINARIES

2.1 Ball and Larus Path Encoding Algorithm

An intra-procedural path is a sequence of instructions following the intra-procedural control flow specified by the control flow graph of a specific procedure. Whereas, an inter-procedural path is a sequence of instructions following both the intra-procedural control flow and the inter-procedural control flow among procedures specified by procedure call instructions so that each inter-procedural path may cross the procedure boundaries and can be as long as the whole control flow of an entire program execution.

Ball and Larus algorithm [Ball and Larus 1996] encodes intra-procedural paths for each procedure independently, and efficiently generates the identifier of each executed intra-procedural path instance during execution. It firstly constructs a directed acyclic graph $G_f = (V_f, E_f)$ for each procedure $f$. Each instruction $v$ in $f$ is represented as a unique node $v$ in $V_f$. Moreover, to model $f$ as a single-entry-single-exit procedure, $V_f$ also contains two special nodes $v_{\text{start}}$ and $v_{\text{end}}$ to denote the start node and the end node of $G_f$, respectively. The transfer from the instruction $v_i$ to the instruction $v_j$ according to the control flow of $f$, is represented by an edge $e = (v_i, v_j)$ in $E_f$. To ensure $G_f$ acyclic, for each back-edge $(v_i, v_j)$ of any loop in the control flow graph of $f$, two edges $(v_{\text{start}}, v_i)$ and $(v_j, v_{\text{end}})$ are added into $E_f$, and the back edge $(v_i, v_j)$ is excluded from $E_f$.

A path $p = (e_1, \ldots, e_n) \in G_f$ consists of a sequence of consecutive edges, where each edge $e_i \in E_f$. If a path $p$ starts at $v_{\text{start}}$ and ends at $v_{\text{end}}$ in $G_f$, i.e., $e_1 = (v_{\text{start}}, v_x)$ and $e_n = (v_y, v_{\text{end}})$, then such path is called a BL path. In particular, the former node $v_x$ in the last edge $e_n$ of a BL path is either the instruction reaching the procedure boundary to exit the procedure (e.g., the return instruction) or a node ended at a loop boundary (i.e., the node $v_x$ of a back-edge $(v_i, v_j)$). For ease of our presentation, we denote the set of all the BL paths in $G_f$ by $P_f$.

The algorithm then assigns each edge $e$ in $E_f$ with an integer denoted by increment($e$). Next, for each BL path $p$ in $P_f$, a path identifier $id_p$ of $p$ is computed as the sum of increment($e$) of all edges in $p$ (i.e., for each path $p$, $id_p = \sum_{e \in p} [\text{increment}(e)]$).

Theorem 1 below ensures that each BL path $p$ is assigned with a unique path identifier $id_p$ within a procedure $f$.

**Theorem 1** [Ball and Larus 1996]. $\forall p, p' \in P_f$ where $p \neq p'$, $id_{p'} \neq id_p$
Note that BL path is a static concept. We use the term **BL path instance** to refer to a dynamic instance of a BL path appearing in an execution trace.

To recognize the execution of each BL path instance $p_n$, the algorithm associates each procedure $f$ with a procedure-local variable $ps_f$, referred to as a **path state**. In the course of execution, $ps_f$ is initialized (or reset) to 0 at the start node $v_{\text{start}}$. When the execution traverses an edge $e$ of $p_n$, the variable $ps_f$ is incremented with the integer value $\text{increment}(e)$ associated with that edge $e$. If the execution reaches the end node $v_{\text{end}}$, the value kept by the variable $ps_f$ results in the unique path identifier of this executed BL path instance $p_n$.

Therefore, whenever the execution visits the end node $v_{\text{end}}$, a **BL path event** for the corresponding executed path instance $p_n$ is generated, which entails the path identifier $id_p$ of the BL path traversed.

Note that, if the above program execution crashes or is abruptly terminated (e.g., by calling the function $\text{exit}()$ in the C code), the execution may not have passed through all the edges in $p$ to reach the end node $v_{\text{end}}$. In this case, the path identifier $id_p$ of $p$ cannot be determined, and no corresponding BL path event is generated.

### 2.2 Path Tracing by BLPT

Larus [1999] extended the path encoding scheme [Ball and Larus 1996] presented in Section 2.1 to perform **path tracing**. To the best of our knowledge, this technique is the de facto approach to collecting the complete dynamic control flow of program executions. It is widely used in, for example, trace compression [Zhang and Gupta 2001; Larus 1999; Tallam et al. 2005] and trace analyses [Huang et al. 2013; Ohmann and Llibit 2013; Law and Rothermel 2003].

For ease of our presentation, we refer to the algorithm presented in [Larus 1999] as **BLPT** (Ball-Larus-based Path Tracing). In BLPT, a program execution is modeled as a sequence of BL path events ordered by their appearances in the execution. Besides, procedure entries and exits are also recorded to indicate the belonging procedure of each BL path.

To maintain the time order among path instances during tracing, BLPT revises the notion of BL path presented in Section 2.1 so that apart from procedure exits and back-edges, each call site also ends a BL path. As such, an inter-procedural path can be composed by the linearization of the revised BL paths. BLPT then records such linearization as a sequence of BL path events to represent the control flow of a program execution.

We use an example to illustrate the resultant trace representation produced by BLPT. Figure 2 depicts a program having two procedures $A$ and $B$. Procedure $A$ contains 8 instructions, and invokes itself at the instruction 4 and invokes procedure $B$ at the instruction 7. Consider the execution $tr$ executing the following sequence of instructions $(1, 2, 3, 4, 1, 2, 3, 5, 2, 6, 8, 5, 2, 6, 7, 9, 8)$. The resulting BLPT trace representation shown in Figure 3 is a sequence of BL path events nested in pairs of procedure entries and procedure exits as shown in the right most column in Figure 3.

We observe from the above example that BLPT is not efficient in representing inter-procedural paths with the linearization of the revised BL path instances. For example, as illustrated in Figure 3, the BL path $p_A^4$ (containing the call site of instruction 4) in the original path encoding algorithm [Ball and Larus 1996] is split into two revised BL paths ($p_{A_1}^4$ and $p_{A_2}^4$) in BLPT. This is because the linearization model of BLPT mandatorily requires that one revised BL path instance completes before the start of another revised BL path instance. Thus, their strategy of splitting an original BL path
Complex real-world programs are modular, and make many procedural calls in their executions. The above strategy of path splitting to achieve a sequential representation of the revised BL path instances leads to an excessive amount of path events generated.

For ease of our subsequent presentation, we refer to each BL path containing at least one call site in the original path encoding algorithm [Ball and Larus 1996] (see Section 2.1) as a **BL call path**. An exemplified BL call path is the BL path $p^A_1$ in the above example.

### 3. DESIGN OF THE HPP FRAMEWORK

In this section, we present the design of our HPP framework.

The execution trace $tr = \langle 1, 2, 3, 4, 1, 2, 3, 5, 2, 6, 8, 5, 2, 6, 7, 9, 8 \rangle$ can be represented as follows:

<table>
<thead>
<tr>
<th>Model</th>
<th>BL Path Identifier</th>
<th>Representation of $tr$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRPT</td>
<td>$p^A_1, p^B_2$</td>
<td>$(\text{enter}_A, p^A_1, \text{enter}_A, p^A_2, \text{exit}_A, p^A_3, \text{exit}_B, p^B_1, \text{exit}_A)$</td>
</tr>
<tr>
<td>HPPTree</td>
<td>$p^A_1, p^B_2$</td>
<td>$\langle \text{enter}_A, p^A_1, \text{exit}_A, p^A_2, \text{exit}_B, p^B_1, \text{exit}_A \rangle$</td>
</tr>
</tbody>
</table>

Fig. 3. The BRPT Model vs. the HPPTree Model.
The HPP framework consists of two phases as depicted in Figure 1. The purpose of Phase 1 (see Section 3.1) is to trace program executions and to model the execution trace of each program execution into an instance of the hierarchical trace representation, HPPTree. In the course of tracing, for each execution, HPP generates a stream of events online, which is equivalent to the post-order traversal of the HPPTree instance of that execution. Then, in Phase 2 (see Section 3.2), without the need to construct HPPTree instances explicitly, HPP iteratively processes the event streams generated by Phase 1, identifies repetitive occurrences of equivalent inter-procedural paths, recursively aggregates each set of equivalent paths into a single path representation, and reuses the representations of shorter subpaths in the representations of corresponding longer subpaths. As a result, a set of HPPTree instances are transformed into a single instance of HPPDAG. Each HPPDAG instance is an efficient, hierarchical, and compositional representation of complete dynamic control flow that allows control flow queries to be performed on it without beforehand transformation or decompression.

We note that unlike existing Ball-Larus-based tracing techniques [Larus 1999; Tallam et al. 2005], HPP follows the definition of BL path presented in [Ball and Larus 1996]. As such, for each BL call path, HPP uses exactly one BL path event to represent it.

### 3.1 Online HPPTree Trace Collection

In this section, we firstly present the base model of HPPTree, which is efficient on modeling inter-procedural paths, and leave two challenges on exception scenarios unhandled. Then, we present each of these two challenges and our refinement on the base model to address them. Finally, we summarize the final model of HPPTree, and introduce how HPP collects the trace events during program execution.

**3.1.1. Base Model of HPPTree.** Suppose that we want to monitor the control flow of an entire program execution of a program. Let the set of monitored procedures in this program be \( F_{\text{monitored}} \). For each procedure \( f \in F_{\text{monitored}} \), we assign it with a unique procedure identifier as \( id_f \in Fid \), where \( Fid \) is the set of all procedure identifiers. For each BL path \( p \) in any procedure \( f \) where \( f \in F_{\text{monitored}} \), we refer to its unique identifier as \( id_p \in Pid \), where \( Pid \) is the set of all BL path identifiers.

An HPPTree instance is an ordered tree built on top of the calling context tree [Sumner et al. 2010] of an execution trace. The base model of HPPTree includes two types of nodes: path node and procedural node.

Each occurrence \( p_n \) of a BL path \( p \) in the execution trace is represented by a path node \( n \) in an HPPTree instance, and the identifier \( id_p \) of \( p \) is taken as the label \( \text{label}(n) \) of node \( n \). For ease of reference, we sometimes refer to the BL path instance \( p_n \) as \( \text{path}(n) \).

Similarly, each procedure invocation \( f_n \) of a procedure \( f \in F_{\text{monitored}} \) in the execution trace is represented by a procedure call node \( n \) in an HPPTree instance, and the identifier \( id_f \) of \( f \) is taken as the label \( \text{label}(m) \) of node \( m \). We also sometimes refer to the procedure invocation \( f_n \) as \( \text{call}(n) \).

Each node \( n \) in an HPPTree instance has a sequence of nodes as its children, denoted by \( \text{Children}(n) \). The \( i \)-th child of \( n \) is the \( i \)-th node in the sequence \( \text{Children}(n) \). If \( n \) is a path node, \( \text{Children}(n) \) is the sequence of procedure call nodes representing the sequence of procedure calls issued from the BL path instance \( \text{path}(n) \). If \( n \) is a procedure call node, \( \text{Children}(n) \) is the sequence of path nodes representing the sequence of BL path instances visited in the procedure invocation \( \text{call}(n) \).
Figure 3 also illustrates the ordered tree instance of the HPPTree model for the execution trace $tr$ described in Section 2.2. We use circles and squares to denote the procedure call nodes and path nodes in this HPPTree instance, respectively. For the ease of our presentation, in the HPPTree instance shown in Figure 3, we refer to each path node by its label. The two procedure call nodes labelled with $A$ are referred to as $n_{A0}$ and $n_{A1}$ from top to bottom and the procedure call node labelled with $B$ is referred to as $n_B$. The entry of $tr$ is the first invocation of the procedure $A$, which is represented by the procedure call node $n_{A0}$. $Children(n_{A0})$ is the sequence $(p_1^A, p_2^A)$, representing the two BL path instances executed in the first invocation of $A$. During the execution of $path(p_1^A)$, the invocation $call(n_{A1})$ is issued at the instruction 4 to execute $path(p_2^A)$ and $path(p_3^A)$. Therefore, $Children(p_1^A)$ is a singleton sequence $(n_{A1})$, and $Children(n_{A1})$ is $(p_2^A, p_3^A)$. Similarly, $Children(p_2^A)$ is $(n_B)$, and $Children(n_B)$ is $(p_4^B)$. As such, by a post-order traversal of the HPPTree instance, we retrieve the control flow trace $tr$ of the execution.

Each of the call paths $p_1^A$ and $p_2^A$ in the above HPPTree instance is encoded as two different BL paths in the corresponding BLPT model (i.e., $p_1^A$ and $p_2^A$ instead of $p_1^A$, $p_2^A$, and $p_3^A$ instead of $p_3^A$). As such, the modeling of a BL call path as a single BL path in HPPTree results in fewer BL path events generated. Moreover, in HPPTree, BL path instances of a caller procedure are explicitly linked with those of the corresponding callee procedures.

Nonetheless, by using this strategy, the generated BL paths are no longer totally ordered as what they appear in the corresponding BLPT instances. For example, for the BL path instance $p_1^A$ in Figure 3, among all its five instructions, instructions 1 to 4 are visited before both $p_1^A$ and $p_2^A$ start, and instruction 5 is visited after $p_1^A$ and $p_2^A$ has ended.

Suppose that there is a BL path instance $p_n$ represented by a path node $n$ in an HPPTree instance. Our insight is that the exact order among instructions in $p_n$ and the procedure calls represented by $Children(n)$ can be implicitly modeled in the HPPTree instance. That is, the control flow represented by the subtree rooted at a child node of $n$ must be issued at a specific call site in $p_n$. As such, we specify that each child node of the path node $n$, say child, aligns with a specific call site occurrence in $p_n$, referred to as $callsite(child)$, such that $call(child)$ happens after $callsite(child)$ and before the next instruction of $callsite(child)$ in $p_n$. This property is referred to as call site alignment, which maintains the instruction order of control flow in an HPPTree instance.

In the base model of HPPTree, the call site alignment property is guaranteed by the following approach. By traversing the sequence of program instructions of the BL path instance $p_n$, we obtain the sequence of call site instructions along $p_n$. This sequence forms a one-one correspondence with the sequence of procedure calls represented by the procedure call nodes in $Children(n)$. As such, the $i$-th child node, say child$_{i}$, in $Children(n)$ aligns with the $i$-th call site occurrence, say $cs_{i}$, in $p_n$, i.e., $callsite(child_{i}) = cs_{i}$. For example, in the HPPTree instance in Figure 3, node $n_B$ aligns with the call site in the BL path instance $p_4^B$, i.e., the instruction 7. Thus, we can determine that the invocation of procedure $B$ (i.e., $call(n_B)$) is executed at the instruction 7 (and before the execution of the instruction 8) in $p_4^B$.

Therefore, with the call site alignment property, HPP maintains the instruction order of control flow in an HPPTree instance, and represents each BL call path instance using only one path event. However, two challenges faced by the above base model of HPPTree should be addressed. They are presented in the next two subsections.
3.1.2 Early Termination. A BL path event for a procedure \( f \) is generated only at the time when its corresponding BL path instance reaches its end, that is, at the time that \( v_{\text{end}} \) (i.e., the end node in \( G_f \)) of the procedure \( f \) (see Section 2.1) is visited. Thus, when the execution is aborted due to some reasons (e.g., crash), the corresponding path events of all BL path instances that have not reached their end nodes cannot be generated. We refer to this scenario as \textit{early termination}.

