Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling

Glenn Ammons
Dept. of Computer Sciences
University of Wisconsin-Madison
ammons@cs.wisc.edu

Thomas Ball
Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

James R. Larus*
Dept. of Computer Sciences
University of Wisconsin-Madison
larus@cs.wisc.edu

Abstract

A program profile attributes run-time costs to portions of a program's execution. Most profiling systems suffer from two major deficiencies: first, they only apportion simple metrics, such as execution frequency or elapsed time to static, syntactic units, such as procedures or statements; second, they aggressively reduce the volume of information collected and reported, although aggregation can hide striking differences in program behavior.

This paper addresses both concerns by exploiting the hardware counters available in most modern processors and by incorporating two concepts from data flow analysis–flow and context sensitivity–to report more context for measurements. This paper extends our previous work on efficient path profiling to flow sensitive profiling, which associates hardware performance metrics with a path through a procedure. In addition, it describes a data structure, the calling context tree, that efficiently captures calling contexts for procedure-level measurements.

Our measurements show that the SPEC95 benchmarks execute a small number (3–28) of hot paths that account for 9–98% of their L1 data cache misses. Moreover, these hot paths are concentrated in a few routines, which have complex dynamic behavior.

1 Introduction

A program profile attributes run-time costs to portions of a program's execution. Profiles can direct a programmer's attention to algorithmic bottlenecks or inefficient code [Knu71], and can focus compiler optimizations on the parts of a program that offer the largest potential for improvement [CMH91]. Profiles also provide a compact summary of a program's execution, which forms the basis for program coverage testing and other software engineering tasks [WHH80, RBDL97]. Although program profiling is widely used, most tools report only rudimentary profiles that apportion execution frequency or time to static, syntactic units, such as procedures or statements.

This paper extends profiling techniques in two new directions. The first exploits the hardware performance counters becoming available in modern processors, such as Intel's Pentium Pro, Sun's UltraSPARC, and MIPS's R10000. These processors have complex microarchitectures that dynamically schedule instructions. These machines are difficult to understand and model accurately. Fortunately, these processors also contain readily accessible hardware counters that record a wide range of events. For example, Sun's UltraSPARC processors [Sun96] count events such as instructions executed, cycles executed, instruction stalls of various types, and cache misses (collectively, *hardware performance metrics*). Existing profiling systems do not exploit these counters and are limited to a few, simple metrics, such as instruction execution frequency [BL94], time in a procedure [GKM83, Sof93], or cache misses [LW94].

Our second extension increases the usefulness of program profiles by reporting a richer context for measurements, by applying two concepts from static program analysis—flow and context sensitivity. A flow sensitive profile associates a performance metric with an acyclic path through a procedure. A context sensitive profile associates a metric with a path through a call graph. Most profiling systems are flow and context insensitive and attribute costs only to syntactic program components such as statements, loops, or procedures.

Flow sensitive profiling enables programmers both to find hot spots and to identify dependencies between them. For example, a flow insensitive measurement might find two statements in a procedure that have high cache miss rates, whereas a flow sensitive measurement could show that the misses occur when the statements execute along a common path, and thus are possibly due to a cache conflict.

Context sensitive profiling associates a metric with a sequence of procedures (a *calling context*) that are active during intervals of a program's execution. Labeling a metric with its calling context can separate measurements from different invocations of a procedure. Without a calling context, profiling tools can only approximate a program's context-dependent behavior. For example, profiling systems such as

*This research supported by: Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550; NSF NYI Award CCR-9357779, with support from Hewlett Packard and Sun Microsystems; and NSF Grant MIP-9625558.

To appear at the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, Nevada, 15-18 June, 1997.

gprof [GKM83] or *qpt* [BL94] apportion the cost of a procedure to its callers in proportion to the relative frequency of calls between each pair of procedures, although this calculation can produce misleading results [PF88].

Moreover, context is essential to elucidate hardware measurements. Many hardware events (such as instruction stalls and cache misses) are affected by execution relationships among program components. Performance tools that associate a metric with an isolated component of a program overlook these relationships. Paths capture temporal relationships, by reporting the sequence of statements leading up to the behavior of interest. In addition, our measurements found that one important metric, data cache misses, are heavily concentrated along a small number of paths in a few routines that have complex dynamic behavior. Tools that report cache misses at the procedure or statement level cannot isolate these hot paths.

1.1 Contributions

This paper describes how to instrument programs to record hardware performance metrics efficiently in a flow sensitive and/or context sensitive manner. Our contributions are three-fold:

- Flow sensitive profiling extends our technique of path profiling, which previously recorded only the execution frequency of paths in a procedure's control flow graph [BL96]. This paper generalizes path profiling by associating hardware performance metrics with paths.
- Context sensitive profiling provides a calling context for flow sensitive (or other) procedure-level profiles. It uses a run-time data structure, called the *calling context tree (CCT)*, to label an arbitrary metric or set of metrics with its dynamic calling context. The CCT captures a program's calling behavior more precisely than a call graph, but its size is bounded, unlike a complete dynamic call tree. Our techniques for constructing a CCT are general and handle indirect function calls, recursion, and non-local returns.
- These two techniques may be combined, by using a CCT to record the calling context for paths within a procedure. This combination provides an efficient approximation to interprocedural path profiling.

1.2 Measurements

Using the Executable Editing Library (EEL) [LS95], we built a tool called PP (Path Profiler) that instruments program executables to record flow sensitive and context sensitive profiles. PP records not only instruction frequency, but also fine-grain timing and event count information by accessing the hardware counters on UltraSPARC processors. The run-time overhead of flow sensitive and context sensitive profiling for the SPEC95 benchmarks is 60–80%, on average. Furthermore, the results show that CCTs are quite compact in practice.

Our measurements of the SPEC95 benchmarks show that these programs contain a small number of *hot* paths, which each incur at least 1% of the L1 data cache misses. Collective, 3–28 of these paths account for 59–98% of the misses in programs other than 099.go and 126.gcc. These two programs execute many more paths, but lowering the hot path threshold to 0.1% of the L1 misses finds that approximately 1% of the paths (172 and 139, respectively) again account for 42 and 56% of the misses. Most hot paths have above average miss rates. Moreover, hot paths are concentrated in a small number of routines (1–24), which incur most cache misses (44–99%) and execute roughly ten times as many paths as the cold routines.

1.3 Overview

The paper is organized as follows. Section 2 summarizes our result on intraprocedural path profiling and Section 3 shows how it can be generalized to perform flow sensitive profiling of hardware counters. Section 4 describes the CCT data structure, how it is built, and how it can be used to record context sensitive profiles. Section 5 describes the implementation of PP, which uses both techniques to record a variety of hardware-specific metrics on Sun UltraSPARC processors. Section 6 presents experimental measurements that show that the overhead and perturbation of this technique are reasonable, and examines the hot path phenomenon. Section 7 describes related work.

