Efficient On-the-fly Analysis of Program Behavior and Static Cache Simulation *

Frank Mueller and David B. Whalley

Dept. of Computer Science, Florida State University, Tallahassee, FL 32306-4019
e-mail: whalley@cs.fsu.edu *phone: (904) 644-3506*

Abstract. The main contributions of this paper are twofold. First, a general framework for control-flow partitioning is presented for efficient on-the-fly analysis, *i.e.* for program behavior analysis during execution using a small number of instrumentation points. The formal model is further refined for certain analyses by transforming a program's call graph into a function-instance graph. Performance evaluations show that the number of measurement points can be reduced by one third using these methods.

Second, the method of static cache simulation is introduced. Static cache simulation provides the means to predict a large number of cache references prior to the execution time of a program. The method is based on a variation of an iterative data-flow algorithm commonly used in optimizing compilers. It utilizes control-flow partitioning and function-instance graphs for predicting the caching behavior of each instruction. To our knowledge, no prior work has been done on predicting caching behavior statically. A detailed description is provided for instruction cache analysis, which is then discussed for a variety of applications ranging from fast instruction cache performance evaluation to analytical bounding of execution time for real-time applications.

1 Introduction

Program analysis through profiling and tracing has long been used to evaluate new hardware and software designs. The traditional approach relies on generating a program trace during execution that is analyzed later by a tool. The problem of generating a minimal trace, which can later be expanded to a full event-ordered trace, can be regarded as solved [3]. A near-optimal (often even optimal) solution to the problem for a control-flow graph G can be found by determining a maximum spanning tree $max(G)$ for the control-flow graph and inserting code on the edges of $G - max(G)$.

Recently, tracing and analyzing of programs has been combined using on-the-fly analysis [5]. This analysis technique requires that events are analyzed as they occur but does not require the storage of trace data. The analysis is performed during program execution and is specialized for a certain application (*e.g.* counting hits and misses for cache evaluation). The results of the analysis are available at program termination such that no post-execution analysis by any tool is required. If the application or the configuration changes, the program has to be executed again. In contrast, trace data can be analyzed by several tools and for several configurations once the data is generated. But the generation of trace data is typically slow and space consuming since the data is written to a file and later read again by a tool.

On-the-fly analysis requires that the program be instrumented with code that performs the analysis. Many applications, including cache simulation, require that all events are simulated in the order in which they occur. In the past, each basic block was instrumented with code to support event-ordered analysis [11]. Inserting code based on the maximum spanning tree (or, to be more

* Supported in part by the Office of Naval Research, contract # N00014-94-1-0006.

precise, on its complement) does not cover all events and is therefore not applicable to on-the-fly analysis. This paper provides the framework to reduce code instrumentation to a small number of places. This framework supports efficient on-the-fly analysis of programs with regard to path partitioning. The execution overhead can be further reduced by analyzing instances of functions. The construction of a function-instance graph from a program's call graph is presented in this paper.

One application for program analysis is cache evaluation. Different cache configurations can be evaluated by determining the number of cache hits and misses for a set of programs. Cache analysis can be performed on-the-fly or by analyzing stored trace data, though faster results have been reported for the former approach [12]. This paper introduces the method of static cache simulation which predicts the caching behavior of a large number of references prior to execution time. The method employs a novel view of cache memories, which is, to our knowledge, unprecedented. The method is based on a variation of an iterative data-flow algorithm commonly used in optimizing compilers. It can be used to reduce the amount of instrumentation code inserted into a program for on-the-fly analysis. It can also be used to enable a program timing tool to take the effects of caching into account. A more comprehensive overview of static cache simulation including further applications can be found elsewhere [8].

2 Control-Flow and Call-Graph Analysis

In this section, terms and methods are introduced to analyze the call graph of a program and the control-flow graphs of each function. The analysis is performed to find a small set of measurement points suitable for on-the-fly analysis. The analysis provides a general framework to reduce the overhead of event-ordered profiling and tracing during program execution.

The first part of this section provides a formal approach for determining a small set of measurement points for on-the-fly analysis. The focus is restricted to the analysis of the control-flow graph of a single function. In the second part, the analysis is extended to the call graph of the entire program by transforming a call graph into a function-instance graph.

2.1 Partitioning the Control-Flow Graph into Unique Paths

The control flow of each function is partitioned into unique paths (UPs) to provide a small set of measurement points. The motivation for restructuring the control flow into UPs is twofold.

1. Each UP has a unique vertex or edge that provides the insertion point for instrumentation code at a later stage. This code may perform arbitrary on-the-fly analysis, *e.g.* simple profiling or more complex cache performance analysis.
2. Each UP is comprised of a range of instructions that are executed in sequence *if and only if* the unique vertex or edge is executed. This range of instructions does not have to be contiguous in the address space. The range of instructions provides a scope for static analysis to determine the instrumentation code for dynamic on-the-fly analysis, which preserves the order of events.

The first aspect, the strategy of instrumenting edges (or vertices where possible), is also fundamental to the aforementioned work on optimal profiling and tracing by Ball and Larus [3]. It is the second aspect that distinguishes this new approach from their work. The option of performing static analysis on the control

flow to determine and optimize the instrumentation code for order-dependent on-the-fly analysis requires the definition of ranges for the analysis. Naively, one could choose basic blocks to comprise these ranges. But it has been demonstrated for profiling and tracing that fewer instrumentation points can be obtained by a more selective instrumentation technique. UPs provide such a framework supporting efficient instrumentation for on-the-fly analysis.