Moreover, given the directed acyclic graph \( G_f \) of a procedure \( f \) produced by the path encoding algorithm [Ball and Larus 1996] (see Section 2.1), we refer to a path starting from \( u_{\text{start}} \) and ending at some node \( u_{\text{target}} \), which is not \( v_{\text{end}} \), as an \textit{unfinished path}. Note that each unfinished path is also the prefix of some BL path(s) in the set \( P_f \).

When an occurrence of early termination appears, for each procedure call on the current call stack of the execution, there is one instance of unfinished path under traversal. Additionally, the control flow information of these unfinished path instances has not been recorded yet because no trace event has been generated. An example of this type of scenario is shown in Figure 4. The trace \( tr' = \langle 1, 2, 3, 4, 1, 2, 3, 5, 2, 6, 8, 5, 2, 6, 7, \text{crash} \rangle \) crashes when it is executing an instance of the BL path \( p_{i1} \) in the procedure \( B \). At the moment of the execution crash, the call stack of the execution contains two active procedure calls \( \langle A, B \rangle \). In other words, there are two unfinished path instances, one for each of the two procedure calls, currently under traversal.

In Figure 4(a), we use two question marks to label the path nodes of these two unfinished path instances in the HPPTree instance for the trace \( tr' \). (Readers may wish to compare this HPPTree instance with that in Figure 3, which is produced by a trace without encountering any early termination.)

Our insight is that uniquely identifying each unfinished path instance is possible, as a consequence of Theorem 2 below.

\textbf{Theorem 2.} Given the directed acyclic graph \( G_f \) for a procedure \( f \), an unfinished path from \( u_{\text{start}} \) to \( u_{\text{target}} \) can be uniquely determined by the path state kept by \( ps_f \) at the node \( u_{\text{target}} \).
PROOF. We prove the theorem by contradiction to show it is not possible to have two unfinished paths from $u_{\text{start}}$ to $u_{\text{target}}$ having the same path state kept by $ps_f$ at the node $u_{\text{target}}$. Suppose that there are two paths $p_1$ and $p_2$ from $u_{\text{start}}$ to $u_{\text{target}}$ having the same path state kept by the variable $ps_f$ at node $u_{\text{target}}$. First, suppose that $u_{\text{target}} = u_{\text{end}}$, then $p_1$ and $p_2$ are two BL paths with the same path identifier, i.e., $id_{p_1} = id_{p_2} = ps_f$, which contradicts to Theorem 1. Second, suppose that $u_{\text{target}} \neq u_{\text{end}}$, we arbitrarily select a subpath $p_3$ connecting from $u_{\text{target}}$ to $u_{\text{end}}$. Thus, we can construct a pair of BL paths $p = p_r \downarrow p_3$ and $p' = p_r \downarrow p_3$ such that $id_p = \sum_{e \in p_1}[\text{increment}(e)] + \sum_{e \in p_2}[\text{increment}(e)]$, and $id_{p'} = \sum_{e \in p_2}[\text{increment}(e)] + \sum_{e \in p_3}[\text{increment}(e)]$. Because $\sum_{e \in p_1}[\text{increment}(e)] = ps_f = \sum_{e \in p_1}[\text{increment}(e)]$, therefore, we have $id_p = id_{p'}$ which also contradicts to Theorem 1.

Based on Theorem 2, we can uniquely identify (i.e., retrieve) any unfinished path instance in any procedure $f$, providing that both the ending node $u_{\text{target}}$ and the path state $ps_f$ at $u_{\text{target}}$ are available.

However, an early termination may happen at an arbitrary execution point, due to reasons like pointer dereferencing and integer division for example. Nonetheless, monitoring all program statements of such execution points may generate prohibitive runtime overhead. We thus propose the notion of watch point as a lightweight alternative.

A watch point is defined as a static program location in a procedure $f$ under monitoring (i.e., $f \in F_{\text{monitored}}$). Each watch point is assigned with a unique identifier $id_w \in Wid$, where $Wid$ is the set of all watch point identifiers. When a particular watch point $w$ in a procedure $f$ is visited in an execution, HPP records the identifier $id_w$ of $w$ and the current path state $ps_f$ (see Section 2.1) of the invoked instance of $f$. This couple $(id_w, ps_f)$ is referred to as a bookkeeping state.

Moreover, for each unfinished path instance, based on Theorem 2, HPP only needs to keep the bookkeeping state of its most recently visited watch point, if any, and this record will be erased when the unfinished path instance turns into a BL path instance. Specifically, a bookkeeping state provides the information about a node $u_{\text{target}}$ (determined by $id_w$) and $ps_f$. As such, when an occurrence of early termination happens, for any unfinished path instance $u$, a prefix of $u$, which is also an unfinished path instance but ends at the most recently visited watch point of $u$, can be precisely determined according to Theorem 2.

When an occurrence of early termination happens, each active procedure call in the current call stack of the execution, except the one on the top of the call stack, is suspended at exactly one call site waiting for the return of the corresponding issued procedure call placed on top of it in the call stack. For example, for the execution trace $tr'$ in Figure 4, the first call to the procedure $A$ from the procedure $main()$ is suspended at the call site, i.e., instruction 7, and waits for the return from the call to the procedure $B$. For ease of reference, we refer to the current call stack as $(A, B)$.

HPP places one watch point right before each call site in the monitored procedures of the program. Besides, users of HPP may extend this set of watch points to include other watch points at places of interest to them. For the program in Figure 2, two watch points $w_1$ and $w_2$ are placed right before the call sites at the instructions 4 and 7, respectively.

With the above design of watch points, when an occurrence of early termination happens, HPP ensures the following:
(1) For the unfinished path instance \( p_u \) of each procedure call in the current call stack of the execution, except for the topmost procedure call, \( p_u \) is recorded in the HPPTree instance.

(2) For the unfinished path instance \( p_u \) of the topmost procedure call on the current call stack of the execution, if there exists a most recently visited watch point \( w \) of \( p_u \), a prefix of \( p_u \) ending at \( w \) is recorded in the HPPTree instance.

To accommodate unfinished paths, we refine the base model of HPPTree by introducing a new type of node named unfinished path node. Each unfinished path node \( u \) represents an unfinished path instance \( p_u \), and the bookkeeping state \((id_w, ps_w)\) of the most recently visited watch point \( w \) of \( p_u \) is taken as the label of \( u \). In other words, each unfinished path node \( u \) is a path node having \( label(u) = (id_w, ps_w) \) (instead of the identifier of some BL path).

As depicted in Figure 4, the two watch points \( w_1 \) and \( w_2 \) placed in the program have been traversed at the execution points of the trace \( tr' \), indicated by the arrows right below the trace. Note that both \( w_1 \) and \( w_2 \) are placed in procedure \( A \). At the moment \( tr' \) crashes, the current call stack of the execution is \((A, B)\). For the unfinished path instance of procedure \( A \), say \( u_A \), its most recently visited watch point holds the bookkeeping state \((id_{w2}, ps_{w2})\). Thus, an unfinished path node with the bookkeeping state \((id_{w2}, ps_{w2})\) models \( u_A \) in the HPPTree instance shown in Figure 4(b). However, for the unfinished path instance of procedure \( B \), say \( u_B \), it has not reached any watch point. Thus, no control flow information about \( u_B \) is recorded in the HPPTree instance.

3.1.3 Unmonitored control flow. The second challenge addressed by HPP is on the part of the control flow of an execution that has not been monitored. In other words, some procedure \( f \notin F_{monitored} \) may be invoked in an execution. It is because assuming all procedures able to be monitored in an execution is too restrictive. For instance, in some scenarios, both the source code and the binaries of some system functions may not be available on the user’s site (or the tracing is conducted on a public cloud platform). Sometimes, it may be unnecessary (from user’s viewpoint) to instrument and monitor a particular procedure. For instance, a real-world program may heavily rely on libraries that are of no interest to subsequent trace analysis techniques on analyzing the dynamic behavior of the program.

We recall that the call site alignment of the HPPTree model requires that each child of a path node \( n \) aligns with a call site occurrence in the BL path instance \( path(n) \). In the base model of HPP, call site alignment is guaranteed because the \( i \)-th child node, say \( child_i \), of \( n \) aligns with the \( i \)-th call site occurrence, say \( cs_i \), in \( path(n) \), i.e., \( cs_i = callsite(child_i) \). The presence of unmonitored control flow in an execution, however, may invalidate this one-one correspondence, and thus fails the call site alignment property.

On the one hand, a call site occurrence may not be aligned with any procedure call node. If a call site occurrence \( cs \) in a BL path instance issues a procedure call to an unmonitored procedure \( f \notin F_{monitored} \), this call site occurrence \( cs \) is unable to be aligned with any procedure call node because each procedure call node only models an invocation of a monitored procedure in an HPPTree instance. On the other hand, a procedure call node may not align with any call site occurrence. A monitored procedure call, say, represented by a procedure call node \( n \) in an HPPTree instance, may be issued from an unmonitored procedure (e.g., a monitored callback procedure is invoked by an unmonitored library call). Thus no call site occurrence can be aligned with this node \( n \) in the HPPTree instance.

Consider the exemplified program in Figure 5. The procedure \( main \) contains four call sites, which are located at instructions 2, 3, 4, and 5, denoted as \( cs_2, cs_3, cs_4, \) and
Suppose that we are able to statically determine that \( cs_2 \), \( cs_4 \) and \( cs_5 \) invoke the procedures \( X \), \( Y \) and \( Z \), respectively, where \( X \in F_{monitored} \) and both \( Y \) and \( Z \notin F_{monitored} \). At the same time, \( cs_3 \) performs a procedure call through a function pointer that cannot be statically determined to invoke a monitored procedure. Let us consider an execution trace \( tr'' \) depicted in Figure 5(a), which starts with the invocation of the procedure \( main \). During the execution of the BL path instance of \( main \), denoted by \( p_{main} \), the call site occurrence of \( cs_3 \) invokes some unmonitored procedure, and the procedure invocation on \( Y \) issued from the occurrence of \( cs_4 \) further invokes \( X \). For ease of our subsequent discussion, we use \( p_X \) and \( p'_X \) to denote the BL path instances of the two invocations on \( X \) in \( tr'' \). Figure 5(b) illustrates the HPPTree instance of \( tr'' \) using the base model of HPPTree. We observe that the four occurrences of call sites in \( p_{main} \) cannot form one-one correspondences with the two procedure call nodes labelled with \( X \). Specifically, we cannot determine whether the second procedure call node in the illustrated HPPTree instance aligns with which of \( cs_3 \), \( cs_4 \), and \( cs_5 \).

HPP addresses the above illustrated problem due to unmonitored control flow so that it is able to handle real-world programs and support customized program tracing. HPP classifies all the call sites into three disjoint sets, namely **monitored call sites**, **unmonitored call sites** and **indirect call sites**. The classification is as follows: Suppose that there is a call site \( cs \) in a BL path of a program. If \( cs \) invokes a procedure through function pointer, we refer to \( cs \) as an indirect call site. Otherwise, if the procedure \( f \) invoked from \( cs \) does not belong to the set of monitored procedures, i.e., \( f \notin F_{monitored} \), we refer to \( cs \) as an unmonitored call site. Finally, if \( cs \) is neither an unmonitored call site nor an indirect call site, then \( cs \) is referred to as a monitored call site. We note that HPP does not require the targeted procedures of indirect call sites to be statically determined, and thus whether an indirect call site will call a monitored or an unmonitored procedure is not known before execution.

For example, in Figure 5, \( cs_2 \) is a monitored call site, \( cs_3 \) is an indirect call site, and both \( cs_4 \) and \( cs_5 \) are unmonitored call sites, which invoke unmonitored procedures \( Y \) and \( Z \), respectively.
From the example shown in Figure 5, we can see that procedure calls represented by procedure call nodes may be indirectly invoked from some unmonitored procedure calls issued by unmonitored call sites or indirect call sites (e.g., the second call to X is issued indirectly from the unmonitored call site cs).

To solve the above problem, we further refine the base model of HPPTree by adding two more types of tree node, namely unmonitored call site node and indirect call site node. An unmonitored (or indirect) call site node ncs explicitly models the indirect calling relation between an unmonitored (or indirect) call site and procedure calls, which are represented by the children of ncs. In short, ncs represents the monitored control flow issued, directly or indirectly, from its aligned call site occurrence callsite(ncs).

The modeling of control flow semantics for these two additional types of nodes is as follows. Suppose that in an execution trace, an unmonitored call site occurrence, say cs, of a BL path instance p, invokes an unmonitored procedure instance, say f. During the execution, HPP maintains the index i of cs in p, which is to represent that cs is the i-th encountered unmonitored call site occurrence in p. Whenever f or any procedure call indirectly issued from f (i.e., the procedures calls having been atop f in the call stack of the execution before f returns) invokes any monitored procedure, an unmonitored call site node ncs is created to model cs in the HPPTree instance, and we set label(ncs) = i. Otherwise, cs is not modeled. On the other hand, each occurrence of an indirect call site is modeled by one indirect call site node ncs, with an empty label (i.e., label(ncs) = ⊥), in the HPPTree instance.

The parent-children relationship of these two types of nodes is as follows. Suppose that there is a path node np and an occurrence of either an unmonitored call site or an indirect call site, say cs, in the BL path instance path(np). Moreover, cs is modeled by either an unmonitored call site node or an indirect call site node, say ncs, accordingly. As such, we have ncs ∈ Children(np). Moreover, Children(ncs) contains a list of procedure call nodes, representing the monitored procedure call(s) issued from cs, either through invocations of unmonitored procedures or through function pointers (indirectly).

With the introduction of unmonitored call site node and indirect call site node, the call site alignment property is guaranteed by the following approach, which is different from that described in the base model of HPP in Section 3.1.1.

Suppose there is a path node n and a child node m in Children(n). If m is an unmonitored call site node and label(m) = i, callsite(m) is the i-th unmonitored call site occurrence in path(n). If m is the i-th indirect call site node in Children(n), callsite(m) is the i-th indirect call site occurrence in path(n). If m is the i-th procedure call node in Children(n), callsite(m) is the i-th monitored call site occurrence in path(n).

The difference in handling indirect call site node and unmonitored call site node is based on the insight that an unmonitored call site, which typically invokes a library call or system call, usually will not further invoke monitored procedures. Thus, we do not model every occurrence of unmonitored call sites by one unmonitored call site node, as presented above, to reduce the number of nodes in the HPPTree instance as well as the number of trace events generated during the tracing in Phase 1.

Figure 5(c) illustrates the revised approach to guarantee call site alignment in an HPPTree instance. We name the node labelled pmain as nmain. The three children of nmain as child1, child2 and child3, respectively. Node child1 is the first procedural call node in Children(nmain) so that it aligns with the first monitored call site occurrence cs1 in pmain, i.e., callsite(child1) = cs1. Node child2 is the first indirect call site node in Children(nmain) so that it aligns with the first indirect call site occurrence cs2 in pmain, i.e., callsite(child2)
= cs3. Node child2 is an unmonitored call site node with 1 as its label so that it aligns with the first unmonitored call site occurrence cs4 in pmain, i.e., callsite(child2) = cs4. We note that no monitored control flow is issued directly or indirectly from cs3 or cs5. Thus, child2 has no child, and the unmonitored call site cs5 is not modeled by any node in the HPPTree instance. As such, each child node of any path node in the HPPTree instance aligns with a specific call site occurrence so that the call site alignment property holds.