2 Efficient Path Profiling

This section summarizes our previous work on intraprocedural path profiling [BL96], which this work extends. Given a procedure's control flow graph (CFG), our path profiling algorithm:

- Assigns an integer label to every edge in an acyclic CFG such that the sum of integers is unique along each unique path from the entry to exit of a procedure. This labelling is also compact, as path sums fall in the range $0 \ldots n-1$, where n is the number of potential paths from entry to exit.
- Inserts simple instrumentation to track the path sum in an integer register at run-time. The path sum can directly index an array of counters or be used as a key into a hash table of counters (if the number of potential paths is large).
- Transforms a CFG containing cycles (loops), which contain an unbounded number of potential paths, into an acyclic graph with a bounded number of paths. The algorithm can handle reducible and irreducible CFGs.

Figure 1(a) illustrates the technique in a simple graph containing six unique paths from A to F. Each path has a unique path sum, as shown in Figure 1(b). Figure 1(c) shows a simple instrumentation scheme for tracking the path sum in a register r, while Figure 1(d) shows an optimized instrumentation scheme (see [BL96] for details).

To instrument a CFG for efficient path profiling, it must have a unique entry vertex $ENTRY$ from which all vertices are reachable and a unique exit vertex $EXIT$ that is reachable from all vertices. The algorithm has a straightforward extension for CFGs that do not meet this requirement.

The description below breaks the algorithm in two parts: instrumentation of acyclic CFGs and a method of transforming a CFG containing cycles into an acyclic CFG.

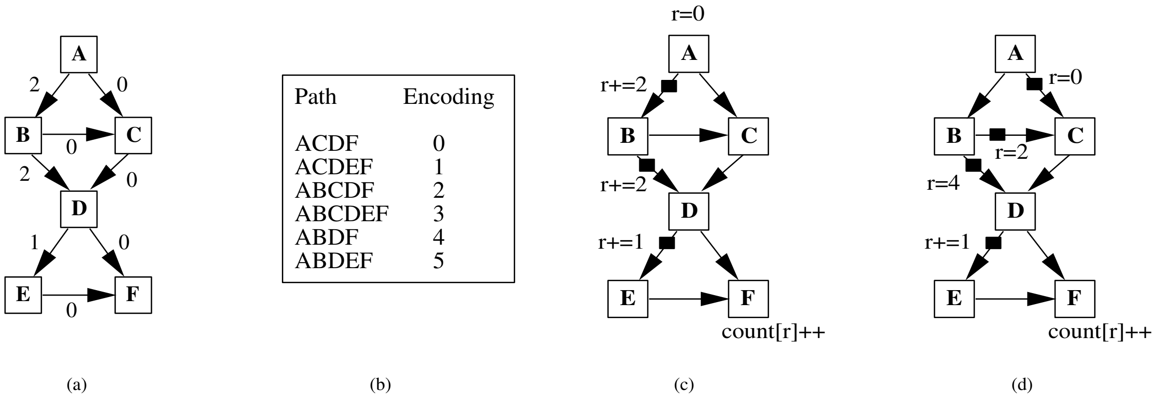

Figure 1: Path profiling edge labelling and instrumentation. (a) An integer labelling with unique path sums; (b) The six paths and their path sums; (c) Simple instrumentation for tracking the path sums; (d) Optimized instrumentation for tracking the path sums.

2.1 Path Profiling of Acyclic CFGs

Conceptually, the algorithm makes two linear-time traversals of the CFG, visiting vertices in reverse topological order in each pass. The two passes easily can be combined into a single linear-time pass.

In the first pass, each vertex v is labelled with an integer $NP(v)$, which denotes the number of paths from v to $EXIT$. These values are well-defined because the graph is acyclic. As a base case, $NP(EXIT) = 1$. If v has successors $w_1 ... w_n$ (which are already labelled, since vertices are visited in reverse topological order), $NP(v) = NP(w_1) + ... + NP(w_n)$.

The second pass labels each edge e of the CFG with an integer $Val(e)$ such that:

- Every path from $ENTRY$ to $EXIT$ generates a path sum in the range $0 ... NP(ENTRY) - 1$.
- Every value in the range $0 ... NP(ENTRY) - 1$ is the path sum of some path from $ENTRY$ to $EXIT$.

No processing is required for the $EXIT$ vertex in this pass, as it has no outgoing edges. Consider vertex v with successors $w_1 ... w_n$. Since the algorithm traverses the CFG in reverse topological order, we may assume that for each w_i, all the edges reachable from w_i have been labelled so that each path from w_i to $EXIT$ generates a unique path sum in the range $0 ... NP(w_i) - 1$. Given a vertex v, let v's successors be totally ordered $w_1 ... w_n$ (the order chosen is immaterial). The value associated with edge $e_i = v \rightarrow w_i$ is simply the sum of the number of paths to $EXIT$ from all successors $w_1 ... w_{i-1}$:

$$Val(e_i) = \sum_{j=1}^{i-1} NP(w_j).$$

Figure 2 illustrates the labelling process for a vertex v with three successors, $w_1 ... w_3$. The paths from each w_i to $EXIT$ generate path sums in the range $0 ... NP(w_i) - 1$. The label on edge $v \rightarrow w_1$ is 0, so paths from v to $EXIT$ beginning with this edge will generate path sums in the range $0 ... NP(w_1) - 1$. The label on edge $v \rightarrow w_2$ is $NP(w_1)$, so paths from v to $EXIT$ beginning with this edge will generate path sums in the range $NP(w_1) ... NP(w_1) + NP(w_2) - 1$, and so on. Path sums for paths from v to $EXIT$ therefore lie in the range $0 ... NP(w_1) + NP(w_2) + NP(w_3) - 1$.

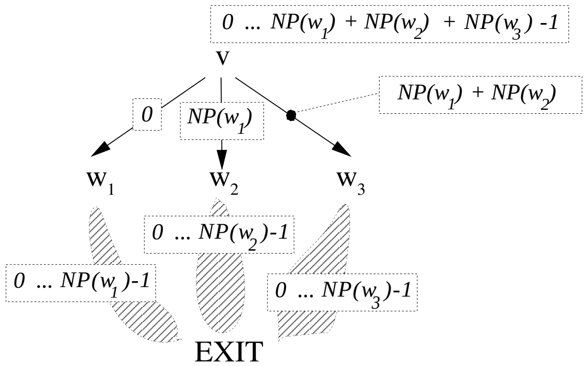

Figure 2: Edge labelling phase.