The set of UPs is called a unique path partitioning (UPPA) and is defined as follows: Let $G(V, E)$ be the control-flow graph (directed graph) of a function with a set of edges (transitions) E and a set of vertices (basic blocks) V.

Let p be a path $p = \nu_0, \epsilon_1, \nu_1, ..., \epsilon_n, \nu_n$ with the ordered set of edges $\epsilon_p = \{\epsilon_1, ..., \epsilon_n\} \subseteq E$ and the ordered set of vertices $\nu_p = \{\nu_0, ..., \nu_n\} \subseteq V$, *i.e.*, a sequence of distinct vertices connected by edges [6]. The edge ϵ_i may also be denoted as $\nu_{i-1} \rightarrow \nu_i$. Vertex ν_0 is called an *head vertex* and vertex ν_n a *tail vertex*, while all other ν_i are *internal vertices*. Let H be the set of all head vertices and T be the set of all tail vertices.

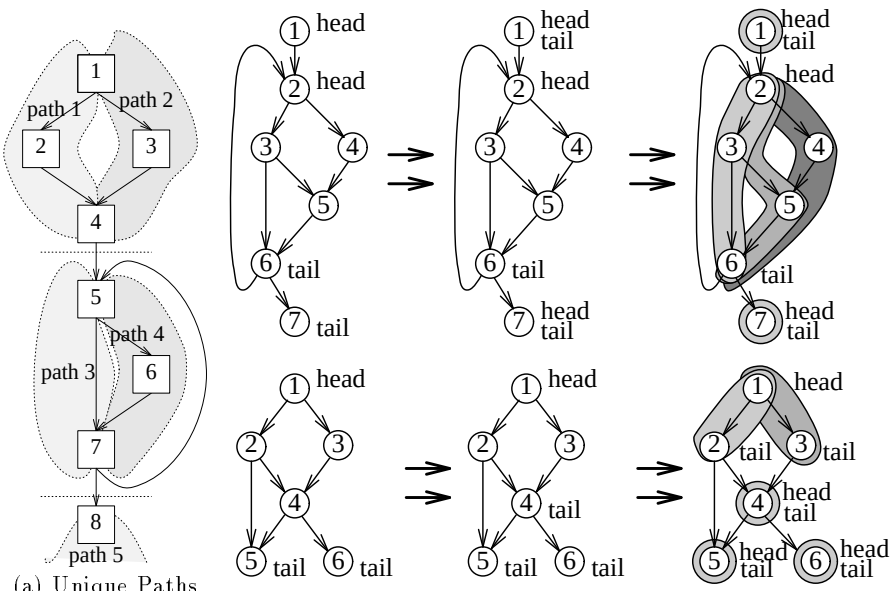
Definition 1 (UPPA). A unique path partitioning, $UPPA$, for a control-flow graph G(V,E) is a set of paths $p(\nu, \epsilon)$ with the following properties:

1. all vertices are covered by paths: $\forall_{v \in V} \exists_{p \in UPPA} \; v \in \nu_p$.
2. each edge is either on a path or it connects a tail vertex to a head vertex: $\forall_{e=(v \rightarrow w) \in E} \exists_{p \in UPPA} \; e \in \epsilon_p \oplus v \in T \wedge w \in H$
3. each path has a feature f, an edge or a vertex, which is *globally* unique, *i.e.* f is in no other path: $\forall_{p \in UPPA} \; (\exists_{e \in E} \; e \in \epsilon_p \wedge \forall_{q \in UPPA \setminus \{p\}} \; e \notin \epsilon_q) \vee (\exists_{v \in V} \; v \in \nu_p \wedge \forall_{q \in UPPA \setminus \{p\}} \; v \notin \nu_q)$
4. overlapping paths only share an initial or final subpath: $\forall_{p,q \in UPPA} \; \nu_p \cap \nu_q = \alpha \cup \beta$ where α and β denote the vertices of a common initial and final subpath, respectively. In other words, let $\nu_p = \{\nu_0, ..., \nu_m\}$ and $\nu_q = \{\omega_0, ..., \omega_n\}$ be the ordered sets of vertices for paths p and q. Then, $\alpha = \phi$ or $\alpha = \{\nu_0 = \omega_0, ..., \nu_i = \omega_i\}$ and $\beta = \phi$ or $\beta = \{\nu_k = \omega_l, ..., \nu_m = \omega_n\}$ for $i < k$ and $i < l$.
5. proper path chaining: $\forall_{p,q \in UPPA} \; \forall_{v \in \nu_p, w \in \nu_q} \; e = (v \rightarrow w) \in E \wedge e \notin \epsilon_p \cup \epsilon_q \Rightarrow v \in T \wedge w \in H$
6. break at calls: Let $C \subseteq V$ be the set of vertices (basic blocks) terminated by a call instruction. $\forall_{v \in C, p \in UPPA} \; v \in \nu_p \Rightarrow v \in T$
7. break at loop boundaries: Let L_i be the set of vertices in loop (cycle) i and let L be the set of all L_i. $\forall_{e=(v \rightarrow w) \in E, \; p \in UPPA, \; L_i \in L} \; e \in \epsilon_p \Rightarrow (v \in L_i \Leftrightarrow w \in L_i)$

The properties 6 and 7 are operational restrictions motivated by the application of the partitioning for on-the-fly analysis of program behavior. The break at calls allows the insertion of instrumentation code for separate compilation. Thus, the compiler is not required to perform interprocedural analysis. The break at

loop boundaries ensures that the frequency of events can be identified. The frequency of events outside a loop differs from the frequency inside loops (unless the loop was iterated only once). Thus, a UP associated with an event should not cross loop boundaries.