3.1.4 HPPTree and Trace Data Collection. In this subsection, we summarize our final HPPTree model, and explain how an HPPTree instance models the complete dynamic control flow of an execution trace.

An HPPTree instance is an ordered tree, each node of which has two attributes, namely type and label. Consider a node n in an HPPTree instance where the instance models an execution trace \( \tau \). We denote the type of the node n by \( \text{type}(n) \) and the label of n by \( \text{label}(n) \). There are five types of nodes in the HPPTree model, namely path node, procedure call node, unfinished path node, unmonitored call site node, and indirect call site node. For ease of reference, we denote them as \( \alpha, \beta, \chi, \delta, \varepsilon \), respectively. That is \( \text{type}(n) \in \{\alpha, \beta, \chi, \delta, \varepsilon\} \). We denote the subtree rooted at n in an HPPTree instance by \( T_n \). The node n has a (possibly empty) sequence of nodes as its children, denoted by Children(n). We refer to the \( i \)-th node in Children(n) as \( Children(n)[i] \). Moreover, let \( F_{\text{monitored}} \) denote the set of monitored procedures in the program under tracing. The following properties on modeling the execution trace hold.

- If \( \text{type}(n) = \alpha \), then \( \text{label}(n) = id_p \), where \( n \) represents a BL path instance \( p \) in \( \tau \), and \( id_p \) is the path identifier of \( p \). For the child \( ni = Children(n)[i] \), \( \text{type}(n) \) is an element in the set \( \{\beta, \delta, \varepsilon\} \). Specifically, if \( \text{type}(n) = \beta \), \( ni \) represents the procedure call of a monitored procedure that aligns with an occurrence of a monitored call site in \( p \); if \( \text{type}(n) = \delta \), \( ni \) represents an unmonitored call site occurrence in \( p \); and if \( \text{type}(n) = \varepsilon \), \( ni \) represents an indirect call site occurrence in \( p \).
- If \( \text{type}(n) = \beta \), then \( \text{label}(n) = id_f \), where \( n \) represents an invocation of the monitored procedure \( f \) in \( \tau \) (i.e., \( f \in F_{\text{monitored}} \)), and \( id_f \) is the unique procedure identifier of \( f \). For each child \( ni = Children(n)[i] \), \( \text{type}(ni) \) is \( \alpha \) or \( \chi \). If \( \text{type}(ni) = \alpha \), \( ni \) represents the \( i \)-th BL path instance executed in this invocation of \( f \); if \( \text{type}(ni) = \chi \), \( ni \) must be the last child in \( Children(n) \) and represents an unfinished path instance.
- If \( \text{type}(n) = \chi \), then \( \text{label}(n) = (id_u, ps_u) \), where \( n \) represents an unfinished path instance \( u \) in \( \tau \) and \( (id_u, ps_u) \) is the bookkeeping state of the most recently visited watch point of \( u \). \( Children(n) \) shares the same modeling as the path node presented above.
- If \( \text{type}(n) = \delta \), then \( \text{label}(n) = j \), where \( n \) represents an unmonitored call site occurrence \( cs \) in \( \tau \), and \( j \) is the index of \( cs \) (i.e., \( cs \) is the \( j \)-th unmonitored call site occurrence in the BL path instance containing \( cs \)). For each child \( ni = Children(n)[i] \), \( \text{type}(ni) = \beta \), and \( ni \) represents the \( i \)-th procedure call of a monitored procedure indirectly invoked from \( cs \).
- If \( \text{type}(n) = \varepsilon \), then we do not assign the node with any label (denoting as \( \text{label}(n) = \bot \)), where \( n \) represents an indirect call site occurrence \( cs \) in \( \tau \) and \( \bot \) denotes empty. For each child \( ni = Children(n)[i] \), \( \text{type}(ni) = \beta \), and \( ni \) represents the \( i \)-th procedure call of a monitored procedure indirectly invoked from \( cs \).
To construct an HPPTree instance from an execution, HPP collects in total eight types of trace events, and each trace event is an operand-opcode pair as summarized in Figure 6.

These events are as follows: For each monitored procedure \( f \in F_{monitored} \), an event \( enter\_proc:id_f \) is generated at the entry of each invoked instance of the procedure \( f \), where \( id_f \) is the identifier of \( f \), and an event \( exit\_proc \) is generated right before the exit of the invoked procedure. An event \( bl\_path:id_p \) is generated at the end of each BL path instance (either at a back-edge or at the procedure exit as stated in Section 2.1).

During the period of invocation of a procedure from an unmonitored call site, if a monitored procedure is called, right before the invocation of such monitored procedure and right after the occurrence of this unmonitored call site, an event \( enter\_unmonitored\_cs \) and an event \( exit\_unmonitored\_cs \) are generated, respectively. Right before and right after each occurrence of any indirect call site, an event \( enter\_indirect\_cs \) and an event \( exit\_indirect\_cs \) are generated, respectively. A pair of \( enter\_unmonitored\_cs \) and \( exit\_unmonitored\_cs \) events helps record the parent-child relationship between an unmonitored call site node and the procedure call nodes representing callback function calls issued indirectly from the unmonitored call site. It is because the call back functions must happen in between this pair of events. The same can be interpreted for a pair of \( enter\_indirect\_cs \) and \( exit\_indirect\_cs \) events.

When HPP detects an early termination occurrence, each monitored procedure in the current call stack records the bookkeeping state of its unfinished path instance as an trace event \( unfinished\_path:id_{psw} \) starting from the topmost procedure in the call stack until (and including) the procedure at the bottom, one after one.

The above generated event stream explicitly follows a nested structure. For instance, for each invoked instance of a procedure \( f_m \), all BL path events generated in between the entrance and the exit of that instance of \( f_m \) are nested in between the corresponding \( enter\_proc \) event and \( exit\_proc \) events of the same instance of \( f_m \). As such, HPP generates a stream of events, and that event stream corresponds to the post-order traversal of an HPPTree instance of the execution.

For instance, the event stream corresponding to the HPPTree instance shown in Figure 3 is \( \langle enter\_proc:id_A, enter\_proc:id_A, bl\_path:p_A^4, bl\_path:p_A^4, exit\_proc, bl\_path:p_A^4, enter\_proc:id_B, bl\_path:p_B^6, exit\_proc, bl\_path:p_B^6, exit\_proc \rangle \). If an occurrence of early termination happens as illustrated in Figure 4, then the event stream with the bookkeeping state illustrated in Figure 4(b) is \( \langle enter\_proc:id_A, enter\_proc:id_A, bl\_path:p_A^4, bl\_path:p_A^4, exit\_proc, bl\_path:p_A^4, enter\_proc:id_B, unfinished\_path:id_{w2}, ps_{w2} \rangle \). Similarly, the event stream for Figure 5(c) with the illustrated occurrences of unmonitored call site and indirect call site is \( \langle enter\_proc:id_{main}, enter\_proc:id_X, \ldots \rangle \).
3.2 HPPDAG Construction

In this section, we present Phase 2 of HPP.

Each subtree in an HPPTree instance models a segment of the inter-procedural control flow, which may exhibit multiple occurrences both in its belonging execution trace and across different execution traces of the same program. In Phase 2, HPP identifies and merges all equivalent subtrees appearing in the entire set of HPPTree instances, and reuses the representations of shorter subpaths when constructing those of longer inter-procedural paths. As such, the set of HPPTree instances is transformed into a directed acyclic graph, referred to as HPPDAG.

Consider a node \( m \) in an HPPDAG instance \( G_0 \), which is transformed from a set of HPPTree instances \( I \). The node \( m \) has the same attributes, which are \( \text{type}(m) \) and \( \text{label}(m) \), as an HPPTree node defined in Section 3.1.4. The node \( m \) has a sequence of HPPDAG nodes as its children, denoted by \( \text{Children}(m) \). We refer to the \( i \)-th node in \( \text{Children}(m) \) as \( \text{Children}(m)[i] \). The subgraph that consists of \( m \) as well as all the descendants of \( m \) in the HPPDAG instance is denoted as \( G_m \). We also refer to \( m \) as the root of \( G_m \) for consistency with the HPPTree model.

For ease of presentation, we define the following auxiliary terminology:

- **Subtree equivalence relation (\( \equiv \)):** Two subtrees \( T_a \) and \( T_b \) are equivalent (denoted as \( T_a \sim T_b \)) if and only if (1) \( \text{type}(a) = \text{type}(b) \) and \( \text{label}(a) = \text{label}(b) \), and (2) for each node \( a' = \text{Children}(a)[i] \) and each node \( b' = \text{Children}(b)[j] \), if \( i = j \), then \( T_{a'} \sim T_{b'} \).

- **Equivalence class of subtrees:** Given the set of subtrees \( \Phi \) in all the HPPTree instances in \( I \) and the subtree equivalence relation \( \sim \), \( \Phi \) is partitioned into a set of equivalence classes. The equivalence class of a subtree \( T_n \) is denoted as \( [T_n] \), where \( [T_n] = \{ T \in \Phi \mid T \sim T_n \} \).

- **Subgraph representation relation (\( \supseteq \)):** For each equivalence class \( [T_i] \) in \( \Phi \), a subgraph \( G_m \) of \( G_0 \) represents \( [T_i] \), denoted as \( G_m \supseteq [T_i] \), if and only if \( \forall T_n \in [T_i], (1) \text{type}(n) = \text{type}(m) \) and \( \text{label}(n) = \text{label}(m) \), and (2) for each node \( n' = \text{Children}(n)[i] \) and each node \( m' = \text{Children}(m)[j] \), if \( i = j \), then \( G_{n'} \supseteq G_{m'} \).

- **Subgraph equivalence relation (\( \approx \)):** If \( G_m \supseteq [T_i] \) and \( G_{m'} \supseteq [T_j] \), then we say \( G_m \) is equivalent with \( G_{m'} \), denoted as \( G_m \approx G_{m'} \).

In the HPPDAG instance \( G_0 \), for each subtree \( T_n \) in \( I \), there exists one and only one subgraph, denoted as \( G_m \), where \( G_m \approx [T_n] \). Moreover, for each subgraph \( G_{m'} \) in \( G_0 \), there must exist a subtree \( T_n' \) in \( I \), where \( G_{m'} \approx [T_n'] \). As such, each equivalence class of subtrees in \( I \) is aggregated into one single subgraph in \( G_0 \).

Figure 7 illustrates two HPPTree instances and its corresponding HPPDAG instance after the merging of the equivalent subtrees in these HPPTree instances. In the figure, 2, 2, 3, 4, 4, and 4 repetitive subtrees rooted at nodes labeled with \( f_{1b} \), \( p_{b1} \), \( p_{a2} \), \( f_{1} \), \( p_{c1} \), and \( p_{c2} \) are merged as subgraphs \( G_{fb}, G_{p_{b1}}, G_{pa2}, G_{fc}, G_{p_{c1}}, \) and \( G_{p_{c2}} \), in the HPPDAG instance, respectively. For brevity, we refer to a root node by its label in the HPPDAG instance in Figure 7.

As such, the HPPDAG instance with 11 nodes, i.e., 11 subgraphs, in total is smaller than the HPPTree instance with 24 nodes, i.e., 24 subtrees, in total. Moreover, this HPPDAG instance preserves the hierarchical structure and the order of edges under each node of each HPPTree instance. This design allows path queries (e.g., retrieval of an intra/inter-procedural path starting at an arbitrary node) to be directly operated on.
an HPPDAG instance as if they were performed on the corresponding set of HPPTree instances.

Moreover, the aggregation of equivalent subtrees enables direct reuse of computation results on equivalent path segments. For example, the control dependency chains [Wang and Roychoudhury 2007] calculated based on a subgraph $G_m$ only requires a one-off effort rather than repetitive calculations on each subtree $T_n \in [T_x]$ where $G_m \approx [T_x]$. HPP generates the HPPDAG instance on-the-fly directly from the event streams of HPPTree instances (i.e., there is no need to construct any HPPTree instances beforehand). Specifically, in the course of HPPDAG construction, a \textit{working copy of the subgraph} $G_m$ such that $G_m \approx [T_n]$ is maintained on-the-fly while processing the event sequence of a subtree $T_n$. Whenever the processing of the event sequence for $T_n$ is completed, HPP checks whether there exists a subgraph $G_m'$ in the current HPPDAG instance such that $G_m' \approx G_m$. If $G_m'$ does not exist, then $G_m$ is no longer treated as a working copy, and is taken as the subgraph in the HPPDAG instance to represent $[T_n]$. On the other hand, if $G_m'$ exists, an equivalent subtree of $T_n$ must have already been encountered before and is represented by $G_m'$. Thus, all references to $G_m$ in the current HPPDAG instance will be referred to $G_m'$, and $G_m$ is discarded. This ensures that all subtrees in the same equivalent class in all given HPPTree instances are aggregated into the same subgraph in the HPPDAG instance under construction.

To facilitate the uniqueness check, we introduce the notion of \textit{subgraph signature}. The signature of a subgraph $G_m$ is defined as a triple $\text{sig}(m) = (\text{type}(m), \text{label}(m), \text{Children}(m))$. It is straightforward to observe that if $\text{sig}(m') = \text{sig}(m)$, then $G_m \approx G_m'$.

Algorithm 1 presents the HPPDAG construction algorithm. It accepts a set of event streams (see Figure 6 for the format) as input, and produces an HPPDAG instance. For simplicity, we create a phony node named \textit{root} as the single entry of the HPPDAG instance (i.e., \textit{root} has no predecessor). For each child node of \textit{root}, the subgraph rooted at it represents an HPPTree instance whose event stream has been processed.

The algorithm needs to maintain a few data structures: it maintains a map \textit{sigToGraph} mapping a subgraph signature to the root node of the corresponding subgraph in the HPPDAG instance under construction (line 1). It also maintains a stack \textit{stateStack} to keep records of the working copies of subgraphs representing those

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{\textbf{(a)} HPPTree instances \quad \textbf{(b)} HPPDAG instance}
\end{figure}
unfinished subtrees whose nodes are under processing (line 2). This is because subtrees are processed in a post order traversal manner such that the processing over the event sequence of a subtree \( T_n \) is completed only after the processing over the event sequences of all the subtrees rooted at the descendants of \( n \) have been completed. Specifically, each element in \( \text{stateStack} \) is a triple \((pr, pa, cs)\), where \( pr \) is a procedure call node, \( pa \) is either a path node or an unfinished path node, and \( cs \) is either an unmonitored call site node or an indirect call site node. For ease of our presentation, we refer to the top element of \( \text{stateStack} \) as \((\text{pr}_{\text{top}}, \text{pa}_{\text{top}} \text{ and } \text{cs}_{\text{top}})\) (lines 3-5). When a node in such a triple is not available, we simply use \( \bot \) to denote it.

The algorithm iteratively processes each event stream (lines 6-50), which progressively extends the HPPDAG instance under construction, and merges all equivalent HPPTree subtrees encountered.