Given the edge labelling, the instrumentation phase is very simple. An integer register r tracks the path sum and is initialized to 0 at the $ENTRY$ vertex. Along an edge e with a non-zero value, r is incremented by $Val(e)$. At the $EXIT$ vertex, r indexes an array to update a count (`count[r]++`) or serves as a hash key into a hash table of counters. For details on how to greatly reduce the number of points at which r must be incremented, see [BL96, Bal94].

2.2 Path Profiling of Cyclic CFGs

Cycles in a CFG introduce an unbounded number of paths. Every cycle contains a backedge, as identified by a depth-first search from $ENTRY$. The algorithm only profiles paths that have no backedges or in which backedges occur as the first and/or last edge of the path. As a result, paths fall into four categories:

- A backedge-free path from $ENTRY$ to $EXIT$;
- A backedge-free path from $ENTRY$ to v, followed by backedge $v \rightarrow w$;
- After execution of backedge $v \rightarrow w$, a backedge-free path from w to x, followed by backedge $x \rightarrow y$;
- After executing backedge $v \rightarrow w$, a backedge-free path from w to $EXIT$.

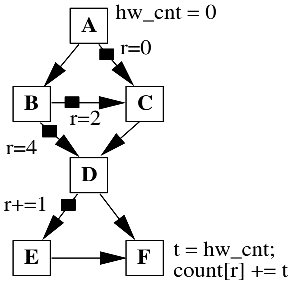

Figure 3: Instrumentation for measuring a metric over a paths. On out-of-order processors, such as the UltraSPARC, is is necessary to read the hardware counter after writing it, to ensure that the write has completed.

The algorithm of the previous section only guarantees uniqueness of path sums for paths starting at a single "entry" point. Paths with different starting points, such as the latter three categories, may compute identical path sums.

However, a simple graph transformation turns a cyclic CFG into an acyclic CFG and extends the uniqueness and compactness properties of path sums to all paths described above, regardless of whether or not they are in the same category.

For each backedge $b = v \to w$ in the CFG, the transformation removes b from the CFG and adds two pseudo edges in its place: $b_{start} = ENTRY \to w$ and $b_{end} = v \to EXIT$. The resulting graph is acyclic and contains a bounded number of paths. The pseudo edges represents paths that start and/or end with a backedge. For example, a path from $ENTRY$ to vertex v, followed by backedge $b = v \to w$ in the original CFG translates directly to a path starting at $ENTRY$ and ending with pseudo edge $b_{end} = v \to EXIT$.

The path algorithm for acyclic CFGs runs on the transformed graph, labelling both original edges and pseudo edges. This labelling guarantees that all paths from $ENTRY$ to $EXIT$ have unique paths sums, including paths containing pseudo edges. After this labelling, instrumentation is inserted as before, with one exception. Since the pseudo edges do not represent actual transfers of control, they cannot be instrumented. However, their values are incorporated into the instrumentation along a backedge:

`count[r+`END`]++; r=`$START$,

where END is the value of pseudo edge b_{end} and $START$ is the value of pseudo edge b_{start}.

3 Flow Sensitive Profiling of Hardware Metrics

This section shows how to extend the path profiling algorithm to accumulate metrics other than execution frequency. These hardware metrics may record execution time, cache misses, instruction stalls, etc. Although this discussion focuses on the UltraSPARC's hardware counters [Sun96], similar considerations apply to other processors.

3.1 Tracking Metrics Along a Path

Associating a hardware metric with a path is straightforward, as shown in Figure 3. At the beginning of a path, set the hardware counter to zero. At the end of the path, read the hardware counter and add its value into an accumulator associated with the path. Since paths are intraprocedural, hardware counters must be saved and restored at procedure calls. A system can either save the counter before a call and restore it after the call, or save the counter on procedure entry and restore it before procedure exit. Our implementation uses the latter approach to reduce code size and to capture the cost of call instructions.

The UltraSPARC provides an instruction that reads its two 32 bit hardware counters into one 64 bit general purpose register. Several additional instructions are necessary to extract the two values from the register. In total, our instrumentation requires thirteen or more instructions to increment two accumulators and a frequency metric for a path.

Both counters can be zeroed by writing a 64 bit register to the two hardware counters. However, because of the UltraSPARC's superscalar architecture, it is necessary to read the counters immediately after writing them, to ensure that the write completes before subsequent instructions execute.[1]

3.2 Measurement Perturbation

An important problem with hardware counters is the perturbation caused by the instrumentation. As a simple example, consider using hardware counters to record instruction frequency. Each instrumentation instruction executed along a path is counted. Figure 3 shows an example, in which the profiling instrumentation introduces two instructions along path $A \to B \to D \to F$—one instruction for the read after the write to the hardware counters, and one instruction along $B \to D$.

Moreover, instrumenting an executable introduces further perturbation. For example, path profiling requires a free local register in each procedure. If a procedure has no free registers, EEL spills a register to the stack, which requires additional loads and stores around instructions that originally used the register. These instructions affect a metric. In addition, EEL's layout of the edited code can introduce new branches.

For simple, predictable metrics, such as instruction frequency, a profiling tool can correct for perturbation by using path frequency to subtract the effect of instrumentation code. For other metrics, however, it is very difficult to estimate perturbation effects. For example, if the metric is processor stalls or cache misses, separating instrumentation from the underlying behavior appears intractable. Moreover, instrumentation that executes at the beginning or end of a path, outside the measured interval, can also cause cache conflicts that increase a program's cache misses. We do not have a general solution for this difficult problem, which has been explored by others [MRW92]. Section 6 contains measurements of the perturbation.

3.3 Overflow

The UltraSPARC's hardware counters are only 32 bits wide. A metric, such as cycle counts, can cause a counter to wrap in a few seconds. However, our intraprocedural instrumentation measures call-free paths, and even 32 bit counters

[1] Ashok Singhal, Sun Microsystems. Personal communication. Oct. 1996.

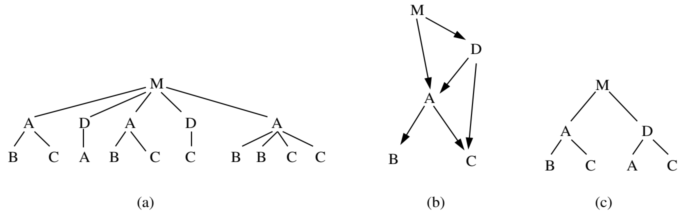

Figure 4: (a) A dynamic call tree, (b) its corresponding call graph, and (c) its corresponding calling context tree.

will not wrap on any conceivable path. Moreover, we accumulated metrics as 64 bit quantities, which will handle any program.

4 Context Sensitive Profiling

This section describes the calling context tree (CCT), a data structure that compactly represents calling contexts, presents an algorithm for constructing a CCT, and shows how a CCT can capture flow sensitive profiling results and other performance metrics.