Example 1. Paths 1 and 2 in Figure 1(a) have two unique transitions each. They comprise an if-then-else structure. Paths 3 and 4 are generated because the loop is entered after basic block 4. Path 3 only has one unique transition while path 4 has two. Basic block 8 is outside the loop and therefore lies in a new path.



(a) Unique Paths

(b) Algorithmic Construction of 2 Small UPPAs

Fig. 1. Unique Paths in the Control-Flow Graph

Theorem 2 (Existence of a UPPA). *Any control-flow graph G(V,E) has a UPPA.*

Proof. Let G(V,E) be a control-flow graph. Then, $UPPA_b = \{\{v_0\}, ..., \{v_n\}\}$ is a unique path partitioning, *i.e* each vertex (basic block) constitutes a UP. Each property of Definition 1 is satisfied:

1. Any vertex v_i is part of a UP $p_i = \{v_i\}$ by choice of the partitioning.
2. $\forall_{p \in UPPA_b} \quad \epsilon_p = \phi$ since all edges connect paths.
3. Each UP $p_i = \{v_i\}$ has a vertex, namely v_i, which is unique to the path.
4. $\forall_{p,q \in UPPA_b} \quad p \neq q \Rightarrow \nu_p \cap \nu_q = \phi$.
   None of the UPs overlap in any vertex as shown for the previous property.
5. $\forall_{p_i = \{v_i\} \in UPPA_b} \quad v_i \in H \wedge v_i \in T$
6. The proof for the previous property suffices to prove this property as well.
7. $\forall_{e \in E,\ p \in UPPA_b}$ $e \notin \epsilon_p$, so the premise can never be satisfied. Thus, the property is preserved. □

Definition 3 (Ordering of UPPAs). For a control-flow graph $G(V, E)$, a partitioning $UPPA_a$ is smaller than a partitioning $UPPA_b$ if $UPPA_a$ contains fewer paths than $UPPA_b$.

The significance of the ordering is related to the number of measurement points for on-the-fly analysis. A smaller partitioning yields fewer measurement points, which improves the performance of on-the-fly analysis. The following algorithm provides a method to find a small partitioning. The algorithm uses the terminology of a loop header for a vertex with an incoming edges from outside the loop. A loop exit is a vertex with an outgoing edge leaving the loop. This is illustrated in Figure 2a.

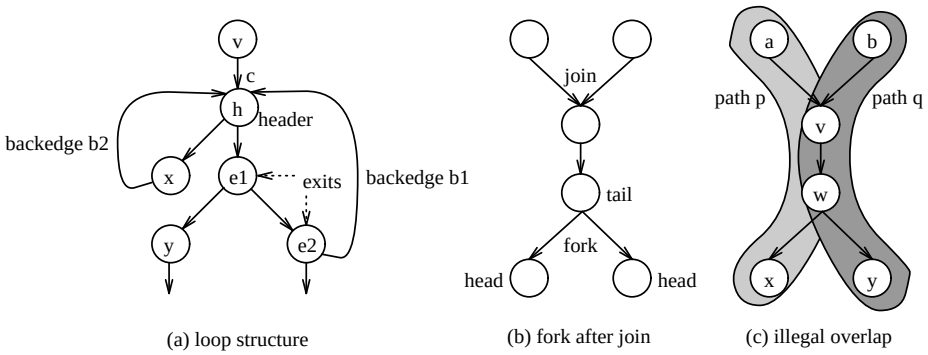


Fig. 2. Sample Graphs

Algorithm 1 (Computation of a Small UPPA) .
Input: Control-flow graph G(V,E).
Output: A small partitioning $UPPA$.
Algorithm: Let C be the set of vertices containing a call, let L_i be the set of vertices in loop i, and let L be the set of all L_i as in Definition 1. The algorithm then determines the beginning of paths (*heads*) and the end of paths (*tails*), for example at loop boundaries. In addition, a vertex is a tail if the path leading to this vertex *joins* with other paths and *forks* at the current vertex (see Figure 2b). Once the heads and tails have been determined, a path comprises a sequence of vertices and edges from a head to a tail in the control flow.

BEGIN
  FOR each $v \in V$ without any predecessor DO
    mark v as *head*; /* entry blocks to the function */
  FOR each $v \in V$ without any successor DO
    mark v as *tail*; /* return blocks from the function */
  FOR each $v \in C$ DO
    mark v as *tail*; /* calls */
  FOR each $e = (v \rightarrow w) \in E$ WITH $v \notin L_i$ AND $w \in L_i$ DO
    mark w as *head*; /* loop headers */
  FOR each $e = (v \rightarrow w) \in E$ WITH $v \in L_i$ AND $w \notin L_i$ DO
    mark v as *tail*; /* loop exits */
  FOR each $v \in V$ DO
    mark v as not *done*;

```
    WHILE change DO
        change:= False;
        propagate_heads_and_tails;
        FOR each v ∈ V WITH v marked as head AND
                not marked as done AND not marked as tail DO
            change:= True;
            mark v as done;
            FOR each e = (v → w) ∈ E DO
                recursive_find_fork_after_join(w, False);

    UPPA = φ
    FOR each v ∈ V with v marked as head DO
        recursive_find_paths(v, {v});
END;

PROCEDURE propagate_heads_and_tails IS
    WHILE local_change DO
        local_change:= False;
        FOR each v ∈ V DO
            IF v marked as head THEN
                FOR each e = (w → v) ∈ E DO
                    IF w not marked as tail THEN
                        local_change:= True;
                        mark w as tail;
            IF v marked as tail THEN
                FOR each e = (v → w) ∈ E DO
                    IF w not marked as head THEN
                        local_change:= True;
                        mark w as head;
END propagate_heads_and_tails;

PROCEDURE recursive_find_fork_after_join(v, joined) IS
    IF v marked as tail THEN
        return;
    IF v joins, i.e. v has more than once predecessor THEN
        joined:= True;
    IF joined AND v forks, i.e. v has more than once successor THEN
        mark v as tail;
        return;
    FOR each e = (v → w) ∈ E DO
        recursive_find_fork_after_join(w, joined);
END recursive_find_fork_after_join;

PROCEDURE recursive_find_paths(v, p) IS
    IF v marked as tail THEN
        UPPA = UPPA ∪ {p};
    ELSE FOR each e = (v → w) ∈ E DO
        recursive_find_paths(w, p ∪ {v → w, w});
END recursive_find_paths;
```