An \textit{enter\_proc} event marks the entrance of a procedure call (line 9), which also implicitly marks the start of a BL path instance. Thus, the algorithm creates a new procedure call node \( \text{pr}_{\text{new}} \) (line 10) and a new path node \( \text{pa}_{\text{new}} \) (line 11), then pushes them into \( \text{stateStack} \) for later processing (line 12). Because this implicit BL path instance may eventually become an unfinished path, which cannot be determined when processing this \textit{enter\_proc} event, the algorithm leaves the assignment of a node type to \( \text{pa}_{\text{new}} \) to a later time (line 25 or line 44).

An \textit{enter\_unmonitored\_cs\_i} event marks the beginning of an unmonitored procedure call from which a monitored procedure is recursively invoked. A new unmonitored call site node \( \text{cs}_{\text{new}} \) is created with \( i \) as its label and assigned to \( \text{cs}_{\text{top}} \) (lines 32–33). Similarly, to handle an \textit{enter\_indirect\_cs} event, which marks the beginning of a procedure call issued from an indirect call site, a new indirect call site node \( \text{cs}_{\text{new}} \) is created and assigned to \( \text{cs}_{\text{top}} \) for later processing (lines 36–37).

The remaining five types of events, namely the \textit{exit\_proc} events (line 14), the \textit{bl\_path} events (line 24), the \textit{unfinished\_path} events (line 43), the \textit{exit\_unmonitored\_cs} events (line 39) and the \textit{exit\_indirect\_cs} events (line 39) each marks the end of a procedure call, the end of a BL path instance, the end of an unfinished path instance, the end of an unmonitored procedure call, and the end of a potentially unmonitored procedure call, respectively. Thus, the corresponding working copy of subgraph, say \( G_m \), where its root node is currently kept in the top element of \( \text{stateStack} \) (pushed by \textit{enter\_proc}, \textit{enter\_unmonitored\_cs}, or \textit{enter\_indirect\_cs} previously), is ready to be processed. So, the procedure \textit{addChild(parent, child)} is called to check for the uniqueness of \( G_m \), which determines whether there already exists a subgraph \( G_{m'} \) such that \( G_{m'} \equiv G_m \). If this is the case, it assigns \( G_{m'} \) to the children list of the parent of \( m \). Otherwise, it assigns \( G_m \) instead. After we have described the remaining few events below, we will describe the procedure \textit{addChild}.

On processing an \textit{exit\_proc} event, the node \( \text{pr}_{\text{top}} \) in \( \text{stateStack} \) is retrieved and assigned to \( pr \) (line 15). Then, \( \text{stateStack} \) pops out its top element (line 16). If \( \text{stateStack} \) becomes empty (line 17), \( pr \) must correspond to the node representing the entry procedure call of the HPPDAG instance under construction, and thus \( pr \) is appended as a child of the root node of the HPPDAG instance (line 18). Otherwise, the algorithm decides using either \( \text{cs}_{\text{top}} \) or \( \text{pa}_{\text{top}} \) as the parent node of \( pr \) based on whether \( \text{cs}_{\text{top}} \) of \( \text{stateStack} \) is available (line 19). After that, the procedure \textit{addChild} is called to update the children list of \( pr \)’s corresponding parent node (line 20 or line 22).

For a \textit{bl\_path} event, the algorithm assigns the node type \( \alpha \) and the corresponding path identifier \( id_{p} \) to \( \text{pa}_{\text{top}} \) (lines 25–26), and calls \textit{addChild} to update the children list of \( \text{pr}_{\text{top}} \) (line 27). Then, \( \text{pa}_{\text{top}} \) is reset to a new path node \( \text{pa}_{\text{new}} \) (lines 28–29). For an
ALGORITHM 1. HPPDAG Construction

Input: Traces: a set of HPPTree event streams
Output: an HPPDAG instance with root as its root node

1. sigToGraph // a hashtable maps a signature to the root node of a HPPDAG subgraph
2. stateStack ← Ø // a stack to keep DAG construction states
3. ptop ← stateStack.top().pr
4. ptop ← stateStack.top().pa
5. ctop ← stateStack.top().cs

for each trace file tr in Traces do
  for each traceEvent e in tr do
    switch (e)
    case enter_proc:id:
      create prnew, type(prnew) ← β, label(prnew) ← id/
      create panew
      stateStack.push((prnew, panew, ⊥))
      break
    case exit_proc:
      pr ← ptop;
      stateStack.pop()
      if (stateStack.empty()) then
        call addChild(pr, root)
      else if (ctop ≠ ⊥) then
        call addChild(ctop, pr)
      else
        call addChild(panew, pr)
      break
    case bl_path:idp:
      type(panew) ← α
      label(panew) ← idp
      call addChild(ppanew, ptop)
      ptop ← panew
      break
    case enter_unmonitored_cs:i:
      create cnew, type(cnew) ← δ, label(cnew) ← i
      cnew ← ctop
      break
    case enter_indirect_cs:
      create cnew, type(cnew) ← ε
      cnew ← ctop
      break
    case exit_unmonitored_cs, exit_indirect_cs:
      call addChild(panew, cnew)
      cnew ← ⊥;
      break
    case unfinished_path:idw,psw:
      type(panew) ← χ
      label(panew) ← {idw, psw}
      call addChild(ppanew, ptop)
    goto line 14
  end switch
end for

Procedure addChild(panew, child)
  m ← sigToGraph(sig(child))
  if m does not exist then
    sigToGraph(sig(child)) ← child
    Children(panew).append(child)
  else
    Children(panew).append(m)
end Procedure
For an unfinished path event, the algorithm firstly assigns the node type \( \gamma \) and the bookkeeping state \((id_n, ps_n)\) to \( \mathbf{p}_\text{top} \) (lines 44–45), and calls \( \text{addChild} \) to update the children list of \( \mathbf{p}_\text{top} \) (line 46). Then, it goes through the handling steps of an exit\(_\text{proc}\) event (line 47), that is, executing lines 15–23, because an unfinished\_path event implicitly marks the end of the procedure call of its representing unfinished path instance.

The procedure \( \text{addChild}(\text{parent}, \text{child}) \) accepts two nodes as input parameters, where they are the root nodes of the two working copies of subgraph \( G_{\text{parent}} \) and \( G_{\text{child}} \). We recall that one working copy of subgraph is constructed for each subtree, whose sequence of events has not been finished processing by the algorithm. Suppose \( G_{\text{parent}} \) and \( G_{\text{child}} \) represent the subtrees \( T_{\text{parent}} \) and \( T_{\text{child}} \), respectively. The preconditions for the invocation of \( \text{addChild} \) is that \( T_{\text{child}} \) is a subtree of \( T_{\text{parent}} \) and the sequence of trace events of \( T_{\text{child}} \) has all been processed.

The algorithm firstly computes the signature \( \text{sig}(\text{child}) \) of \( G_{\text{child}} \), and checks whether \( \text{sig}(\text{child}) \) has been kept in the map \( \text{sigToGraph} \) to determine whether any subgraph, say \( G_m \), such that \( G_m \approx G_{\text{child}} \) has been encountered before (lines 52–53). There are two cases to consider: (1) If there is no such signature in the map \( \text{sigToGraph} \), it indicates that no such subgraph \( G_m \) has been encountered. Thus, a new mapping \( \text{sig}(\text{child}) \rightarrow \text{child} \) is added to \( \text{sigToGraph} \) (line 54). Note that this also means that the subtree \( T_{\text{child}} \) has not been encountered before. Then \( G_{\text{child}} \) is used to represent the equivalence class \( [T_{\text{child}}] \) and \( \text{child} \) is appended to \( \text{Children}(\text{parent}) \) so that \( G_{\text{child}} \) becomes a child of node \( \text{parent} \) (line 55). (2) If the mapping \( \text{sig}(\text{child}) \rightarrow m \) exists in the map \( \text{sigToGraph} \), the algorithm appends \( m \) to \( \text{Children}(\text{parent}) \) (line 57), and \( \text{child} \) is discarded.

After the transformation, each equivalence class of subtrees in the given set of HPPTree instances is represented by the same subgraph in the corresponding HPPDAG instance (e.g., the equivalence classes of subtrees rooted at \( f_b \) and \( pa_2 \) in Figure 7(a) are represented by subgraphs \( G_{f_b} \) and \( G_{pa_2} \) in Figure 7(b), respectively).

To ease the understanding of Algorithm 1, in Figure 8 we illustrate how equivalent subtrees are identified and merged by handling HPPTree event streams. We reuse the left HPPTree instance shown in Figure 7(a) and show its detailed event stream in the top part of Figure 8 (for brevity, we skip some trace events irrelevant with our illustration). Figure 8(a) shows the algorithm’s state right before the processing of the two bolded events, i.e., \( \text{exit\_proc} \) and \( \text{bl\_path}\_pa_0 \) in the trace \( es_1 \). Figure 8(b) and 8(c) show how Algorithm 1 processes the two bolded events, respectively.

Figure 8(a) illustrates one property held by Algorithm 1, that is, an HPPDAG node finishes its processing only after all of its children nodes have finished their processing. Moreover, before an HPPDAG node finishes its processing (i.e., checked on equivalence at lines 52–57 in Algorithm 1), a working copy is maintained for it. For example, in Figure 8, there are three working copies of nodes illustrated as 1 hollow square and 2 hollow circles. We denote them as \( \mathbf{w}_{f_0} \), \( \mathbf{w}_f \) and \( \mathbf{w}_{pa} \) with respect to their labels, respectively. Each of these nodes still has its descendants nodes under processing. Note that the algorithm cannot determine the path identifier of \( \mathbf{w}_f \) until the corresponding path event, i.e., \( \text{bl\_path}\_pa_0 \) in \( es_1 \), is encountered (line 26 in Algorithm 1) after all its children nodes have finished processing. Thus, \( \mathbf{w}_f \) is just labelled with a question mark in Figure 8(a) and Figure 8(b).

Figure 8(b) shows how Algorithm 1 processes the \( \text{exit\_proc} \) event. The \( \text{exit\_proc} \) event pairs up with the last preceding \( \text{enter\_proc}\_f_b \) event in the event stream. It

\section*{Hierarchical Program Paths}

39:21

exit\_unmonitored\_cs event or an exit\_indirect\_cs, the algorithm simply calls \( \text{addChild} \) to update the children list of \( \mathbf{p}_\text{top} \) (line 40) and resets \( c_{\text{Stop}} \) to \( \perp \) (line 41).

For an unfinished\_path event, the algorithm firstly assigns the node type \( \gamma \) and the bookkeeping state \((id_n, ps_n)\) to \( \mathbf{p}_\text{top} \) (lines 44–45), and calls \( \text{addChild} \) to update the children list of \( \mathbf{p}_\text{top} \) (line 46). Then, it goes through the handling steps of an exit\_proc event (line 47), that is, executing lines 15–23, because an unfinished\_path event implicitly marks the end of the procedure call of its representing unfinished path instance.

The procedure \( \text{addChild}(\text{parent}, \text{child}) \) accepts two nodes as input parameters, where they are the root nodes of the two working copies of subgraph \( G_{\text{parent}} \) and \( G_{\text{child}} \). We recall that one working copy of subgraph is constructed for each subtree, whose sequence of events has not been finished processing by the algorithm. Suppose \( G_{\text{parent}} \) and \( G_{\text{child}} \) represent the subtrees \( T_{\text{parent}} \) and \( T_{\text{child}} \), respectively. The preconditions for the invocation of \( \text{addChild} \) is that \( T_{\text{child}} \) is a subtree of \( T_{\text{parent}} \) and the sequence of trace events of \( T_{\text{child}} \) has all been processed.

The algorithm firstly computes the signature \( \text{sig}(\text{child}) \) of \( G_{\text{child}} \), and checks whether \( \text{sig}(\text{child}) \) has been kept in the map \( \text{sigToGraph} \) to determine whether any subgraph, say \( G_m \), such that \( G_m \approx G_{\text{child}} \) has been encountered before (lines 52–53). There are two cases to consider: (1) If there is no such signature in the map \( \text{sigToGraph} \), it indicates that no such subgraph \( G_m \) has been encountered. Thus, a new mapping \( \text{sig}(\text{child}) \rightarrow \text{child} \) is added to \( \text{sigToGraph} \) (line 54). Note that this also means that the subtree \( T_{\text{child}} \) has not been encountered before. Then \( G_{\text{child}} \) is used to represent the equivalence class \( [T_{\text{child}}] \) and \( \text{child} \) is appended to \( \text{Children}(\text{parent}) \) so that \( G_{\text{child}} \) becomes a child of node \( \text{parent} \) (line 55). (2) If the mapping \( \text{sig}(\text{child}) \rightarrow m \) exists in the map \( \text{sigToGraph} \), the algorithm appends \( m \) to \( \text{Children}(\text{parent}) \) (line 57), and \( \text{child} \) is discarded.

After the transformation, each equivalence class of subtrees in the given set of HPPTree instances is represented by the same subgraph in the corresponding HPPDAG instance (e.g., the equivalence classes of subtrees rooted at \( f_b \) and \( pa_2 \) in Figure 7(a) are represented by subgraphs \( G_{f_b} \) and \( G_{pa_2} \) in Figure 7(b), respectively).

To ease the understanding of Algorithm 1, in Figure 8 we illustrate how equivalent subtrees are identified and merged by handling HPPTree event streams. We reuse the left HPPTree instance shown in Figure 7(a) and show its detailed event stream in the top part of Figure 8 (for brevity, we skip some trace events irrelevant with our illustration). Figure 8(a) shows the algorithm’s state right before the processing of the two bolded events, i.e., \( \text{exit\_proc} \) and \( \text{bl\_path}\_pa_0 \) in the trace \( es_1 \). Figure 8(b) and 8(c) show how Algorithm 1 processes the two bolded events, respectively.

Figure 8(a) illustrates one property held by Algorithm 1, that is, an HPPDAG node finishes its processing only after all of its children nodes have finished their processing. Moreover, before an HPPDAG node finishes its processing (i.e., checked on equivalence at lines 52–57 in Algorithm 1), a working copy is maintained for it. For example, in Figure 8, there are three working copies of nodes illustrated as 1 hollow square and 2 hollow circles. We denote them as \( \mathbf{w}_{f_0} \), \( \mathbf{w}_f \) and \( \mathbf{w}_{pa} \) with respect to their labels, respectively. Each of these nodes still has its descendants nodes under processing. Note that the algorithm cannot determine the path identifier of \( \mathbf{w}_f \) until the corresponding path event, i.e., \( \text{bl\_path}\_pa_0 \) in \( es_1 \), is encountered (line 26 in Algorithm 1) after all its children nodes have finished processing. Thus, \( \mathbf{w}_f \) is just labelled with a question mark in Figure 8(a) and Figure 8(b).