4.1 The Calling Context Tree

The CCT offers an efficient intermediate point in the spectrum of run-time representations of calling behavior. The most precise but space-inefficient data structure, as shown in Figure 4(a), is the *dynamic call tree (DCT)*. Each tree vertex represents a single procedure activation and can precisely record metrics for that invocation. Tree edges represent calls between individual procedure activations. The size of a DCT is proportional to the number of calls in an execution.

At the other end of the spectrum, a *dynamic call graph (DCG)* (Figure 4(b)), compactly represents calling behavior, but at a great loss in precision. A graph vertex represents all activations of a procedure. A DCG has an edge $X \rightarrow Y$ iff there is an edge $X \rightarrow Y$ in the program's dynamic call tree. Although the DCG's size is bounded by the size of the program, each vertex accumulates metrics for (perhaps unboundedly) many activations. This accumulation leads to the "gprof problem," in which the metric recorded at a procedure C cannot be accurately attributed to C's callers. Furthermore, a call graph can contain "infeasible" paths, such as $M \rightarrow D \rightarrow A \rightarrow C$, which did not occur during the program's execution.

The *calling context tree (CCT)* compactly represents all calling contexts in the original tree, as shown in Figure 4(c). It is defined by the following equivalence relation on a pair of vertices in a DCT. Vertices v and w in a DCT are *equivalent* if:

- v and w represent the same procedure, and
- the tree parent of v is equivalent to the tree parent of w, or $v = w$.

The equivalence classes of vertices in a DCT define the vertex set of a CCT. Let $Eq(x)$ denote the equivalent class of vertex x. There is an edge $Eq(v) \rightarrow Eq(w)$ in the CCT iff there is an edge $v \rightarrow w$ in the DCT.

Figure 4(c) shows that the CCT preserves the two unique calling contexts associated with procedure C ($M \rightarrow A \rightarrow C$ and $M \rightarrow D \rightarrow C$). A CCT contains a unique vertex for each unique path (call chain) in its underlying DCT. Stated another way, a CCT is a projection of a dynamic call tree that discards redundant contextual information while preserving unique contexts. Metrics from identical contexts will be aggregated, again trading precision for space. A CCT can accurately record a metric along different call paths—thereby solving the "gprof problem". Another important property of CCTs is that the sets of paths in the DCT and CCT are identical.

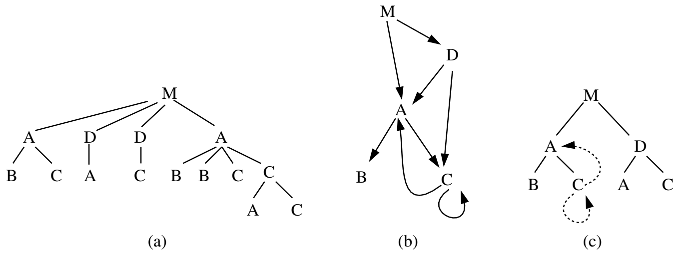

Figure 5: (a) A dynamic call tree containing recursive calls, (b) its corresponding call graph, and (c) its corresponding calling context tree.

As defined above, the out-degree of a vertex in the CCT is bounded by the number of unique procedures that may be called by the associated procedure. Thus, the breadth of the CCT is bounded by the number of procedures in a program.[2] In the absence of recursion, the depth of a CCT also is bounded by the number of procedures in a program. However, with recursion, the depth of a CCT may be unbounded. To bound the depth of the CCT, we redefine vertex equivalence. Vertices v and w in a DCT are *equivalent* if:

- v and w represent the same procedure, and
- the tree parent of v is equivalent to the tree parent of w, or $v = w$, or there is a vertex u such that u represents the same procedure as v and w and u is an ancestor of both v and w.[3]

The modified second condition ensures that all occurrences of a procedure P below and including an instance of P in the DCT are equivalent. As a result of this new equivalence relation, the depth of a CCT never exceeds the number of procedures in a program since each procedure occurs at most once in any path from the root to a leaf of the CCT.

The new equivalence relation introduces backedges into the CCT, so it is not strictly a tree. However, CCTs never include cross-edges or forward edges. This implies that tree edges are uniquely defined and separable from backedges. Unfortunately, backedges (which arise solely due to recursion) destroy the context-uniqueness property of a CCT with respect to a DCT.

Figure 5(a) shows a dynamic call tree with recursive calling behavior and its corresponding call graph and CCT. The recursive invocation of procedure A and its initial invocation are represented by the same vertex in the CCT.

A space-precision trade-off in a CCT is whether to distinguish calls to the same procedure from different call sites in a calling procedure. Distinguishing call sites requires more

[2] The breadth of a DCT is unbounded due to loop iteration.

[3] A vertex is an ancestor of itself.

```
struct List;
struct CallRecord;
typedef union {List* le;
               CallRecord* cr; int offset} Callee;
typedef union {CallRecord* cr; int offset} ListElem;

struct CallRecord {
   int        ID;           // procedure identifier
   CallRecord *parent;      // tree parent
   int        metrics[NM];  // the metrics
   Callee     children[];   // the callees
};

struct List {               // list of dynamic callees
   ListElem pr;
   List   *next;
};
```

Figure 6: CCT data structures.

space, but is useful for path profiling, as distinct intraprocedural paths reach the different call sites. Other applications of a CCT may aggregate a metric for a pair of procedures, and would not benefit from this increased precision. The approach described below assumes that call sites are distinguished; changes to combine them are minor.

4.2 Constructing a Calling Context Tree

A CCT can be constructed during a program's execution, as we now describe. Each vertex of a CCT is referred to as a "call record," although different run-time activation records may share the same call record. Let CR be the call record associated with the currently active procedure C. The principle behind building a CCT is simple: if D has just been called by C and is already represented by a record that is one of CR's children, use the existing call record. Otherwise, look for a call record for D in C's ancestors. If a record DR is found, the call to D is recursive, so create a pointer from CR to DR (a backedge), and make DR the current call record. If D is not an ancestor of C, create a new call record DR for D and make it a child of CR.

The first time that a new callee is encountered, we pay the cost of traversing the parent pointers. However, at subsequent calls, a callee immediately finds its call record, regardless of whether the call is recursive.

The C structures in Figure 6 define the basic data types in a CCT. A `CallRecord` contains two scalar fields and two arrays:

- `ID` is an `int` that identifies the procedure or function associated with the `CallRecord`. We currently use a procedure's starting address its identifier, although more compact encodings are possible.

- `parent` is a pointer to the `CallRecord`'s tree parent. This pointer is `NULL` if the `CallRecord` represents the root of the CCT.