Example 2. Figure 1(b) illustrates two examples of the construction of a small UPPA using Algorithm 1. For the first example (upper part of Figure 1(b)), vertices without predecessor (successor) are marked as head (tail). In addition, loop headers are heads and loop exits are tails. The second picture shows the same graph after `propagate_heads_and_tails` has been applied. Block 1 is marked as a tail since block 2 is a head. Conversely, block 7 is marked as a head since block 6 is a tail. The last picture depicts the graph after path partitioning through `recursive_find_paths`. Each head is connected to the next tail by one or more paths, depending on the number of different ways to reach the tail. The resulting UPPA has 5 paths.

The second example (lower part of Figure 1(b)) initially shows a graph whose vertices without predecessor (successor) are marked as heads (tails). The second picture shows an additional tail found by `recursive_find_fork_after_join` since there is a possible traversal for the head block 1 to, for example, the tail block 6, which encounters a join followed by a fork in block 4. The final graph depicts the effect of `propagate_heads_and_tails`. Blocks 5 and 6 are a head since 4 was a tail. Block 2 is a tail since block 5 is now a head. Thus, block 4 becomes a head. This causes block 3 to be marked as a tail. Finally, `recursive_find_paths` partitions the graph resulting in a UPPA with 5 paths.

Theorem 4 (Correctness of Algorithm 1). *Algorithm 1 constructs a UPPA for a control-flow graph $G(V, E)$.*

Proof. **Termination:** It suffices to show that the WHILE loops and the recursive routines terminate. Both WHILE loops terminate since one more vertex is marked as head or tail during each iteration. This process terminates either when all vertices are marked as heads and tails or when none of the conditions for marking vertices are satisfied any longer. The recursive routine `recursive_find_fork_after_join` terminates for the following reasons. Initially, all loop headers are marked as heads. The propagation of heads and tails ensures that all predecessors of loop headers are marked as tails, in particular the vertices preceding a backedge in a loop. Since `recursive_find_fork_after_join` terminates when a tail is encountered, it will stop at a tail vertex with an outgoing backedge or at a tail vertex without any successor since it can only traverse forward edges in the control-flow graph. This also applies for `recursive_find_paths`.
Output is a UPPA: It has to be shown that the properties of a UPPA as stated in Definition 1 hold for Algorithm 1.

1. All vertices are covered since `recursive_find_paths` includes all vertices between a head and a tail in some path. Due to `propagate_heads_and_tails`, an outgoing edge of a tail vertex always leads to a head vertex, *i.e.* there cannot be any intermediate vertices between a tail and a head. Furthermore, at least the initial vertex (without predecessor) is a head and the final vertices (without successors) are tails.
2. Consider any edges between a head and a tail. These edges are included in some path by `recursive_find_paths`, and these are all edges on paths. The remaining edges are those connecting tails to heads and are not in any path.
3. The following cases have to be distinguished for construction of paths by `recursive_find_paths`: If there are no forks between a head h and the next tail, then there will only be one path starting at h, and h is a unique vertex for this path. If there are forks after a head h but no joins, then the tail vertex will be unique for each path starting in h. If there are forks after a head h, followed by the first join at vertex v along some path starting in h, then the edge immediately preceding v on this path will be unique (since no other path has joined yet). Notice that there cannot be another fork after the join in v within the path since any forking vertex would have been marked as a tail by `recursive_find_fork_after_join`.
4. Property 3 ensures that any two overlapping paths differ in at least an edge. (Notice that a unique vertex implies a unique edge for non-trivial paths with multiple vertices.) Assume there exist two paths p, q that overlap in a subpath $\{v, ..., w\}$ (see Figure 2c) and v is preceded by distinct vertices a

and b in p and q, respectively. Also, w is succeeded by distinct vertices x and y in p and q, respectively. In other words, p and q overlap somewhere in the middle of their paths. Then, two edges join in vertex v and two edges fork from vertex w, *i.e.* a join is followed by a fork. Thus, w should have been mark as a tail by `recursive_find_fork_after_join`. Therefore, w should have been the last vertex of paths p and q. Contradiction.