Figure 8(b) shows how Algorithm 1 processes the \( \text{exit\_proc} \) event. The \( \text{exit\_proc} \) event pairs up with the last preceding \( \text{enter\_proc}\_f_b \) event in the event stream. It
indicates the end of the procedure call on \( f_b \) so that \( w_{fb} \), is ready to be processed. The signature \( \text{sig}(w_{fb}) \) is generated and checked against the hashtable \( \text{sigToGraph} \), which has stored the signatures of all subgraphs that have completed their processing (line 52 in Algorithm 1). Then, an existing subgraph rooted at the node outlined in dotted line in Figure 8(b), say \( n' \), is found such that \( \text{sig}(n') = \text{sig}(w_{fb}) \) indicating \( G_{n'} \cong G_{w_{fb}} \). Thus, \( w_{fa} \), which is the parent node of \( w_{fb} \), replaces \( w_{fb} \) by \( n' \) as its child and \( w_{fb} \) is discarded (line 57 in Algorithm 1).

Figure 8(c) explains how the event \( \text{bl_path}:p_{a3} \) in the trace \( es_1 \) is handled. As we stated in Section 2.1, a BL path event is generated at the end of an executed BL path instance. Thus the working copy \( w_{pa3} \) in Figure 8(b) reveals its path identifier as \( p_{a3} \) (line 26 in Algorithm 1) and we use \( w_{pa3} \) to denote it in Figure 8(c). Then the signature \( \text{sig}(w_{pa3}) \) is checked for subgraph equivalence (line 52 in Algorithm 1). Since there is no any previously constructed subgraph equivalent with the subgraph rooted at \( w_{pa3} \), the mapping from \( \text{sig}(w_{pa3}) \) to \( w_{pa3} \) is saved to the hashtable \( \text{sigToGraph} \) (lines 53-54 in Algorithm 1). Moreover, \( w_{pa3} \) is added into the HPPDAG instance as an HPPDAG node.

After the construction of an HPPDAG instance, path queries can be directly performed on this instance because the hierarchical structure used to model the control flow in HPPTree is maintained, even though this HPPDAG instance is more condensed.

### 3.3 Retrieval of Inter-procedural (sub)Paths

From the subgraph representation relation \( (\cong) \) presented in Section 3.2, we can see that the hierarchical structure of an HPPTree instance, which is used to model the control flow of execution traces, is maintained in its corresponding HPPDAG instance. As such, path queries can be directly performed on the HPPDAG instance. For example, the retrieval of an inter-procedural (sub)path can be trivially performed as if it were performed on the corresponding set of HPPTree instances.
To be more specific, for an HPPDAG node \( n \), we denote the inter-procedural path represented by the subgraph rooted at \( n \) as \( \text{interPath}(n) \). For any node \( m \), if \( m \) is a procedure call node, an unmonitored call site node or an indirect call site node, \( \text{interPath}(m) \) is the concatenation of \( \text{interPath}(\text{child}_i) \) where \( \text{child}_i = \text{Children}(m)[i] \) for \( i \) starting from 1 to \( |\text{Children}(m)| \), i.e., \( \text{interPath}(\text{child}_1)\text{^-}\text{interPath}(\text{child}_2)\text{^-}...\text{^-interPath}(\text{child}_{|\text{Children}(m)|}) \). On the other hand, if \( m \) is a path node or an unfinished path node, we need to firstly retrieve the intra-procedural path of \( m \), denoted as \( \text{intraPath}(m) \), using \( \text{label}(m) \). We note that the intra-procedural path of a path node \( m \) can be directly acquired from its recorded BL path identifier (i.e., \( \text{label}(m) \)). Similarly, the intra-procedural path of an unfinished path node can be acquired from its recorded bookkeeping state (i.e., \( \text{label}(m) \)) based on Theorem 2 presented in Section 3.1.2. We also recall that the call site alignment property guarantees that for a path node \( m \), each of its children nodes, if any, aligns with a call site occurrence in the BL path instance represented by \( m \), which is \( \text{intraPath}(m) \). Then \( \text{interPath}(m) \) is obtained by inserting \( \text{interPath}(\text{child}) \), where \( \text{child} \) is a child node of \( m \), right after the aligned call site \( \text{callsite}(\text{child}) \) of \( \text{child} \) in \( \text{intraPath}(m) \). The inter-procedural path \( \text{interPath}(n) \) for an HPPDAG node \( n \) relies on the inter-procedural paths of \( n \)'s children, if any. Thus, it is straightforward that this recursive construction can be completed by graph traversals on the corresponding HPPDAG instance.

Take the HPPTree instance in Figure 5(c) as an example. No two subtrees in this HPPTree instance are equivalent to one another. Thus, it is also a trivial HPPDAG instance. We denote the node labeled with \( \text{main} \) as \( n \), the node labeled with \( \text{pmain} \) as \( m \) and the children of \( m \) as \( \text{child}_1 \), \( \text{child}_2 \) and \( \text{child}_3 \) from left to right, respectively. The inter-procedural path of \( n \), i.e., \( \text{intraPath}(n) \), is then equal to the inter-procedural path of \( m \), i.e., \( \text{interPath}(m) \). Because \( m \) is a path node with path identifier \( \text{pmain} \), we retrieve its intra-procedural path, i.e., \( \text{intraPath}(m) = (1, 2, 3, 4, 5) \). With regards to the call site alignment, we can get that \( \text{callsite}(\text{child}_1) = 2 \), \( \text{callsite}(\text{child}_2) = 3 \) and \( \text{callsite}(\text{child}_3) = 4 \). As such, \( \text{interPath}(m) = (1, 2, \text{interPath}(\text{child}_1), 3, \text{interPath}(\text{child}_2), 4, \text{interPath}(\text{child}_3), 5) \). Finally, after we recursively resolve the inter-procedural paths of \( \text{child}_1, \text{child}_2 \) and \( \text{child}_3 \), we obtain \( \text{interPath}(n) = (1, 2, P_x, 3, 4, P'_x, 5) \). Note that for simplicity, we directly use the path identifiers \( P_x \) and \( P'_x \) to represent the intra-procedural paths of the two leaf path nodes. Moreover, \( \text{interPath}(\text{child}_3) \) is empty because it is an indirect call site node without any child.

4. EVALUATION

4.1 Prototype Implementation of HPP

We have implemented a prototype of HPP on LLVM 3.5 [Lattner and Adve 2004] in 5000+ lines of source code. Each program to be traced was compiled into LLVM bitcode on which instrumentation was performed. The runtime system used for tracing in our prototype was compiled as a static library and to be linked with each instrumented program. The current version of the prototype supports multithreaded C and C++ programs using the Pthreads model. However, programs with unstructured control flow (e.g., \text{setjmp} and \text{longjmp} functions in C) and process fork are not supported yet.

In Section 3, for simplicity of presentation, we assume the program under tracing in Phase 1 (trace collection) of HPP is single-threaded. However, HPP can be trivially extended, as we have done in the prototype of HPP, to handle multi-threaded programs by tracing each thread in a program execution independently. During the tracing of an execution, each thread generates the event stream of an HPPTree instance.
representing the complete dynamic control flow of that thread. Note that HPP does not capture the thread-interleaving and the event stream of each thread is generated independently in a thread-local way (i.e., no synchronization with other threads) during execution. Specifically, in Phase 1, each thread in a program execution is associated with a shadow state, which keeps a trace log handle, for dumping the event stream generated by the thread. The HPPTree instances of all executed threads are simply considered as subtrees of one overall HPPTree instance with a notional root node. Then in Phase 2 (HPPDAG construction), for each program execution, Algorithm 1 simply processes the event streams of all these threads one by one. For each benchmark, we used the above method to generate one final HPPDAG instance for all executions that had been traced. Moreover, the HPPTree instance of each thread of each execution was represented by a subgraph in this final HPPDAG instance.

For each trace event collected in Phase 1, we used one byte to encode the opcode of the event. The operands path identifier, procedure identifier, watch point identifier, and index i for unmonitored call site nodes each took four bytes (a 32-bit unsigned integer) to encode. We used eight bytes (a 64-bit signed integer) to keep each operand of path state.

In our experiment, for the transformation of HPPTree into HPPDAG, the prototype read the event stream of each HPPTree instance from the secondary storage, and then constructed an HPPDAG instance followed by keeping the constructed HPPDAG instance into the secondary storage. In our prototype, we serialized each HPPDAG instance into its secondary format using a straightforward approach, which is presented as follows. For each node, say n, in the HPPDAG instance, n was assigned with an identifier idn, which was a 32-bit integer. Then, we wrote (idn, type(n), label(n), ChildrenID(n)); where ChildrenID(n) was the list of identifiers of nodes in Children(n), to the HPPDAG trace file.

The evaluation on the size of BLPT traces, HPPTree traces and HPPDAG instances was all performed on their secondary storage formats. To perform queries on an HPPDAG instance, the HPPDAG instance needed firstly to be restored from its secondary storage format. The loading and restoring times were also reported in evaluations (Table IX).

4.2 Experimental Setup

Platform. All experiments were conducted on Ubuntu 12.04 x86_64 with four 2.9GHz Xeon Cores and 16 GB memory. All subjects were compiled into LLVM bitcode using CLANG 3.5 [CLANG]. Instrumentation was performed with LLVM 3.5.

Benchmarks. We performed our experiment on the nine application subjects in the SPLASH-2 benchmark suite [Woo et al. 1995] and 11 benchmarks in the SPECint 2006 benchmark suite [Henning 2006]. Note that 471.omnetpp in SPECint 2006 was excluded in our experiment due to its use of unstructured control flow (setjmp and longjmp). For the ease of our presentation, we used ocean_cp, ocean_ncp, water-nsp and water-sp to refer to the two versions of benchmarks ocean and water, respectively, in the SPLASH-2 benchmark suite.

The descriptive statistics of the benchmarks are shown in Table I. SPLASH-2 has been widely used to evaluate different aspects (including performances and functionalities) of a wide-range of techniques, and SPECint 2006 is an industry standard benchmark suite to evaluate the system performance on handling compute-intensive integer workload. The application area of each benchmark is shown in the second column of Table I. All the benchmarks in SPLASH2 were multithreaded, and we set the number of threads in each execution run to four so that each thread can be
handled by a single core of our platform. All the benchmarks in SPECint 2006 were single-threaded.

The third column shows the lines of LLVM bitcode of each benchmark. The fourth and fifth columns show the number of monitored and unmonitored procedures in each benchmark, respectively. In our experiment, we specified all the procedures with source code available as monitored procedures; and for library procedures, which were provided in binary formats only, we specified them as unmonitored procedures. The last column shows the total number of threads executed in our experiment for each benchmark. As we have presented in Section 4.1, each thread in either multi-threaded executions of SPLASH2 or single-threaded executions of SPECint 2006 generates a single event stream of its corresponding HPPTree instance.

The test cases executed on each benchmark were selected from the existing test harness of SPLASH2 and SPECint 2006. We selected the test cases of each benchmark to make each native execution run of that benchmark be able to finish close to 10 seconds on our platform. We made this selection because the trace files generated by BLPT and Phase 1 of HPP were already very large (around 373 GB as shown in Table VI) and yet our platform only has a limited secondary storage.

Because both the SPLASH-2 and SPECint 2006 benchmark suites were designed for performance evaluation, no crash had been observed in our experiment. However, some of these benchmark programs could be terminated by invoking the system call exit() (instead of returning from the main() function of each benchmark). Thus, the experiment treated each call to exit() as an early termination (since it can be called from arbitrary program points and possibly in unmonitored library procedures), and we used unfinished_path events to handle all such cases.

Methodology. For comparison purpose, we have also implemented BLPT on the LLVM framework. Our extension followed Larus’s approach [1999], but without the online compression part (i.e., Sequitur [Nevill-Manning and Witten 1997]). Note that

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Application Area</th>
<th>lines of bitcode</th>
<th># of procedures</th>
<th># of executed threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>monitored</td>
<td>unmonitored</td>
</tr>
<tr>
<td>barnes</td>
<td>High performance computing</td>
<td>312</td>
<td>56</td>
<td>32</td>
</tr>
<tr>
<td>fmm</td>
<td>High performance computing</td>
<td>522</td>
<td>92</td>
<td>34</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>High performance computing</td>
<td>1147</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>ocean-ncep</td>
<td>High performance computing</td>
<td>733</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>radioactivity</td>
<td>Graphics</td>
<td>917</td>
<td>179</td>
<td>24</td>
</tr>
<tr>
<td>raytrace</td>
<td>Graphics</td>
<td>1133</td>
<td>140</td>
<td>32</td>
</tr>
<tr>
<td>weldend</td>
<td>Graphics</td>
<td>3902</td>
<td>385</td>
<td>52</td>
</tr>
<tr>
<td>water-nsw</td>
<td>High performance computing</td>
<td>319</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>water-sp</td>
<td>High performance computing</td>
<td>339</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>Programming Language</td>
<td>17058</td>
<td>1864</td>
<td>106</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>Compression</td>
<td>1363</td>
<td>100</td>
<td>15</td>
</tr>
<tr>
<td>402.gcc</td>
<td>C Compiler</td>
<td>164315</td>
<td>5577</td>
<td>62</td>
</tr>
<tr>
<td>445.mcf2</td>
<td>Combinatorial Optimization</td>
<td>184</td>
<td>24</td>
<td>12</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>Artificial Intelligence: Go</td>
<td>27726</td>
<td>2679</td>
<td>40</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>Search Gene Sequence</td>
<td>4485</td>
<td>538</td>
<td>57</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>Artificial Intelligence: chess</td>
<td>2818</td>
<td>144</td>
<td>27</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>Physics/Quantum Computing</td>
<td>877</td>
<td>115</td>
<td>22</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>Video Compression</td>
<td>9109</td>
<td>590</td>
<td>41</td>
</tr>
<tr>
<td>471.astar</td>
<td>Path-finding Algorithms</td>
<td>1134</td>
<td>159</td>
<td>19</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>XML Processing</td>
<td>40879</td>
<td>28785</td>
<td>126</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>279272</td>
<td>41492</td>
<td>799</td>
</tr>
</tbody>
</table>
compression techniques are orthogonal to both HPP and BLPT in the sense that the event streams of HPPTree instances can also be fed into compression techniques like Sequitur [Nevill-Manning and Witten 1997] as the event streams of BLPT. Meanwhile, HPPDAG instances, in their serialized format, can also be further compressed with compression techniques, e.g., the classic LZW or GZIP compression [Welch 1984; GZIP 2010], to be more space efficient. Both HPP and BLPT adopted the same encoding scheme of trace events to ensure a fair comparison. Each test execution was repeated five times, and we compared the mean results of these runs in the experiment because we actually could only observe marginal differences in the results among these five test executions produced by each technique.

### 4.3 Experimental Results on HPP Trace Events (Phase 1 of HPP)

To evaluate the effectiveness of HPPTree on modeling inter-procedural paths as presented in Section 3.1, for each benchmark, we compared the numbers of BL path events generated by HPP and BLPT on same execution runs. To further assess both the time and space efficiency resulting from the reduction of BL path events, we measured the size of overall trace events generated by HPP and BLPT as well as the runtime overhead of HPP and BLPT. Note that we compared the sizes of trace events in their secondary storage format due to their huge sizes. However, these trace events were not necessarily to be kept in secondary storage in practice.

Figure 9 shows the distribution of trace events generated by BLPT and HPP on the same execution. For each benchmark, there is a pair of bars. The left stacked bar shows the events distribution on BLPT and the right one shows the event distribution on HPP. Note that, for each benchmark, the number of events for each type of trace event is normalized against the total number of events on BLPT and the normalized value is shown as the height of the corresponding bar. Because an enter_proc event normally pairs up with an exit_proc event except on rare cases where procedure calls are terminated before returned, the numbers of these two events are close. For simplicity, we counted enter_proc events and exit_proc events as one value, denoted as proc in Figure 9. Similarly, enter_unmonitored_cs and exit_unmonitored_cs are counted.