- `metrics` is an array of counters for recording metrics. The size of this array is usually fixed for all `CallRecords`. When combining path profiling with call graph profiling, the size of the array depends on the number of paths in a procedure.

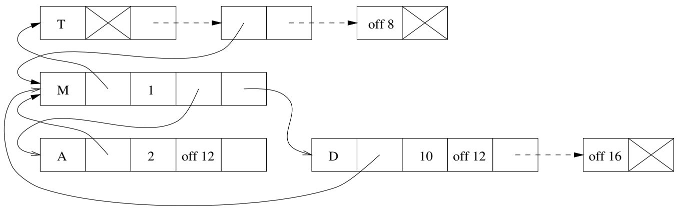

Figure 7: A representation of the first two levels of the CCT from Figure 4. Dashed lines indicate pointers to lists and solid lines indicate pointers to call records. For simplicity, each cell is assumed to be four bytes wide.

- `children` is an array of *callee slots*, one for each call site in the procedure. Each slot may be a pointer to a list of children (for indirect calls), a pointer directly to one child, or its uninitialized value—an offset to the start of the call record. The low-order two bits of a slot form a tag that discriminates among the three possible values.

Figure 7 illustrates the CCT data structure for the first two levels of the CCT in Figure 4, which includes the procedures M, A and D. Each call record in the structure begins with the corresponding procedure's `ID`. The root of the CCT is labeled with the special identifier `T`, which corresponds to no procedure. The distinguished node is useful in two ways. First, our executable editing tool cannot insert the full instrumentation in `_start`, which is a program's entry point in UNIX. Second, if we extended our tool to handle signals, the CCT would need multiple roots, as signal handlers represent additional entry points into a program. The call record for the root does not accumulate metrics.

There are several ways to build a CCT. Our approach allows each procedure to be instrumented separately, without knowledge of its callers or callees. Each procedure creates and initializes its own call records. When a procedure executes a call, the caller passes a pointer to the slot in its call record in which the callee should insert its call record. Our algorithm inserts instrumentation at the following points in a program in order to build a CCT:

Start of program execution. The initialization code allocates a heap for the CCT in a memory-mapped region. The region is demand paged, so physical memory is not allocated until it is actually used. The root call record is allocated and its callee slot is initialized to a list of one element, the offset back to the beginning of the call record. Finally, a SPARC global register, called the callee slot pointer (`gCSP`), is set to the root's call record. Callers use this register to pass a pointer to a callee slot in the record down to the callee.

Procedure entry. The instrumentation code first obtains a pointer to the procedure's call record. The code loads the value pointed to by the `gCSP` (the procedure's callee slot). The low-order 2 bits of this value are a tag. If the tag is 0, the value is a pointer to a call record for this procedure.

If the tag is 1, this procedure has not been called from this context. Masking off the low-order two bits yields an offset from the beginning of the current call record (i.e., the caller's call record). The code then searches the `parent` pointers, looking for an ancestral instance of the callee.

If a record is found, this call is recursive and the old record is reused. If no record is found, the code allocates and initializes a new call record. Initialization sets the `ID` and `children` fields of the new record. Callee slots for direct calls are set to the tagged offset of the slot in the record. Slots for indirect calls are set to a tagged pointer to a list containing a single entry, the tagged offset of the beginning of the call record. In both cases, the code stores a pointer to the found or allocated record in the callee slot.

Finally, if the tag is 2, the callee slot contains a pointer to a list of callees. If this procedure has been called from this site before, its call record is in the list. The code moves its pointer to the front of the list, so it can be found more quickly next time. If there is no such pointer in the list, the list's last element is the tagged offset to the caller's call record. This provides enough information to follow the parent pointers and find or create a call record, as before.

The callee's call record becomes the new current call record, which is held in a local register call `lCRP`. The instrumentation also saves the `gCSP` to the stack. If an exception or signal transfers control to instrumented code, the normal exception mechanisms restore the `lCRP`. If instead control transfers to uninstrumented code, there is no `lCRP` to restore. Thus, exceptions to instrumented code are handled transparently, but the exception handling mechanism must be changed to handle exceptions to uninstrumented code. Signals, which have resumption semantics, do not have this problem.

Procedure exit. The instrumentation restores the old `gCSP` from the stack. If an instrumented procedure A calls an uninstrumented procedure B, and B calls the instrumented procedures C and D, saving and restoring the `gCSP` ensures that C and D are correctly counted as children of A.

Procedure call. The instrumentation sets the `gCSP` to the sum of the `lCRP` and the offset to the callee slot for this call site.

Program exit. Immediately before the program terminates, the instrumentation writes the heap containing the CCT to a file from which the CCT can be reconstructed.

4.3 Associating Metrics with a CCT

The instrumentation code uses the `metric` fields in the current call record to capture metrics for the procedure invocation. The simplest metric is execution frequency, which just increments a counter in the `CallRecord`. To combine intraprocedural path profiles with calling context simply requires a minor change to keep a procedure's array of counters or hash table (Section 2) in a `CallRecord`.

Recording hardware metrics is slightly trickier. The least expensive approach is to record a hardware counter upon entry to a procedure, and to compute and accumulate the difference upon exit. This approach has two problems. First, it will not correctly measure functions that are not returned to in the conventional manner (i.e., due to longjmp's or exceptions). Second, the measured interval extends over the entire function invocation, which may cause 32 bit counters to wrap. To avoid these problems, the instrumentation also reads the hardware counters along loop backedges. Although this has higher instrumentation costs, it produces more information and mitigates or eliminates these two problems.

5 Implementation

We implemented these algorithms in a tool called PP, which instruments SPARC binary executables to profile intraprocedural paths and call paths. PP is built using EEL (Executable Editing Library), which is a C++ library that hides much of the complexity and system-specific detail of editing executables [LS95]. EEL provides abstractions that allow a tool to analyze and modify binary executables without being concerned with particular instruction sets, executable file formats, or the consequences of deleting existing code and adding foreign code (i.e., instrumentation).

5.1 UltraSPARC Hardware Counters

The Sun UltraSPARC processors [Sun96] implement sixteen hardware counters, which record events such as instructions executed, cycles executed, instruction stalls, and cache misses. The architecture also provides two program-accessible registers that can be mapped to two hardware counters, so a program can quickly read or set a counter, without operating system intervention. Solaris 2.5 currently does not save and restore these counters or registers on a context switch, so they measure all processes running on the processor. In our experiments, we minimized other activity by running a process locked onto a high-numbered processor of a SMP Server (low-number processors service interrupts). The 32 bit length of these counters is not a problem as PP records the hardware metrics along acyclic, intraprocedural paths.