5. All edges between a head and the next tail are covered by paths, as shown for property 2. Thus, it suffices to observe that edges connecting a tail t to a head h always connect all paths ending with vertex t to the paths starting with vertex h. It is guaranteed by `recursive_find_paths` that a path starts with a head vertex and ends in a tail vertex.
6. Each vertex v containing a call is initially marked as a tail vertex. Thus, vertex v must be the final vertex for any path containing v by construction of the paths (`recursive_find_paths`).
7. Each loop header vertex is initially marked as a head and each loop exit is marked as a tail. Thus, the vertices preceding a loop header are marked as a tail and the vertices succeeding a loop exit are marked as heads by `propagate_heads_and_tails`. Furthermore, the edges crossing loop boundaries connect the paths ending in the tail vertex to the paths starting with the head vertex. As already shown for property 2, edges between a tail and a head cannot be covered by any path.

□

In terms of the ordering of UPPAs, the basic block partitioning $UPPA_b$ is the partitioning with the largest number of measurement points. Algorithm 1 constructs a partitioning that has an equal or smaller number of measurement points. We found that the algorithm produces a much smaller UPPA if possible. The algorithm may in fact produce a minimal UPPA (with the smallest possible number of measurement points). We have not yet succeeded in proving the minimality due to the fact the a given graph may have more than one minimal UPPA.

In summary, the control-flow graph can be transformed into a small UPPA by Algorithm 1. The small set of measurement points is given by a unique vertex or unique edge of each UP. This provides the framework for efficient on-the-fly analysis with regard to the definition of UPPAs.

Another short example for a small UPPA construction shall be given, which is used to discuss the possibility of letting paths begin and end in edges as well as vertices.

Example 3. Consider the subgraph of Figure 2a that is inside the loop. A corresponding $UPPA_s$ can be constructed by Algorithm 1 resulting in the following partitioning:

$$UPPA_s = \{\{h, h \to x, x\}, \{h, h \to e1, e1\}, \{e2\}\}$$

In general, the method may still be further tuned with regard to the dynamic behavior. Currently, a path has to begin and end in a vertex. Consider the notion of *open paths* that can start and end in a vertex *or* an edge. Then, another small UPPA of the loop in Figure 2a would be:

$UPPA_t = \{\{h, h \to x, x\}, \{h, h \to e1, e1, e1 \to y\}, \{h, h \to e1, e1, e1 \to e2, e2\}\}$

Consider the number of measurement points executed during each loop iteration. For $UPPA_s$, there are two measurement points for an iteration reaching $b1$, one each in paths 2 and 3. For $UPPA_t$, there is only one measurement point on $b1$ in path $3'$. The definition of UPPAs does not take dynamic properties into account.

2.2 Call Graph Transformation into Function-Instance Graph

The small set of measurement points provides the location for inserting measurement code that records the order of events. While the actual measurement code depends on the intended analysis of the program, the amount of the measurement code may be further reduced by distinguishing between different call sites of a function. For an event-ordered analysis, the first invocation of a function may trigger certain initialization events. The analysis of subsequent calls to the same function are simplified by the assumption that these initialization events have already occurred. Such an example will be illustrated later in the context of instruction cache analysis.

A program may be composed of a number of functions. The possible sequence of calls between these functions is depicted in a call graph [1]. Functions can be further distinguished by function instances. An instance depends on the call sequence, *i.e.* on the immediate call site of its caller, the caller's call site, etc. The function instances of a call graph are defined below. The definition excludes recursive calls that require special handling and are discussed later. Indirect calls are not handled since the callee cannot be statically determined.

Definition 5 (Function Instances). Let $G(V, EC)$ be a call graph where V is the set of functions including an initial function "main" and EC is a set of pairs (e, c). The edge $e = v \rightarrow w$ denotes a call to w within v (excluding recursive and indirect calls). The vertex c is a vertex of the control-flow graph of v that contains a call site to w. Then, the set of function instances is defined recursively:

1. The function (vertex) "main" has a single instance $main_0$.
2. Let $(f \rightarrow g, c) \in EC$ and f_i be an instance of f. Then, g_{c,f_i} is an instance.
3. These are all the function instances.

The call graph of a program without recursion (*i.e.*, a directed acyclic graph) can be transformed into a tree of function instances by a depth-first search traversal of the call graph. Function instances can then be uniquely identified by their index, where f_i denotes the ith occurrence of function f within the depth-first search. Backedges in the call graph corresponding to recursive calls can be detected by marking vertices as visited during the depth-first traversal. If an already visited edge is encountered again, the last edge in the current traversal is due to recursion. The depth-first search will then backtrack and retain this backedge as a special edge in the function-instance graph (see algorithm in [8]).

Example 4. In Figure 3, function f contains three calls: a call to g and two calls to h. Function g calls i and k. Function h calls k. Function i calls g, which is an indirect recursive call. The corresponding function-instance graph contains two instances of h (for each call from f_0) and three instances of k (for the calls from g_0, h_0, h_1). The backedge $i \rightarrow g$ due to indirect recursion is retained as a special edge in the function-instance graph.

2.3 Performance Evaluation

This section evaluates the benefits of control-flow partitioning and function-instance graphs to reduce the number of measurement points. Table 1 summarizes the performance tests for user programs, benchmarks, and UNIX utilities. The numbers were produced by modifying the back-end of an optimizing compiler VPO (Very Portable Optimizer) [4] to determine measurement points by partitioning the control flow and by creating the function-instance graph.