![Fig. 9. Distribution of different types of trace events generated by BLPT and HPP. For each benchmark, the left stacked bar represents the distribution on BLPT and the right one represents that on HPP; the numbers of events of different types each is normalized against the total number of events on BLPT.](image-url)
together, denoted as unmonitored_cs. enter_indirect_cs and exit_indirect_cs events were counted together, denoted as indirect_cs.

Figure 9 visually shows that enter_proc, exit_proc and bl_path events take up the vast majority of all the trace events generated in HPP. On raytrace, 400.perlbench and 464.h264ref, there were also noticeable numbers of enter_indirect_cs and exit_indirect_cs events because these programs heavily used indirect procedure calls through function pointers in the course of their executions. enter_unmonitored_cs, exit_unmonitored_cs and unfinished_path events took a small portion of the total number of events and were barely seen in the figure. Because BLPT and HPP traced the same executions for each benchmark, their numbers of enter_proc and exit_proc events were the same. We also observed from the figure that, except on raytrace, 400.perlbench and 464.h264ref, BLPT generated more trace events than HPP on the same executions. All such reductions were contributed by generating smaller numbers of bl_path events due to the more compact modeling of BL call paths in HPP than BLPT. On raytrace, 400.perlbench and 464.h264ref, HPP generated fewer bl_path events, but the total number of events generated by HPP outnumbered the total number of events generated by BLPT due to the additional needs of generating enter_indirect_cs and exit_indirect_cs events. However, neither enter_indirect_cs nor exit_indirect_cs has any operand (see Figure 6) so that they consume fewer bytes for encoding than a bl_path event (one byte versus five bytes). The resultant trace size of BLPT was still larger than that of HPP as shown in Table III.

In Table II, we show the numbers of HPP trace events in correspondence to Figure 9. From the table, we observed that only executions of volrend, 403.gcc and 464.h264ref had encountered procedure calls invoked from unmonitored procedures. Indirect procedure calls through function pointers were found to be more common and had taken place in the executions of 13 out of 20 benchmarks. Eleven (11) out of 20 benchmarks terminated their executions by calling the C library function exit(), which resulted in unfinished_path events. These auxiliary events took up only a small portion of all the trace events in the experiment. Nonetheless, these events frequently appear

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>proc</th>
<th>bl_path</th>
<th>unmonitored_cs</th>
<th>indirect_cs</th>
<th>unfinished_path</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>789,128,181</td>
<td>607,586,819</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>fmm</td>
<td>77,682,117</td>
<td>2,147,483,647</td>
<td>0</td>
<td>14,076,838</td>
<td>1</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>6,399</td>
<td>848,745,069</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ocean-ncp</td>
<td>3,441</td>
<td>854,591,867</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>radiosity</td>
<td>1,353,525,255</td>
<td>1,688,324,024</td>
<td>0</td>
<td>516,208</td>
<td>1</td>
</tr>
<tr>
<td>raytrace</td>
<td>1,643,257,463</td>
<td>1,642,199,425</td>
<td>0</td>
<td>1,608,584,416</td>
<td>1</td>
</tr>
<tr>
<td>volrend</td>
<td>22,317,681</td>
<td>172,105,903</td>
<td>1,800</td>
<td>384,600</td>
<td>1</td>
</tr>
<tr>
<td>water-nx</td>
<td>171,327,835</td>
<td>1,459,524,891</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>water-sp</td>
<td>227,900,269</td>
<td>1,982,094,791</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>507,490,050</td>
<td>476,379,060</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>401.hzip2</td>
<td>88,940,136</td>
<td>1,514,126,981</td>
<td>0</td>
<td>243,944,520</td>
<td>16</td>
</tr>
<tr>
<td>403.gcc</td>
<td>329,279,362</td>
<td>430,768,058</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>429.mcf</td>
<td>709,864,314</td>
<td>1,086,799,703</td>
<td>0</td>
<td>144</td>
<td>0</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>445,750,760</td>
<td>1,007,026,966</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>95,142,394</td>
<td>552,968,946</td>
<td>14,574</td>
<td>20,628,442</td>
<td>0</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>347,092,960</td>
<td>782,783,339</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>29,085,000</td>
<td>1,292,925,765</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>2,145,627,184</td>
<td>2,147,483,647</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>473.atlas</td>
<td>2,672,129,444</td>
<td>2,097,293,273</td>
<td>0</td>
<td>526,812</td>
<td>0</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>250,634,842</td>
<td>137,772,315</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
in program executions of many programs, and handling them is crucial to maintain the soundness of HPP as presented in Section 3.

Table III shows the results on the number of *bl_path* events and the total size of traces kept in the secondary storage. (We note that we were unable to measure their in-memory size due to the limited physical memory of our platform.) The second and third columns are the total numbers of BL path events in the event streams produced by BLPT and HPP, followed by the difference between them in ratio. The next two columns show the total size of the trace logs produced by each technique. The seventh column presents the fraction of the size of trace logs produced by HPP in relation to BLPT.

In Table III, on each benchmark, HPP improves BLPT in terms of both the number of BL path events generated and trace size. On average, BLPT generated 42.1% more BL path events than HPP, and produced more bulky trace logs by 28.2%.

The above relative results are consistent with the difference between the approaches to handle BL call paths taken by HPP and BLPT. We recall from Section 2.2 that each BL call path is encoded as one BL path in HPP; whereas, the same BL call path is encoded as multiple BL paths in BLPT. As such, HPP is able to use fewer BL path events to model the same control flow path segments.

The time costs for online tracing of both HPP and BLPT are shown in Table IV. The execution time (i.e., the time cost) of each technique on each benchmark is the sum of the CPU time of each concurrently executed thread of the benchmark in single execution. The second column shows the execution time of the native run of each subject. Columns 3 and 4 show the time costs of BLTP and HPP, respectively. The next two columns present the slowdown factors incurred by BLTP and HPP in relation to the native run, respectively.

The slowdown factor is calculated by the following formula: (technique execution time ÷ native execution time) – 1. For instance, on barnes, the slowdown factor incurred by HPP is computed as (106.9 ÷ 6.9) – 1, which yields 14.52. The rightmost column shows the slowdown factor incurred by HPP relative to the slowdown factor incurred by HPP.
Hierarchical Program Paths

39:29

Table IV. Comparison on runtime overhead between HPP (Phase 1) and BLPT

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Execution Time (in second)</th>
<th>Slowdown Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Native</td>
<td>BLPT</td>
</tr>
<tr>
<td>barnes</td>
<td>6.9</td>
<td>146.4</td>
</tr>
<tr>
<td>ffm</td>
<td>16.6</td>
<td>293.4</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>8.9</td>
<td>99.6</td>
</tr>
<tr>
<td>ocean-ncp</td>
<td>12.6</td>
<td>110.6</td>
</tr>
<tr>
<td>radixsort</td>
<td>14.6</td>
<td>280.9</td>
</tr>
<tr>
<td>raytrace</td>
<td>12.8</td>
<td>311.6</td>
</tr>
<tr>
<td>volrend</td>
<td>4.7</td>
<td>19.6</td>
</tr>
<tr>
<td>water-na</td>
<td>4.5</td>
<td>236.5</td>
</tr>
<tr>
<td>water-sp</td>
<td>8.8</td>
<td>309.5</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>5.7</td>
<td>62.2</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>10.0</td>
<td>94.4</td>
</tr>
<tr>
<td>403.gcc</td>
<td>3.2</td>
<td>51.2</td>
</tr>
<tr>
<td>429.mcf</td>
<td>22.1</td>
<td>171.1</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>8.3</td>
<td>92.6</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>7.2</td>
<td>46.2</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>6.0</td>
<td>78.2</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>3.5</td>
<td>73.4</td>
</tr>
<tr>
<td>464.hmmer</td>
<td>34.1</td>
<td>435.3</td>
</tr>
<tr>
<td>473.astar</td>
<td>15.3</td>
<td>313.4</td>
</tr>
<tr>
<td>483.xalanbench</td>
<td>0.31</td>
<td>24.8</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

by BLTP. From the table, on average, HPP incurred 15.19x slowdown, and BLPT incurred 19.14x slowdown. The mean difference is 21%.

We observe that the experimental results on trace size and slowdown are consistent with the trace events data shown in Figure 9, which demonstrates the benefits of HPP on representing BL call paths.

On fm, volrend, 401.bzip2 and 462.libquantum, HPP only achieved marginal reduction of BL path events. We found that their major computations were performed in loops that invoke no procedure call on monitored or unmonitored procedures in each iteration. The lack of BL call path instances in execution of these two benchmarks led to the comparable performances of tracing with HPP and BLPT. HPP significantly outperformed BLPT on all other benchmarks, which seems indicating that HPP has a good potential to outperform BLPT on whole program path tracing as far as the online tracing overhead is concerned.

4.4 Experimental Results on HPPDAG Generation (Phase 2 of HPP)

In this section, we present the experimental results on Phase 2 of HPP. We recall from Section 3 that no concrete HPPTree instance has to be built by HPP. Thus, we use the event streams, each of which is the events in the post-order traversal of a corresponding HPPTree instance, generated by Phase 1 of HPP to measure the size of these HPPTree instances. We firstly quantitatively assessed the effect of merging of equivalent subtrees in Phase 2 by comparing the number of subtrees in HPPTree instances and the number of subgraphs in the corresponding HPPDAG instances. Then, we compared the sizes of BLPT traces, HPPTree event streams, and HPPDAG instances in the secondary storage format. We then evaluated that to what extent the HPP framework is able to work together with a compression technique to further reduce the size of the HPPDAG traces. We measured the sizes of BLPT traces, HPPTree event streams, and serialized HPPDAG instances after applying the
compression technique. We also evaluated the compression/decompression overhead on BLPT traces and serialized HPPDAG traces.

As we have presented in Section 3.2, each equivalence class of subtrees is represented by a subgraph in the corresponding HPPDAG instance. Moreover, each HPPTree node \( n \) represents a subtree rooted at \( n \), and each HPPDAG node \( m \) represents a subgraph rooted at \( m \). Therefore, we measured the numbers of nodes in HPPTree instances and the corresponding HPPDAG instances. Table V shows the number of HPPTree nodes and the number of HPPDAG nodes for the executions of each benchmark in the second and third column, respectively. The fourth column shows the mean number of subtrees represented by one subgraph. We divided the total number of HPPTree nodes by the total number of HPPDAG nodes to obtain this value for each benchmark. From the table, we found that HPPTree instances entail a large number of repetitive occurrences of identical inter-procedural paths. Consider all benchmarks as a whole, on average, each subgraph represents 341 equivalent subtrees. This ratio is significant. This also indicates that although HPPTree instances enable path queries to be performed directly, without properly handling the repetition of path segments, HPPTree instances can be largely redundant.

Table VI further shows the trace log size of each benchmark both before and after applying Phase 2 of HPP. The result shows that on average, the trace log sizes of BLPT traces and HPPTree were 72.19 times and 54.83 times, respectively, larger than that of HPPDAG. The differences were also significant.

The total time cost (including the I/O time) of Phase 2 is presented in the rightmost column entitled “HPPDAG Construction Time”. As shown in the table, the HPPDAG construction time in Phase 2 is much larger than the program tracing time in Phase 1 (shown in Table IV). It is mainly due to the subgraph equivalence checks in Algorithm 1. To avoid largely slowing down the online tracing in Phase 1 of HPP, Phase 2 was performed as an offline phase.

HPPDAG is not designed to be a replacement of existing trace compression techniques, even though the above results have shown that HPPDAG instances can be
Hierarchical Program Paths

Table VI. Comparison on sizes of BLPT traces, HPPTree and HPPDAG, and HPPDAG construction time (Phase 2)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trace Size (in GB)</th>
<th>HPPDAG Construction Time (in second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLPT (A)</td>
<td>HPPTree (B)</td>
</tr>
<tr>
<td>barnes</td>
<td>7.766</td>
<td>5.034</td>
</tr>
<tr>
<td>fmm</td>
<td>16.397</td>
<td>15.737</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>5.514</td>
<td>3.953</td>
</tr>
<tr>
<td>ocean-ncp</td>
<td>5.521</td>
<td>3.980</td>
</tr>
<tr>
<td>raytrace</td>
<td>15.056</td>
<td>11.645</td>
</tr>
<tr>
<td>volrend</td>
<td>16.087</td>
<td>13.737</td>
</tr>
<tr>
<td>water-ns</td>
<td>0.944</td>
<td>0.864</td>
</tr>
<tr>
<td>water-sp</td>
<td>13.266</td>
<td>7.275</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>4.744</td>
<td>3.864</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>7.507</td>
<td>7.300</td>
</tr>
<tr>
<td>403.gcc</td>
<td>3.670</td>
<td>2.945</td>
</tr>
<tr>
<td>429.mcf</td>
<td>11.448</td>
<td>9.813</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>6.874</td>
<td>5.936</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>3.210</td>
<td>2.841</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>5.588</td>
<td>4.672</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>6.173</td>
<td>6.103</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>32.350</td>
<td>28.700</td>
</tr>
<tr>
<td>473.astar</td>
<td>23.456</td>
<td>17.569</td>
</tr>
<tr>
<td>481.xalancbmk</td>
<td>1.927</td>
<td>1.353</td>
</tr>
<tr>
<td>Total</td>
<td>205.61</td>
<td>163.19</td>
</tr>
<tr>
<td>Mean</td>
<td>10.28</td>
<td>8.16</td>
</tr>
</tbody>
</table>

significantly smaller in size than their BLPT counterparts. An HPPDAG instance maintains the hierarchical structures of the corresponding HPPTree instances so that it can be directly queried on without any decompression process. Nonetheless, the hierarchical structures also impose constraints on HPPDAG construction disallowing aggressive elimination of redundancy as done by compression techniques, which are unaware of the control flow semantics. This indicates that HPPDAG instances can be further compressed. Besides, HPPDAG instances are smaller in size compared with their BLPT counterparts, which leads to faster compression and decompression. Thus in the scenarios like storage or transmission of execution trace files where space efficiency is a major concern, HPP could be a good complement of compression techniques.

In Figure 10, we show the typical workflow, in which HPP is incorporated with some compression technique. After the HPPDAG construction, a serialized HPPDAG instance is firstly generated and then compressed by a compression technique to achieve better space efficiency. Upon trace queries, the compressed instance is firstly decompressed, and then loaded and restored into the in-memory HPPDAG format, on which trace queries can then be directly performed.

We explored the possibility of further size reduction by compressing each of the BLPT traces and the HPPDAG instances of all benchmarks using the gzip 1.4 tool [GZIP 2010]. The results on size after compression are shown in Table VII. In the table, the column entitled “BLPT” and “HPPDAG” are the sizes of the compressed trace logs for these BLPT and HPPDAG trace logs presented in Table VI, respectively. The rightmost column is the ratio of the compressed trace logs of BLPT and HPPDAG on each benchmark.
First, from Table VI, after compression, the total size of the HPPTree traces was 9.82 times that of the HPPDAG traces. At the same time, the size of HPPTree traces was reduced from 164.92 GB to 3013.85 MB, or 54.7 in compression ratio after compression by gzip. This shows that HPPDAG alone cannot outperform gzip on the reduction of trace size, which agrees with our assumption that maintaining the hierarchical structure in HPPDAG constrains its ability to aggressively reduce the size of traces.