6 Experimental Results

This section presents measurements of the SPEC95 benchmarks, made using PP. The benchmarks ran on a Sun Ultra-server E5000–12 167MHz UltraSPARC processors and 2GB of memory–running Solaris 2.5.1. C benchmarks were compiled with gcc (version 2.7.1) and Fortran benchmarks were compiled with Sun's f77 (version 3.0.1). Both compilers used only the -O option. In the runs, we used the *ref* input dataset. Elapsed time measurements are for the entire dataset. Hardware metrics are for the entire run, or the last input file in the case of programs, such as gcc, with multiple input files.

6.1 Run-Time Overhead

The overhead of intraprocedural path profiling is low (an average of 32% overhead on the SPEC95 benchmarks, roughly twice that of efficient edge profiling [BL94]). Details are reported elsewhere [BL96]. The extensions described in this paper increase the run-time cost of profiling, but the overheads remain reasonable. Table 1 reports the cost of three forms of profiling. Recording hardware metrics along intraprocedural paths (**Flow and HW**) incurs an average overhead of 80%. Recording hardware metrics along with call graph context (**Context and HW**) incurs an average overhead of 60%. Finally, recording path frequency (no hardware metrics) along with call graph context (**Context and Flow**) incurs an average overhead of 70%.

6.2 Perturbation

PP's instrumentation code can perturb hardware metrics. Table 2 reports a rough estimate of the perturbation for several metrics. The baseline in each case is the uninstrumented program, which we measured by sampling the UltraSPARC's hardware counters every six seconds (to avoid

Benchmark	Base Time (sec)	Flow and HW Time (sec)	Flow and HW Overhead (x base)	Context and HW Time (sec)	Context and HW Overhead (x base)	Context and Flow Time (sec)	Context and Flow Overhead (x base)
099.go	850.9	2517.0	3.0	1642.6	1.9	1949.2	2.3
124.m88ksim	551.8	1446.9	2.6	1347.6	2.4	1011.4	1.8
126.gcc	330.9	1461.5	4.4	1425.4	4.3	2984.5	9.0
129.compress	343.2	968.7	2.8	904.8	2.6	594.0	1.7
130.li	479.5	1169.5	2.4	1270.4	2.6	961.2	2.0
132.ijpeg	766.2	1494.0	1.9	1438.9	1.9	1130.7	1.5
134.perl	333.2	906.7	2.7	790.5	2.4	976.7	2.9
147.vortex	648.1	1545.1	2.4	1672.4	2.6	1938.7	3.0
CINT95 Avg	538.0	1438.7	2.7	1311.6	2.4	1443.3	2.7
101.tomcatv	505.9	670.1	1.3	586.7	1.2	652.2	1.3
102.swim	693.0	786.7	1.1	766.9	1.1	793.5	1.1
103.su2cor	468.5	587.4	1.3	561.0	1.2	553.5	1.2
104.hydro2d	795.6	1490.8	1.9	961.0	1.2	1147.1	1.4
107.mgrid	877.5	1088.6	1.2	1001.5	1.1	962.1	1.1
110.applu	710.1	1517.6	2.1	911.1	1.3	1293.3	1.8
125.turb3d	1063.1	1847.6	1.7	1750.1	1.6	1452.2	1.4
141.apsi	515.0	627.1	1.2	636.1	1.2	611.5	1.2
145.fpppp	2090.4	2172.2	1.0	1978.4	0.9	2310.5	1.1
146.wave5	635.6	815.8	1.3	741.6	1.2	773.9	1.2
CFP95 Avg	835.5	1160.4	1.4	989.4	1.2	1055.0	1.3
SPEC95 Avg	703.3	1284.1	1.8	1132.6	1.6	1227.6	1.7

Table 1: Overhead of profiling. **Base** reports the execution time of the uninstrumented benchmark. **Flow and HW** reports the cost of intraprocedural path profiling using hardware counters. **Context and HW** reports the cost of context sensitive profiling using hardware counters. **Context and Flow** reports the cost of context sensitive profiling using only frequency counts (no hardware counters).

	Cycles		Insts		DCache Read Misses		DCache Write Misses		ICache Miss Stalls		Mispredict Stalls		Store Buffer Stalls		FP Stalls	
Benchmark	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**	**F**	**C**
099.go	1.22	1.20	1.04	1.10	1.18	1.11	1.07	1.04	3.83	2.26	0.84	0.95	0.04	0.49	0.00	0.00
124.m88ksim	1.38	1.33	1.21	1.17	2.37	2.49	1.77	1.28	1.45	2.43	0.53	0.52	0.00	18.54	10.74	-
126.gcc	1.19	1.31	0.97	1.37	1.11	1.00	0.99	0.94	1.28	2.82	0.60	6.17	0.05	130.69	1.13	1442.89
129.compress	1.58	1.55	1.48	1.40	0.90	0.89	1.00	1.00	1.87	3.28	0.66	0.67	0.31	0.33	0.05	0.05
130.li	1.57	1.50	1.33	1.25	1.24	1.23	1.19	1.29	7.41	3.78	0.99	0.79	38.60	21.65	1.76	1.45
132.ijpeg	1.18	0.83	1.04	0.80	1.07	1.11	0.86	0.85	9.55	3.19	0.84	0.45	1.54	1.43	1.28	0.73
134.perl	1.77	1.49	1.52	1.18	1.77	2.04	1.62	1.77	2.26	1.67	1.10	1.19	0.27	0.18	6.22	3.98
147.vortex	1.46	1.34	1.30	1.14	1.33	1.84	0.75	1.12	2.09	2.13	1.30	1.53	0.31	0.78	0.00	0.00
CINT95 Avg	1.45	1.34	1.27	1.16	1.25	1.36	0.98	1.10	2.35	2.17	0.83	0.83	0.65	1.52	0.05	1.42
101.tomcatv	1.12	1.05	1.13	1.04	1.03	0.98	1.27	1.00	1.45	1.29	1.10	1.07	5.44	1.21	0.89	0.89
102.swim	1.06	1.07	1.05	1.05	1.11	1.11	1.27	1.27	1.20	1.22	0.99	0.99	0.85	0.90	1.00	1.00
103.su2cor	1.06	1.06	1.07	1.04	1.01	1.00	1.05	1.00	5.58	2.24	1.06	1.00	0.84	0.91	0.98	0.99
104.hydro2d	1.12	1.05	1.22	1.02	0.99	1.00	1.78	1.01	1.32	1.81	0.20	2.30	0.94	0.98	0.99	0.99
107.mgrid	1.09	1.05	1.03	1.03	1.01	1.01	0.84	0.87	1.27	1.37	1.01	2.34	0.66	1.00	1.02	1.02
110.applu	1.25	1.03	1.21	1.06	1.00	0.98	1.26	1.00	1.11	1.02	1.11	1.02	1.08	1.01	0.97	1.00
125.turb3d	1.39	1.14	1.13	1.04	0.98	0.98	0.95	0.94	27.93	3.19	0.99	0.96	2.21	10.31	0.98	1.18
141.apsi	1.02	0.98	1.08	1.02	0.99	1.00	0.98	1.02	2.04	3.23	1.02	1.87	0.54	0.86	0.97	1.02
145.fpppp	0.96	0.90	1.00	1.00	0.97	1.00	0.94	0.97	0.44	0.40	0.97	0.97	0.05	0.02	0.86	0.88
146.wave5	1.07	0.98	1.11	1.01	1.01	0.99	1.12	0.99	0.19	0.13	0.67	0.53	0.92	0.86	1.00	1.00
CFP95 Avg	1.10	1.01	1.09	1.03	1.00	1.00	1.05	0.97	0.61	0.44	0.96	1.12	0.60	0.93	0.94	0.96
SPEC95 Avg	1.19	1.10	1.14	1.06	1.00	1.00	1.04	0.98	0.93	0.76	0.86	0.90	0.60	0.94	0.94	0.96