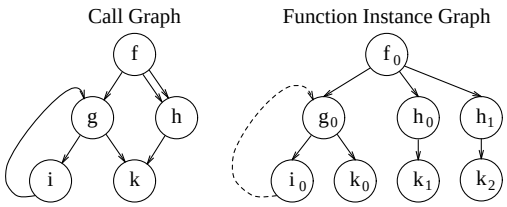


Fig. 3. Construction of Function-Instance Graph

Table 1. Test Set of C Programs

Name	Description	Size [bytes]	Instructions exec.	Measure Pts. static	Measure Pts. exec.
cachesim	Cache Simulator	8,460	2,995,817	73.38%	60.56%
cb	C Program Beautifier	4,968	3,974,882	89.62%	65.61%
compact	Huffman Code Compression	5,912	13,349,997	68.89%	56.56%
copt	Rule-Driven Peephole Optimizer	4,148	2,342,143	84.19%	74.88%
dhrystone	Integer Benchmark	1,916	19,050,093	81.61%	72.73%
fft	Fast Fourier Transform	1,968	4,094,244	78.43%	74.08%
genreport	Detailed Execution Report Generator	17,720	2,275,814	71.58%	81.31%
mincost	VLSI Circuit Partitioning	4,448	2,994,275	83.19%	76.27%
sched	Instruction Scheduler	8,272	1,091,755	73.16%	58.29%
sdiff	Side-by-side Differences between Files	7,288	2,138,501	72.13%	77.82%
tsp	Traveling Salesman	4,724	3,004,145	64.08%	58.67%
whetstone	Floating point benchmark	4,816	8,520,241	70.49%	68.25%
average		6,220	5,485,992	75.90%	68.75%

The size of the programs varied between about 2kB and 18kB (see column 3). The number of instructions executed for each program comprised a range of 1 to 19 million using realistic input data for each program (see column 4) . Column 5 indicates the percentage of measurement points required for our method versus the number of measurement points inserted in conventional on-the-fly analysis (*i.e.*, one measurement point per basic block). Our method requires only 76% of the measurement points required for the traditional trace-driven analysis, *i.e.* about 24% fewer measurement points statically. The run-time savings (column 6) are even higher, requiring only about 69% of the measurement points executed under traditional trace-driven analyses. The additional dynamic savings are due to reducing sequences of basic blocks inside loops to fewer UPs, sometimes just to a single UP. In other words, unique paths tend to be longer than basic blocks. For example, an iteration through the innermost loop may only require a single measurement point in one path (with multiple basic blocks).

3 Static Cache Simulation

One application for on-the-fly program analysis is cache performance evaluation. This section introduces the method of static cache simulation, which statically predicts the caching behavior of a large number of instruction references [2]. The method employs a novel view of cache memories that seems to be unprecedented.

[2] Data cache references could be predicted similarly but are not discussed here.

3.1 Instruction Categorization

Static cache simulation calculates the abstract cache states associated with UPs. The calculation is performed by repeated traversal of the function-instance graph and the UPPA of each function.

Definition 6 (Potentially Cached). A program line l can potentially be cached if there exists a sequence of transitions in the combined UPPAs and function-instance graph such that l is cached when it is reached in the UP.

Definition 7 (Abstract Cache State). The abstract cache state of a program line l within a UP and a function instance is the set of program lines that can potentially be cached prior to the execution of l within the UP and the function instance.

The notion of an abstract cache state is a compromise between a feasible storage complexity of the static cache simulation and the alternative of an exhaustive set of all cache states that may occur at execution time with an exponential storage complexity.

Based on the abstract cache state, it becomes possible to statically predict the caching behavior of each instruction of a program. Instructions may be categorized as *always-hit*, *always-miss*, *first-miss*, or *conflict*. The semantics for each category is as follows. Always-hit (always-miss) instructions will always result in a cache hit (miss) during program execution. First-miss instructions will result in a cache miss on the first reference to the instruction and in a cache hit for any consecutive references. Conflict instructions may result in a cache hit or a cache miss during program execution, *i.e.* their behavior cannot be predicted statically through this simulation method. The different categories are defined below after introducing the notion of a reaching state.

Definition 8 (Reaching State). The reaching state of a UP within a function instance is the set of program lines that can be reached through control-flow transitions from the UP of the function instance.

Definition 9 (Instruction Categorization). Let i_k be an instruction within a UP and a function instance. Let $l = i_0..i_{n-1}$ be the program line containing i_k and let i_{first} be the first instruction of l within the UP. Let s be the abstract cache state for l within the UP. Let l map into cache line c, denoted by $l \rightarrow c$. Let t be the reaching state for the UP. Then, the instruction categorization is defined as

$$\text{category } (i_k) = \begin{cases} \text{always-miss} & \text{if } k = first \wedge l \notin s \\ \text{always-hit} & \text{if } k \neq first \vee (l \in s \wedge \underset{m \rightarrow c, m \neq l}{\forall} m \notin s) \\ \text{first-miss} & \text{if } k = first \wedge l \in s \wedge \underset{m \rightarrow c, m \neq l}{\exists} m \in s \wedge \underset{m \rightarrow c, m \neq l}{\forall} m \in s \Rightarrow m \notin t \\ & \quad \wedge \underset{0 \leq x < n}{\forall} category(i_x) \in \{\text{always-hit, first-miss}\} \\ \text{conflict} & \text{otherwise} \end{cases}$$

An always miss occurs when instruction i_k is the first instruction encountered in program line l and l is not in the abstract cache state s. An always hit occurs either if i_k is not the first instruction in l or l is the only program line in s

mapping into c. A first miss occurs if the following conditions are met. First, i_k is first in l and if l and at least one other program line m (which maps into c) are in s. Second, if one such line m is in s, then the line must not be reachable anymore from the current UP. Third, all other instructions in the program line have to be either always hits or first misses. A conflict occurs in all other cases.