Second, comparing Table VI and Table VII, both BLPT and HPPDAG traces can be significantly compressed. For BLPT, the total trace size of all benchmarks was reduced from 195.46 GB to 3236.01 MB, or 60.4 in compression ratio. For HPPDAG, the total trace size was reduced from 14.61 GB to 1212.38 MB, or 12.1 in compression ratio.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Trace Size after Compression (in MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLPT (A)</td>
</tr>
<tr>
<td>barnes</td>
<td>101.72</td>
</tr>
<tr>
<td>fmm</td>
<td>126.95</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>13.54</td>
</tr>
<tr>
<td>ocean-ncp</td>
<td>13.49</td>
</tr>
<tr>
<td>radiosity</td>
<td>229.79</td>
</tr>
<tr>
<td>raytrace</td>
<td>125.48</td>
</tr>
<tr>
<td>voldemd</td>
<td>42.67</td>
</tr>
<tr>
<td>water-ns</td>
<td>83.34</td>
</tr>
<tr>
<td>water-ap</td>
<td>147.38</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>232.98</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>132.49</td>
</tr>
<tr>
<td>401.gcc</td>
<td>120.56</td>
</tr>
<tr>
<td>429.mcfc</td>
<td>257.45</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>404.72</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>167.98</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>251.15</td>
</tr>
<tr>
<td>461.libquantum</td>
<td>20.19</td>
</tr>
<tr>
<td>464.mcf4ref</td>
<td>559.50</td>
</tr>
<tr>
<td>471.astar</td>
<td>479.67</td>
</tr>
<tr>
<td>484.xalancbmk</td>
<td>54.73</td>
</tr>
<tr>
<td>Total</td>
<td>3565.80</td>
</tr>
<tr>
<td>Mean</td>
<td>178.29</td>
</tr>
</tbody>
</table>
difference in the compression ratios suggests that the HPPDAG traces were significantly more compact than the BLPT traces observed in the experiment.

Finally, from Table VII, the compressed HPPDAG traces were found to be strictly smaller than the compressed BLPT traces on all benchmarks, except 456.hmmer on which HPPDAG trace was less than 1% larger. The difference in ratio ranged from almost no difference on voldrend and 456.hmmer to more than 331 on 400.perlbench. The mean difference in ratio was 44.29, and taking all the benchmarks as a whole, the difference in ratio was 2.47.

We also show the results of the compression and decompression time using gzip on BLPT and HPPDAG traces in Table VIII. For BLPT traces and HPPDAG traces, the second and the third columns show the compression time on them, respectively and the sixth column shows their difference in ratio. Similarly, the fourth, the fifth and the last column show the decompression time on the two kinds and their difference in ratio, respectively. We can see that the results are consistent with those shown in Table VIII. Taking all the benchmarks as a whole, compression and decompression on BLPT traces took 6.37x and 12.43x the time compared with those on HPPDAG traces, respectively. Moreover, the mean differences in ratio were 59.83 and 65.17, respectively.

In summary, the results in Table VII and Table VIII suggest that the HPP framework and existing compression techniques can be good complements to each other on reducing the size and the compression/decompression overhead of control flow traces.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Compression Time (in second)</th>
<th>Decompression Time (in second)</th>
<th>(A) ÷ (B)</th>
<th>(C) ÷ (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BLPT (A)</td>
<td>HPPDAG (B)</td>
<td>BLPT (C)</td>
<td>HPPDAG (D)</td>
</tr>
<tr>
<td>barnes</td>
<td>66.32</td>
<td>8.66</td>
<td>42.02</td>
<td>2.29</td>
</tr>
<tr>
<td>fmm</td>
<td>135.38</td>
<td>4.61</td>
<td>92.56</td>
<td>1.06</td>
</tr>
<tr>
<td>ocean-cp</td>
<td>42.93</td>
<td>0.11</td>
<td>30.09</td>
<td>0.09</td>
</tr>
<tr>
<td>ocean-ncp</td>
<td>41.92</td>
<td>0.11</td>
<td>30.19</td>
<td>0.08</td>
</tr>
<tr>
<td>radiosity</td>
<td>144.51</td>
<td>25.18</td>
<td>78.14</td>
<td>8.60</td>
</tr>
<tr>
<td>raytrace</td>
<td>136.85</td>
<td>3.38</td>
<td>88.72</td>
<td>0.68</td>
</tr>
<tr>
<td>voldrend</td>
<td>15.08</td>
<td>12.27</td>
<td>5.30</td>
<td>3.12</td>
</tr>
<tr>
<td>water-ns</td>
<td>109.36</td>
<td>4.16</td>
<td>63.50</td>
<td>2.66</td>
</tr>
<tr>
<td>water-sp</td>
<td>152.11</td>
<td>3.88</td>
<td>88.06</td>
<td>2.18</td>
</tr>
<tr>
<td>400.perlbench</td>
<td>70.48</td>
<td>0.42</td>
<td>27.95</td>
<td>0.26</td>
</tr>
<tr>
<td>401.bzip2</td>
<td>80.59</td>
<td>34.08</td>
<td>40.42</td>
<td>9.00</td>
</tr>
<tr>
<td>403.gcc</td>
<td>46.05</td>
<td>7.44</td>
<td>20.80</td>
<td>2.08</td>
</tr>
<tr>
<td>429.mcf</td>
<td>143.69</td>
<td>74.38</td>
<td>62.60</td>
<td>17.29</td>
</tr>
<tr>
<td>445.gobmk</td>
<td>117.94</td>
<td>31.56</td>
<td>44.17</td>
<td>8.08</td>
</tr>
<tr>
<td>456.hmmer</td>
<td>66.25</td>
<td>58.87</td>
<td>19.94</td>
<td>12.89</td>
</tr>
<tr>
<td>458.sjeng</td>
<td>83.14</td>
<td>33.19</td>
<td>32.41</td>
<td>8.75</td>
</tr>
<tr>
<td>462.libquantum</td>
<td>46.97</td>
<td>2.50</td>
<td>35.00</td>
<td>1.82</td>
</tr>
<tr>
<td>464.h264ref</td>
<td>335.05</td>
<td>38.20</td>
<td>179.98</td>
<td>13.87</td>
</tr>
<tr>
<td>473.astar</td>
<td>249.65</td>
<td>30.96</td>
<td>132.57</td>
<td>7.03</td>
</tr>
<tr>
<td>481.xalancbmk</td>
<td>25.84</td>
<td>0.57</td>
<td>12.30</td>
<td>0.15</td>
</tr>
<tr>
<td>Total</td>
<td>2108.10</td>
<td>374.52</td>
<td>1126.71</td>
<td>101.94</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td>59.92</td>
<td>65.25</td>
</tr>
</tbody>
</table>
4.5 Experimental Results on Control Flow Queries on HPPDAG

Two properties of the HPPDAG model are beneficial for control flow queries on top of it. First, the hierarchical structure of HPPDAG interconnects related path instances and procedure calls. Second, HPPDAG enables the reuse of the equivalent interprocedural paths. The first property allows related control flow information interconnected in HPPDAG rather than simply arranged by the time order. Thus, trace queries could save the cost on traversing through unrelated information. The second property benefits trace queries, which fall in the dynamic programming paradigm. That is the trace query can be broken down into several simpler ones (e.g., query or analysis on a subpath) and the results of these simple queries can be stored and reused. In our opinion, any control flow query or analysis, which could make use on either property, could largely benefit from HPP framework.

To assess to what extent path querying on HPPDAG can be efficient, we applied HPPDAG to the problem of backtracing of statically-determined program points. Specifically, given a recorded trace, such query on a program point \( x \) should report all backtraces, each of which leading to one executed instance \( x_n \) of \( x \). Note that the backtrace of \( x_n \) in our application of HPPDAG contained not only the active procedures in the call stack of the execution, as generated by gdb [GDB 2015], but also the executed intra-procedural paths of these active procedures. Such a backtrace contains all the control-dependency of the executed trace leading to \( x_n \) in the execution, and can be directly fed to many trace analysis techniques such as execution indexing [Xin et al. 2008] and program slicing [Zhang and Gupta 2001; Ohmann and Liblit 2013; Wang and Roychoudhury 2007].

In our experiment, we assessed backtrace queries by randomly selecting a subset of monitored procedures in the set \( F_{\text{monitored}} \) of each benchmark and setting their entries as the program points to retrieve their backtraces. We implemented the query on top of HPPDAG and also the traces generated by BLPT.

For queries on HPPDAG, the set of procedure call nodes (denoted as \( N_p \)) representing all procedure calls on a program point \( p \) to be back-traced in the execution trace, were firstly located. Then, a reverse depth-first traversal was performed for each node \( n_i \in N_p \), which enumerated all the paths from \( n_i \) to the root node of the HPPDAG instance. All procedure calls in one such path and their intra-procedural paths leading to the proceeding procedure calls were outputted to represent a backtrace of \( p \). For queries on a set of BLPT traces, the trace event stream of each BLPT trace was iteratively traversed with a stack maintained. Each stack element contained an unreturned procedure call and its intra-procedural paths. The stack was pushed on a procedure entry event, and was popped on a procedure exit event. On the occurrence of a procedure entry event matching with the program point \( p \) to be back-traced, the snapshot of the current stack was outputted to represent a backtrace of \( p \).

We note that, for each benchmark in our experiment, each executed thread of the benchmark resulted in one BPLT trace in the course of tracing, and readers may refer to Table I for the number of executed threads in our experiments. As such, the trace queries on BLPT traces had to be separately performed on each single trace and the query result of each trace has to be aggregated to form the final query result.

On the other hand, one single HPPDAG instance has represented the control flow of all executed threads for each benchmark. Thus, a single query on that HPPDAG instance is able to generate the final query result. Moreover, since an HPPDAG instance is composed of a set of subgraphs, each of which represents the control flow of an executed thread of a program, trace queries over the control flow of a user-specified executed thread can also be straightforwardly performed.
On each benchmark, we assessed the CPU time cost spent by 10 backtrace queries each on a randomly selected (without replacement) procedure entry point.

The experimental results on the efficiency to complete the queries on HPPDAG and BLPT are shown in Table IX. For each query of a program point x on an execution trace, each occurrence of the instances of x resulted in one reported backtrace. The total number of backtraces reported for the 10 performed queries on each benchmark is shown in the second column. The CPU time for processing the 10 queries on these BLPT traces and that on these HPPDAG instances are shown in the third and the fourth columns, respectively. The comparison is shown in the fifth column. Since BLPT traces were too large to be kept in the physical memory of our platform, they were dynamically loaded from its secondary storage to perform the backtrace queries. To allow a fair comparison, we also load the HPPDAG instances from the secondary storage and restore them in the main memory. The CPU time to load and restore HPPDAG instances is presented in the last column of Table IX. However, we note that the HPPDAG instance of each benchmark only needs to be loaded into the main memory once. All following queries for the same benchmark can be performed on the loaded HPPDAG instance.

The results show that in the experiment, the time cost spent by querying on HPPDAG strictly improved that of the queries on BLPT traces. Considering all the benchmarks as a whole, querying on BLPT traces takes 75.61 folds the time spent by querying on HPPDAG traces.

We view that the underlying reason on the efficiency of performing backtrace querying on HPPDAG is twofold. First, the hierarchical structure of HPPDAG interconnects the caller procedures and the corresponding callee procedures with their intra/inter-procedural paths. For backtrace queries, this organization circumvents the
overhead of processing unrelated trace data; whereas, querying on a BLPT trace requires filtering out those unrelated procedure call events and BL path events appearing along the trace. Second, all occurrences of the same equivalent inter-procedural paths have been consolidated into a single subgraph in an HPPDAG instance. Thus, one reversed depth-first traversal was able to locate multiple backtraces instead of needing to separately process each single one as what a query on each BLPT trace had to be done.

5. AN EXEMPLIFIED APPLICATION: PATH PROFILING ON INTER-PROCEDURAL PATHS

In this section, we present an exemplified and novel application of HPPDAG on path profiling.

Hot paths [Ball and Larus 1996], which are those frequently executed paths of a program execution, are useful information popularly used in compiler optimization and are able to guide developers to identify the performance bottlenecks of their programs. **Path profiling**, which is used to collect the execution frequency of a specified set of paths [Ball and Larus 1996; Melski and Reps 1999], is widely used to identify hot paths from program executions. To the best of our knowledge, the path profiling technique presented by Ball and Larus [1996], which adopts the path encoding algorithm revisited in Section 2.1, is regarded as an efficient and de facto technique to collect path profiles.

However, the above technique [Ball and Larus 1996] is restricted to profiling acyclic intra-procedural paths. Thus, it is unable to provide opportunities for further optimization on longer paths crossing an *indefinite* number of loop iterations or crossing any procedural boundaries. Previous work [Tallam et al. 2004; Roy and Srikant 2009; Li et al. 2012; D’Elia and Demetrescu 2013] has tried to alleviate this problem by profiling longer paths with the cost of incurring significant additional runtime overheads. For instance, D’Elia and Demetrescu [2013] contribute one of the most recent work in this area. Their work supports up to \(k\) concatenation of acyclic paths. However, \(k\) is only affordable to be a small number due to the combinatorial explosion problem of path combinations, and moreover, the value of \(k\) should be predetermined before the corresponding execution trace is profiled. Once the execution trace has been profiled, changing to a larger value of \(k\) without a new round of profiling is impossible. **WPP** [Larus 1999] records the entire control flow of program execution encoded in a context-free grammar. Each offline query of path profiles, performed as a search in its compressed data structure, is restricted to a specified maximal length \(L\), i.e., the queried paths must compose of \(L\) or less acyclic intra-procedural paths. These restrictions highly limit the applicability of these techniques.

HPPDAG allows efficient retrievals of the profiles of **complete** inter-procedural paths, which are referred to as paths that each starts at the entry of some procedure call and ends at the exit of that procedure call. The profiled complete inter-procedural paths have no constraint on their lengths, i.e., the number of comprising intra-procedural path segments and the number of involved procedure calls. Note that the complete inter-procedural paths captured by HPPDAG each can be represented by concatenation of acyclic paths, and thus are represented with coarser granularity than the representation taken by \(k\)-concatenation acyclic paths [D’Elia and Demetrescu 2013]. Our coarse granularity approach bypasses the exponential explosion of \(k\)-concatenation acyclic paths with the increase in \(k\), yet is capable of capturing the control flow profiles for inter-procedural paths across loop backedges and procedure boundaries.
In each HPPDAG instance, all the equivalent occurrences of the same complete inter-procedural path, modeled as an equivalence class as defined in Section 3.2, are represented by one and only one subgraph in the HPPDAG instance. Therefore, the profile of a complete inter-procedural path can be retrieved by counting the number of paths leading to the corresponding directed acyclic subgraph from the root node of that HPPDAG instance. We present our novel algorithm to retrieve the path profiles in an HPPDAG instance, which is shown in Algorithm 2.