Table 2: Perturbation of hardware metrics from profiling. **F** reports the ratio of each metric, recorded using flow sensitive profiling (intraprocedural paths), to the corresponding metric in the uninstrumented program. **C** reports the ratio of a metric, recorded using context sensitive profiling (call graph paths), to the metric in the uninstrumented program.

			Avg	Avg Out	Height		Max	Call Sites		
Benchmark	**Size**	**Nodes**	**Node Size**	**Degree**	**Avg**	**Max**	**Replication**	**All**	**Used**	**One Path**
099.go	1.1e7	26894	376.2	6.3	10.5	18.0	9308	39585	29393	3226
124.m88ksim	1.2e6	2617	416.8	4.7	6.5	13.0	228	10282	3247	1868
126.gcc	2.1e7	28863	812.0	6.4	10.2	25.0	1724	277042	65820	21867
129.compress	8.9e4	253	316.3	3.6	5.2	12.0	28	847	336	201
130.li	1.0e6	3227	297.5	2.4	8.9	18.0	356	7176	4231	2252
132.ijpeg	1.3e6	2311	541.3	3.7	9.2	18.0	147	7210	2513	1663
134.perl	3.6e6	5517	631.3	2.6	7.3	16.0	539	21392	8465	4634
147.vortex	2.1e8	257710	763.6	2.6	15.2	30.0	30511	992956	443116	231866
101.tomcatv	2.5e5	782	294.3	3.4	7.1	14.0	73	2198	1148	684
102.swim	3.9e5	950	380.7	3.6	6.4	12.0	89	3071	1533	767
103.su2cor	8.8e5	1867	437.0	4.3	5.9	14.0	118	5869	3127	1059
104.hydro2d	1.8e6	3255	529.6	3.9	7.1	16.0	395	10488	4445	2343
107.mgrid	6.5e5	1272	475.7	4.1	6.2	12.0	269	3978	1702	773
110.applu	4.1e5	1193	314.1	3.1	6.2	12.0	133	3768	2248	959
125.turb3d	1.4e6	4295	291.5	3.3	8.0	16.0	1305	9901	5409	2277
141.apsi	8.5e6	12494	638.4	4.9	8.2	18.0	2161	53148	20572	6813
145.fpppp	2.8e5	806	319.2	3.1	6.2	13.0	107	2439	1643	671
146.wave5	1.5e6	2580	535.4	4.5	7.8	15.0	632	7942	3753	1676

Table 3: Statistics for a CCT with intraprocedural path information in the nodes. **Size** is size (in bytes) of the profile file. **Nodes** is the number of nodes in the CCT. **Avg Node Size** reports the average size of an allocated call record (bytes). **Avg Out Degree** reports the average number of children of interior nodes. **Height** reports the average and maximum height of the tree. **Max Replication** reports the maximum number of distinct call records for any routine in the CCT. Finally, **Call Sites** reports the total number of call sites in all call records. **Used** is call sites that were actually reached. **One Path** is call sites that were reached in a given call record by exactly one path from the procedure's entry.

	All Paths			Hot Paths						Cold Paths		
				Dense Paths			Sparse Paths					
Benchmark	**Num**	**Inst**	**Miss**	**Num**	**Inst**	**Miss**	**Num**	**Inst**	**Miss**	**Num**	**Inst**	**Miss**
099.go	26628	3.2e10	1.1e9	7	8.5%	13.1%	4	12.8%	8.2%	26617	78.7%	78.7%
124.m88ksim	1115	7.8e10	6.3e8	22	28.9%	69.6%	2	20.2%	3.1%	1091	50.9%	27.3%
126.gcc	11769	1.7e8	3.8e6	3	4.3%	8.0%	0	0%	0%	11766	95.7%	92.0%
129.compress	249	5e10	1.3e9	11	34.0%	85.5%	4	19.6%	7.5%	234	46.4%	7.1%
130.li	811	5.6e10	1.1e9	13	25.8%	58.7%	10	34.3%	21.7%	788	39.9%	19.6%
132.ijpeg	1284	3.6e10	1.6e8	11	37.3%	77.6%	1	3.2%	3.1%	1272	59.5%	19.2%
134.perl	1427	2.5e10	6.7e8	20	27.3%	48.8%	8	14.4%	10.4%	1399	58.3%	40.8%
147.vortex	2244	7.3e10	1.2e9	14	39.3%	59.3%	3	15.5%	10.3%	2227	45.3%	30.4%
CINT95 Avg				12.6	25.7%	52.6%	4.0	15.0%	8.0%	5674.2	59.3%	39.4%
101.tomcatv	427	5.3e10	2.6e9	3	29.9%	62.7%	2	66.8%	34.9%	422	3.3%	2.4%
102.swim	377	2.7e10	2.5e9	1	24.9%	50.9%	2	72.9%	47.0%	374	2.3%	2.1%
103.su2cor	950	5.2e10	2.3e9	14	57.1%	83.3%	2	12.7%	9.8%	934	30.2%	6.8%
104.hydro2d	1816	9.3e10	3.6e9	17	27.7%	53.9%	10	43.4%	28.4%	1789	28.9%	17.7%
107.mgrid	588	1.4e11	3e9	9	7.4%	17.8%	2	87.5%	73.5%	577	5.1%	8.7%
110.applu	637	9.5e10	1.7e9	22	29.6%	74.1%	3	28.9%	8.2%	612	41.5%	17.7%
125.turb3d	672	1.7e11	3.5e9	12	14.1%	72.9%	6	24.9%	11.2%	654	60.9%	15.9%
141.apsi	1065	5e10	1.8e9	15	24.5%	68.5%	3	7.7%	4.8%	1047	67.8%	26.7%
145.fpppp	1067	2.5e11	6.1e9	5	51.3%	63.3%	7	24.1%	16.2%	1055	24.6%	20.5%
146.wave5	934	6.3e10	3.1e9	12	38.4%	62.9%	8	43.4%	24.5%	914	18.2%	12.6%
CFP95 Avg				11.0	30.5%	61.0%	4.5	41.2%	25.9%	837.8	28.3%	13.1%
SPEC95 Avg				11.7	28.4%	57.3%	4.3	29.6%	17.9%	2987.3	42.1%	24.8%
SPEC95 Avg - go, gcc				12.6	31.1	63.1	4.6	32.5	19.7	961.8	36.4	17.2