This categorization results in some interesting properties. If the size of the program does not exceed the size of the cache, hardly any instructions will be categorized as conflicts. Thus, the cache behavior can mostly be statically predicted.[3] As the program becomes much larger than the cache, the number of conflicts increases to a certain point. This point depends on the ratio between program size and cache size. After this point, conflicts start to decrease again while first misses increase.

3.2 Calculation of Abstract Cache States

Algorithm 2 (Calculation of Abstract Cache States) .
Input: Function-Instance Graph of the program and UPPA for each function.
Output: Abstract Cache State for each UP.
Algorithm: Let `conf_lines(UP)` be the set of program lines (excluding the program lines of UP), which map into the same cache line as any program line within the UP.

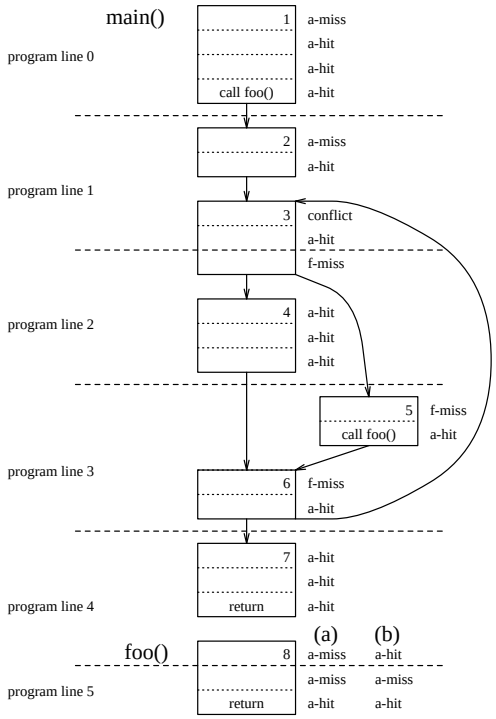
```
input_state(main):= all invalid lines;
WHILE any change DO
    FOR each instance of a UP in the program DO
        input_state(UP):= φ;
        FOR each immediate predecessor P of UP DO
            input_state(UP):= input_state(UP) ∪ output_state(P);
        output_state(UP):= [input_state(UP) ∪ prog_lines(UP)] \ conf_lines(UP)
```

The iterative Algorithm 2 calculates the abstract cache states. In the algorithm, the abstract cache state of the program line of a UP that is referenced first is referred to as `input_state`. Conversely, the abstract cache state after the program line of a UP that is referenced last is referred to as `output_state`. The set of vertices (basic blocks) in a UP provides the scope of program lines to transform an input_state into an output_state.

The algorithm is a variation of an iterative data-flow analysis algorithm commonly used in optimizing compilers. Thus, the time overhead of the algorithm is comparable to that of data-flow analysis and the space overhead is $O(pl * UPs * fi)$, where pl is the number of program lines, UPs is the number of paths, and fi the number of function instances. The correctness of the algorithm for data-flow analysis is discussed in [1]. The calculation can be performed for an arbitrary control-flow graph, even if it is irreducible. In addition, the order of processing basic blocks is irrelevant for the correctness of the algorithm. The reaching states can be calculated using the same base algorithm with `input_state(main) = conf_lines(UP) =` ϕ.

[3] The adaptations of the definition for different applications (described in [8]) provide static predictability of all instructions if the program fits into cache, *i.e.* no instruction will be categorized as a conflict in this case. Since the adaptation depends on the application it could not be incorporated in the original definition in this paper.

Example 5. Figure 4 depicts the calculation of input and output states. The chosen UPPA is $UPPA_b$, the basic block partitioning. Algorithm 2 operates on any UPPA, and the categorization is not influenced by the choice of a UPPA. The $UPPA_b$ simplifies the example but would result in more measurement overhead during on-the-fly analysis than a small UPPA constructed by Algorithm 1.




```
input(1 )=[I I I I                ]
input(8a)=[  I I I 0              ]
input(2 )=[    I I              4 5]
input(3 )=[    I I   1 2 3 4 5]
input(4 )=[      I   1 2 3 4   ]
input(5 )=[      I   1 2 3 4   ]
input(8b)=[          1 2 3 4   ]
input(6 )=[      I   1 2 3 4 5]
input(7 )=[          1 2 3 4 5]

output(1 )=[  I I I 0             ]
output(8a)=[    I I             4 5]
output(2 )=[    I I   1         4  ]
output(3 )=[      I   1 2 3 4   ]
output(4 )=[      I   1 2 3 4   ]
output(5 )=[          1 2 3 4   ]
output(8b)=[            2 3 4 5]
output(6 )=[          1 2 3 4 5]
output(7 )=[          1 2 3 4 5]

reach(1 )=[  1 2 3 4 5]
reach(8a)=[  1 2 3 4 5]
reach(2 )=[  1 2 3 4 5]
reach(3 )=[  1 2 3 4 5]
reach(4 )=[  1 2 3 4 5]
reach(5 )=[  1 2 3 4 5]
reach(8b)=[  1 2 3 4 5]
reach(6 )=[  1 2 3 4 5]
reach(7 )=[           ]
```

Fig. 4. Instruction Categorization in Flow Graph

In the example, there are 4 cache lines and the line size is 16 bytes (4 instructions). Thus, program line 0 and 4 map into cache line 0, program line 1 and 5 map into cache line 1, program line 2 maps into cache line 2, and program line 3 maps into cache line 3. The immediate successor of a block with a call is the first block in that instance of the called function. Block 8a corresponds to the first instance of foo() called from block 1 and block 8b corresponds to the second instance of foo() called from block 5.