Algorithm 2 is intuitively appealing, and runs as follows: For each node \( n \) in the HPPDAG instance, a counter \( \text{counter}(n) \) is created, initialized as 0, to record the number of paths leading to the above mentioned directed acyclic graph starting at \( n \). We use the variable \( \text{root} \) to stand for the root node of the HPPDAG instance which is initialized with counter value of 1. After all nodes have been processed, the HPPDAG path profile is generated. The profile of one complete inter-procedural path can be simply obtained from the counter of its corresponding procedure call node.

As a contrast, to find the hot paths with maximum length specified, WPP [Larus 1999] requires a non-trivial search in its graph-based trace representation on each such query (interested readers could see Figure 7 in the paper of WPP for detailed comparison).

Our implementation of Algorithm 2 is simple and straightforward as its design and takes less than 50 lines of source code. We evaluated the efficiency of the path profiling technique on the same set of subjects in SPLASH-2 benchmark suite [Woo et al. 1995] and the SPECint 2006 benchmark suite [Henning 2006] as we used in Section 4. For each benchmark, after obtaining the HPPDAG instance from its execution runs, we generated the path profile of the HPPDAG instance with our implementation of Algorithm 2.

One common usage of path profiles is to retrieve the set of paths that are heavily executed. Thus we also retrieved the top 100 most frequently executed complete inter-procedural paths (including tie cases) and evaluated the time cost. Specifically, the implementation iterated through all procedure call nodes of that HPPDAG instance.

**Algorithm 2. HPPDAG Path Profile Generation**

**Input:**
- \( \text{root} \): The root node of a HPPDAG instance.
- \( \text{NodeList} \): one topologically ordered list of nodes of that HPPDAG instance.

**Output:** Each procedure call node of the HPPDAG instance is labeled with the number of occurrences of its corresponding complete inter-procedural path.

```
for each node \( n \) in \( \text{NodeList} \) do
    \( \text{counter}(n) \leftarrow 0 \);
end

\( \text{counter}(\text{root}) \leftarrow 1 \);
for each node \( n \) in \( \text{NodeList} \) do
    for \( i \) from 0 to \( |\text{Children}(n)| \) do
        \( m = \text{Children}(n)[i] \)
        \( \text{counter}(m) \leftarrow \text{counter}(m) + \text{counter}(n) \);
    end
end
```

In each HPPDAG instance, all the equivalent occurrences of the same complete inter-procedural path, modeled as an equivalence class as defined in Section 3.2, are represented by one and only one subgraph in the HPPDAG instance. Therefore, the profile of a complete inter-procedural path can be retrieved by counting the number of paths leading to the corresponding directed acyclic subgraph from the root node of that HPPDAG instance. We present our novel algorithm to retrieve the path profiles in an HPPDAG instance, which is shown in Algorithm 2.
and used a binary heap to maintain 100 such nodes with the largest counter values. We recorded the time cost of path profile generation, which was to assess the implementation of Algorithm 2, and the time cost of retrieving the top 100 complete inter-procedural paths, respectively.

The experimental results are shown in Table X. The path profile generation time and the time to retrieve the top 100 hot complete inter-procedural paths are shown in the second and third columns, each being the mean value of five runs. The experimental results indicate that our path profiling technique, built on top of HPP framework, is able to construct the profiles of all the complete inter-procedural paths occurred in an arbitrary set of executions in a highly efficient manner. Moreover, the generated profiles support efficient retrieval of a set of inter-procedural paths of interest to users.

We visualized the path profiles of all complete inter-procedural paths occurred in the executions of each benchmark in Figure 11. For better readability, we divided all the benchmarks into four groups, each illustrated with a sub-figure, so that the path profiles in each group have close numbers of complete inter-procedural paths. For each benchmark, we sorted the complete inter-procedural paths occurred in the traced executions by their execution counts in ascending order. In each sub-figure, the x-axis indicates the indices of complete inter-procedural paths, and the y-axis indicates execution counts in the exponential scale. As such, the path profile of each benchmark is shown as a line in a sub-figure.

To the best of our knowledge, the technique presented in this section is the first path profiling technique that has no constraint on the length of the profiled paths and is able to efficiently and completely count every complete inter-procedural paths occurred in a set of execution traces.
Capturing the complete dynamic control flow of program executions is a core activity in many techniques. Except for path tracing techniques based on Ball and Larus path encoding algorithm [Ball and Larus 1996], many other techniques [Ayers et al. 2005; Zhao et al. 2006] employ straightforward approaches to directly record the sequence of basic blocks executed in an execution trace. The simplest approach is to record the label of each executed basic block, making the generated traces huge. DEP [Zhao et al. 2006] further utilizes the locality of consecutively executed basic blocks and only records a suffix of the basic block label, which is taken as a byte array, if the basic block shares the same prefix with its preceding basic block. TraceBack [Ayers et al. 2005] uses a combination of heavyweight probes and lightweight probes. Each heavyweight probe is labelled with the static address of a basic block and followed by a sequence of lightweight probes, each taking one bit to indicate the control flow branch taken. The advantage of the basic block recording approach is that the control flow path leading to an early termination, as we presented in Section 3.1.2, can be precisely recorded until the last executed basic block, which helps locate the crashing site for subsequent debugging activities. In contrast, BL path events are generated only after the corresponding BL path instances reached their end nodes, i.e., loop back-edges or

Fig. 11. The path profiles of all the complete inter-procedural paths occurred in the executions of each benchmark sorted by their execution counts in ascending order.

6. RELATED WORK
procedure exits (see Section 2.1). Early termination thus may incur loss of record for precisely locating the crashing site.

However, to the best of our knowledge, the above-reviewed approaches could not achieve path encodings as compact as those utilizing Ball and Larus algorithm.

On the contrary, HPP aims at general purpose applications rather than solely debugging. We therefore design HPP by adopting the BL path encoding. Besides, for the early termination problem, HPP provides customizable watch points to support trace recovering. Another reason on the adoption of the BL path encoding is its independence of the underlying system and hardware. BL path encoding works completely in the user mode, which supports a customizable scope of path tracing (i.e., tracking paths of selected procedures only).

Ball and Larus algorithm [Ball and Larus 1996] analyzes each procedure separately to encode intra-procedural path fragments as BL paths. Thus, BL paths cannot cross procedure boundaries and inter-procedural paths have to be represented as the composition of intra-procedural paths of caller and callee procedures. Whereas some work [Melski and Reps 1999] extends Ball and Larus algorithm to explicitly encode longer paths across procedure boundaries, which has the potential to generate less path events if used in path tracing. Melski and Reps [1999] propose to build a supergraph that is the inter-procedural control flow graph which interconnects the control flow graphs of all procedures. After the removal of cyclic paths, each acyclic path, which may be an inter-procedural path across multiple procedure boundaries, in this supergraph is then encoded. However, compared with Ball and Larus algorithm, their approach incurs a heavier runtime overhead. Besides, the supergraph inevitably incurs the problem of path explosion and cannot handle function pointers well. As commented by Tallam et al. [2004], this approach is “too expensive for use in practice”. There is also no experimental result to evaluate its effectiveness or efficiency.

On the contrary, HPP does not extend Ball and Larus algorithm. Rather, with the design of call site alignment, HPP stays with the original intra-procedural path encoding, and avoids the splitting of BL paths containing call sites, which has to be performed in the traditional BLPT technique to maintain the time order of path instances. As such, HPP generates a smaller number of path events, and it does not incur the problem of path explosion on explicitly encoding inter-procedural paths.

Some techniques focus on the collaborative collection and analysis of multiple program profiles, including program control flow. For example, eWPP [Tallam et al. 2005] efficiently collects memory dependency and control flow traces together. DEP [Zhao et al. 2006] captures the control flow, data dependency and memory references of an execution. WET [Zhang and Gupta 2004] is a unified representation of control flow paths, data dependences, values and addresses. These techniques are able to generate a smaller trace as a whole than independently collecting each program profile. More importantly, different profiles are inter-related so that trace queries involving multiple profiles can be efficiently performed. Our current HPP framework only supports the collection and analysis on the complete dynamic control flow of a set of executions. Extension of HPP to enlarge the scope of its representation on program profiles is an interesting future work.

Trace compression techniques, which typically adopt online compression on the fly as opposed to the general purpose compression techniques, have been studied to address the space inefficiency of keeping program traces. Sequitur [Nevill-Manning and Witten 1997] dynamically constructs a context-free grammar to encode a sequence of symbols in a space efficient manner. WPP [Larus 1999] adopts the Sequitur algorithm on an input sequence of control flow trace events to construct a hierarchical
Hierarchical Program Paths

representation of that sequence (i.e., a context-free grammar). Both WPP and HPP can identify and merge repetitive structures observed from control flow traces. However, the hierarchical representation generated by WPP is unaware of the semantics of the inputted sequence of trace events. Suppose that a subsequence of consecutive events in the trace as a whole models the semantics of an inter-procedural path. The context-free grammar, which is unaware of such modeling, may separate such a subsequence into several intermediate structures in the compressed trace. Thus, to query on this inter-procedural path, decompression has to be performed to rearrange the compressed trace to recover the original subsequence. In contrast, each subgraph in an HPPDAG instance models a segment of program path with self-contained semantics as defined in Section 3. At the same time, related path segments (e.g., intra-procedural paths of a same procedure invocation, paths of caller procedure and callee procedure) are interconnected by the hierarchical structure of HPPDAG. Hence, not only trace queries can be directly performed on an HPPDAG instance, but also these queries can efficiently locate related control flow information in execution traces. Another work APP [Renieris et al. 2005] advances the memory efficiency and compression speed of WPP. It models the control flow trace of an execution as a sequence of bits, each indicating the true branch or the false branch taken by a conditional. In the course of collection of this sequence of bits, branch predictors are used to facilitate arithmetic coding for compression. However, it bears the limitation, same as WPP, that the resultant encoded trace is unaware of the semantics of the data being encoded. The VPC algorithm [Burtscher 2004; Burtscher et al. 2005] takes another approach of trace compression by maintaining a set of value predictors, which are updated with incoming values. If the prediction is correct, the incoming value can be represented with fewer bits. Ketterlin and Clauss [2008] propose an algorithm to represent an address trace as a sequence of loop nests. When unfolded, the original trace is able to be regenerated. They report experiments on compression of load address trace, showing that their technique is able to outperform VPC4 [Burtscher et al. 2005] in compression ratio. Since HPP framework adopts a two-phase strategy, the above mentioned online trace compression techniques can be trivially integrated with the online tracing of HPP in Phase 1. As a consequence, however, a decompression process will then be required in advance of the following Phase 2 of HPP. A comprehensive evaluation on the cost and benefit is an interesting future work.

To alleviate the overhead of full trace decompression, some existing studies explore the opportunities to directly perform special-purpose queries on compressed traces. In the paper of WPP [Larus 1999], a technique to retrieve hot subpaths from the compressed hierarchical representation is presented. Scholz and Mehofer [2002] developed a technique to perform dataflow frequency analysis on top of WPP. Lin et al. [2007] extends WPP’s path queries to enable the query on the first occurrence (but not the other occurrences) of a specific path in WPP. However, the scopes of supported direct queries by these techniques are narrow, and each specific type of query requires a specialized technique. TWPP [Zhang and Gupta 2001] aims at providing efficient queries on path traces pertaining to a particular procedure, which requires decompression over only a subset of trace data. TWPP formalized the execution trace into intra-procedural paths linked by the dynamic call graph. All intra-procedural paths of the same procedure are grouped together and the repeated execution patterns are extracted represented with dictionary-like “dynamic basic blocks”. The benefit of such design is on querying the path traces of a particular procedure, TWPP requires only an on-demand decompression on the “dynamic basic blocks” of intra-procedural
paths pertaining to that particular procedure sparing the effort of examining the rest of the compressed trace. However, TWPP’s approach is only able to identify repetitive patterns in intra-procedural paths, and for queries over path traces that span over many different procedures, a large part of the compressed trace still needs to be decompressed. HPPDAG does not incur this limitation.

7. CONCLUSION
In this paper, we have presented a novel path tracing framework HPP to capture the complete dynamic control flow of program executions. HPP constructs a novel graph-based representation HPPDAG to model the control flow of one or multiple program executions in an intuitively appealing manner. Each subgraph of an HPPDAG instance represents a set of identical inter-procedural paths in the complete control flow traces. The execution order of instructions is also maintained by the hierarchical and compositional structure of the HPPDAG instance. Both intra-procedural path querying and inter-procedural path querying on an HPPDAG instance can be performed by firstly locating the corresponding subgraph(s) followed by graph traversals on them. We have performed comprehensive experimentation using the SPLASH2 benchmark suite and the SPECint 2016 benchmark suite to validate HPP. In the experiment, on average, across the benchmark suites, BLPT generated 42.1% more BL path events, 28% larger trace logs, and ran 21% slower than Phase 1 of HPP. The BLPT trace logs were 72.19 folds the size of the HPPDAG counterparts on average. At the same time, when compressed with gzip, BLPT traces were still averagely 44.29 folds the size of its HPPDAG counterparts, which suggests HPP can be a good complement of existing compression techniques. We have also shown that performing backtrace querying on HPPDAG traces is significantly more efficient than performing the same querying on BLPT traces. We further demonstrated the usability of HPPDAG model by formulating a novel and intuitive path profiling technique on top of HPP framework.

In the experiment, by comparing Table VI and Table VII, we have found that applying the compression technique gzip has a potential to further reduce the size of HPPDAG by more than 10 folds. We tend to believe that by uncovering the underlying reasons apart from inefficient symbol representation (e.g., the use of 32 bits to represent each index or number), further significant advancement may be possible. We leave this study as a future work. Another important aspect of HPP is that it is a two-phase strategy. In HPP, an HPPTree event stream is generated on-the-fly while tracing an execution. In theory, each event emitted by Phase 1 of HPP can be immediately processed by Phase 2 of HPP without the need of intermediate secondary storage. However, we do not claim the generation of HPPDAG is an on-the-fly technique yet, because from the current experiment, the generation of HPPDAG is a sequential process and is significantly less efficient than Phase 1. We leave the design of an on-the-fly and efficient version of Phase 2 as a future work.

As shown in Figure 1, the complete control flow traces of a set of executions on the same benchmark can be modeled by a single HPPDAG instance. As such, repetitively occurred inter-procedural path instances both in each single execution and across different executions are represented as one subgraph in the final HPPDAG instance. This enables interesting control flow analysis over executions of a test suite. For example, a path coverage measurement over a test suite can be developed on an HPPDAG instance representing all the executions of this test suite. Besides, comparisons among different execution traces can be retrieved from an HPPDAG instance. For example, a structural execution indexing technique [Xin et al. 2008], can
be developed on top of the HPPDAG model. These interesting works still require substantial efforts and will be explored as our future work.

In our current implementation of HPP, we have not collected the synchronization order among different threads in the same execution. However, HPP can be extended to independently collect the synchronization orders, e.g., happens-before order [Flanagan and Freund 2009] or lock order [Cai and Chan 2014], and match them with instructions captured by the path nodes in HPPDAG instances. Such extension is able to support concurrency analysis based on HPPDAG and can be a piece of interesting future work.

REFERENCES


Hierarchical Program Paths


Received Month YYYY; revised Month YYYY; accepted Month YYYY