After determining the input states of all blocks, each instruction is categorized based on its abstract cache state (derived from the input state) and the reaching state shown in the figure. By inspecting the input states of each block, one can make some observations that may not have been detected by a naive inspection of only physically contiguous sequences of references. For instance, the static simulation determined that the first instruction in block 7 will always be in cache (always hit) due to spatial locality since program line 4 is in `input(7)` and no conflicting program line is in `input(7)`. It was also determined that the first instruction in basic block 8b will always be in cache (always hit) due to temporal locality. The static simulation determined that the last instruction in block 3 will not be in cache on its first reference, but will always be in cache on subsequent references (first miss). This is indicated by `input(3)`, which includes program line 2 but also a conflicting program line "invalid" for cache line 3. Yet, the conflicting program line cannot be reached. This is also true for the first instructions of block 5 and 6 though a miss will only occur on the first reference of either one of the instructions. This is termed a *group first miss*. Finally, the first instruction in block 3 is classified as a conflict since it could either be a hit or a miss (due to the conditional call to foo). This is indicated by `input(3)`, which includes program line 1 and a conflicting program line 5 that can still be reached.

The current implementation of the static simulator imposes the restriction that only direct-mapped cache configurations are allowed. Recent results show that direct-mapped caches have a faster access time for hits, which outweighs the benefit of a higher hit ratio in set-associative caches for large cache sizes [7].

4 Applications

The partitioning of the control-flow graph into unique paths and the construction of a function-instance graph can be used for on-the-fly analysis of program behavior in general. This section focuses on combining these techniques with static cache simulation for different applications. Our implementation modifies the back-end of an optimizing compiler to partition the control flow of functions. The static cache simulator uses the partitioning information to construct the function-instance graph and to perform the instruction categorization.

4.1 Fast Instruction Cache Analysis

Instruction cache analysis can be performed on-the-fly in an efficient manner simulating much of the caching behavior statically. The simulation is performed for the function-instance graph of the program and a small UPPA for each function. Thus, a small set of measurement points can be determined. The program can be instrumented with measurement code on a unique edge/vertex within all UPs. The measurement code consists of frequency counters for each UP and state transitions similar to a deterministic finite automaton to simulate "conflict" instructions whose caching behavior cannot be determined prior to execution time. This keeps the measurement overhead fairly low.

During program execution, counters record the frequency of each instruction. At program termination, the total number of cache hits and misses is determined by relating the frequency with the instructions of a UP and their categorization. First misses are initially counted as hits. Then, the set of first miss lines is correlated with the corresponding frequency counters and if any counter is greater than zero, the total hit count is reduced by one while the miss count is incremented by one. For group first misses (see discussion of Figure 4), only one first miss is counted as a miss.

The implementation of the described method showed that the execution time of the instrumented program takes about twice the time of the uninstrumented program [9]. Other methods to perform on-the-fly instruction cache analysis result in an overhead of 6 to 16 at best [12].

4.2 Analytical Bounding of Execution Time

In real-time systems, the scheduling analysis of a task set is based on the assumption that the worst-case execution time (WET) be known. Timing tools have been developed to statically analyze programs for a given target processor and predict their WET. Until now, the presence of cache memories was widely regarded as a source of unpredictability which prevents a tight prediction of the WET. Consequently, caches are often disabled in hard real-time systems.

Static cache simulation provides a method to predict the caching behavior of a large percentage of instructions prior to program execution. By making the instruction categorization available to a timing tool, the WET can be predicted much more tightly. Thus, real-time applications can finally enable caches to achieve better performance without sacrificing predictability.

The instruction categorization is refined to meet the needs of a timing tool. Instructions are categorized separately for each function instance level and for each loop level. Several refinements for conflict instructions ensure that the WET estimation remains relatively tight. An exhaustive discussion would be beyond the scope of this paper and can be found elsewhere [2]. Other aspects of predicting instruction caching are discussed in [10, 8].

5 Conclusion

In this paper, a formal method is developed to perform efficient on-the-fly analysis of program behavior with regard to path partitioning. The method partitions the control-flow graph into a small set of unique paths, each of which contain a unique edge or vertex where instrumentation code can be placed. Furthermore, the construction of the function-instance graph from a program's call graph refines the analysis. Performance evaluations show that the number of measurement points can be reduced by one third using these methods. Taking advantage of path partitionings and the function-instance graph, the method of static cache simulation is introduced which allows the prediction of a large number of cache references prior to program execution. A number of applications for this method are discussed, ranging from faster instruction cache analysis to the analytical bounding of execution time by static analysis for real-time tasks.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, December 1994.
3. T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.
4. M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.
5. A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *International Symposium on Computer Architecture*, pages 270–279, May 1990.
6. G. Chartrand and L. Lesniak. *Graphs & Digraphs*. Wadsworth & Brooks, 2nd edition, 1986.
7. M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.
8. F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.
9. F. Mueller and D. B. Whalley. Fast instruction cache analysis via static cache simulation. TR 94-042, Dept. of CS, Florida State University, April 1994.
10. F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.
11. C. Stunkel and W. Fuchs. Trapeds: Producing traces for multicomputers via execution driven simulation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.
12. D. B. Whalley. Techniques for fast instruction cache performance evaluation. *Software Practice & Experience*, 19(1):195–203, January 1993.