Table 4: L1 data cache misses paths. **Hot Paths** are intraprocedural paths that incur at least 1% of a program's cache misses. **Cold paths** are the other paths. **Dense Paths** are hot paths that have an above average miss rate. **Sparse paths** are hot paths with a below average miss rate. **Num** is the quantity of paths in the category. **Inst** is the instructions executed along these paths. **Miss** is the L1 data cache misses along the path.

Benchmark	Hot Procedures						Cold Procedures		
	Dense Procedures			Sparse Procedures					
	Num	Path/Proc	Misses	Num	Path/Proc	Misses	Num	Path/Proc	Misses
099.go	9	182.9	30.7%	8	405.9	39.3%	394	55.2	29.9%
124.m88ksim	12	18.0	77.2%	3	14.3	16.9%	207	4.1	5.9%
126.gcc	14	88.4	28.0%	10	155.1	16.1%	1055	8.5	55.9%
129.compress	2	22.5	91.6%	4	6.5	7.9%	63	2.8	0.5%
130.li	9	6.0	66.1%	6	9.5	27.0%	242	2.9	6.9%
132.ijpeg	9	42.2	86.9%	1	11.0	7.0%	255	3.5	6.1%
134.perl	13	14.2	76.2%	6	16.7	14.1%	217	5.3	9.6%
147.vortex	17	11.7	66.4%	4	2.2	12.2%	611	3.3	21.5%
CINT95 Avg	10.6	48.2	65.4%	5.2	77.7	17.6%	380.5	10.7	17.0%
101.tomcatv	1	41.0	99.7%	0	0.0	0.0%	142	2.7	0.3%
102.swim	1	13.0	51.9%	2	12.0	48.0%	137	2.5	0.1%
103.su2cor	8	32.4	90.1%	2	13.0	8.6%	196	3.4	1.4%
104.hydro2d	7	20.0	46.8%	5	171.4	49.0%	213	3.8	4.1%
107.mgrid	3	37.7	20.8%	2	7.0	78.9%	153	3.0	0.2%
110.applu	3	25.7	71.0%	3	33.0	28.9%	134	3.4	0.1%
125.turb3d	11	7.5	80.7%	3	11.7	18.1%	171	3.2	1.2%
141.apsi	12	9.4	66.5%	5	12.8	26.1%	224	4.0	7.4%
145.fpppp	2	8.0	51.0%	3	222.3	47.9%	135	2.8	1.1%
146.wave5	5	34.6	28.0%	3	32.3	68.5%	203	3.3	3.5%
CFP95 Avg	5.3	22.9	60.6%	2.8	51.6	37.4%	170.8	3.2	1.9%
SPEC95 Avg	7.7	34.2	62.8%	3.9	63.2	28.6%	264.0	6.5	8.6%
SPEC95 Avg - go, gcc	7.2	21.5	66.9%	3.2	36.0	28.7%	206.4	3.4	4.4%

Table 5: L1 data cache misses per procedure. **Hot Procedures** incur at least 1% of a program's cache misses. **Dense Procedures** have above average miss ratios, while **Sparse Procedures** have below average miss ratios. **Cold Procedures** incur fewer than 1% of a program's cache misses. **Path/Proc** is the average number of paths executed in procedures in each category. **Misses** is the fraction of misses incurred by procedures in the category.

counter wrap). These measurements also slightly perturbed the hardware counters, as the data collection process suspends the application process and briefly runs. Since the hardware counters were set to only record user processes events, the kernel was not directly measured. However, kernel code could indirectly affect metrics, for example by polluting the cache.

The table reports the ratio of each metric, collected with both flow and context sensitive profiling, to the uninstrumented program. In some cases, instrumentation can improve performance, for example by spreading apart stores and reducing store buffer stalls. The average perturbation for all metrics was small, though some programs exhibited large errors. We have not yet isolated the cause of these results. However, it is encouraging that the two techniques, flow and context sensitive profiling, typically obtained similar results.

6.3 CCT Statistics

Table 3 contains information on the size of a CCT. This data structure increases in size by a factor of 2–3x when the CCT is built on a call site basis. The total size of the data structure is a few hundred thousand bytes for most of the benchmarks. However, the CCT can be quite large for programs, such as 147.vortex, that contain many call paths. The table also shows that a CCT is a bushy, rather than tall, tree. In addition, the figures show that the routine with the most call records (**Max Replication**) often accounts for a large fraction of the tree nodes.

The final three columns report the total number of call sites in the allocated call records, how many of these sites are used, and how many sites only are reached by one intraprocedural path. The latter figure is particularly interesting, because in this case the combination of flow and context sensitive profiling produces as precise a result as complete interprocedural path profiling.

6.4 Flow Sensitive Profiling

This sections contains some sample measurements of hardware performance metrics along intraprocedural paths. Processor stalls have many manifestations—cache misses, branch mispredicts, load latency, or resource contention—but, in general, they occur when operations with long latencies cannot be overlapped or when excessive contention arises for resources. A processor's dynamic scheduling logic delays operations to preserve a program's semantics or resolve resource contention. Compiler techniques, such as trace scheduling, instruction scheduling, or loop transformations, reorder instructions to reduce stalls. Most compilers operate blindly, and apply these optimizations throughout a program, without an empirical basis for making tradeoffs. Our measurements show that this conventional approach is inefficient at best, as stalls are heavily concentrated on a very small number of paths, which we call hot paths.

6.4.1 Cache Misses, By Path

As an example, Table 4 reports hot paths in the SPEC95 benchmarks, for the UltraSPARC's L1 data cache, which is an on-chip 16 Kb, direct mapped cache. *Hot paths* are intraprocedural paths that incur at least 1% of the total cache misses—this threshold is a parameter to control the number of paths. Paths that are not hot are *cold paths*. A *dense path* is a hot path with a miss ratio above the program's average miss ratio. A *sparse path* is a hot path with a below average miss ratio. Sparse paths incur a large number of misses because they execute heavily, rather than because of poor data locality. Dense paths are more common than sparse paths, which is fortunate because it seems likely that dense paths are more likely to be optimized by a compiler.

In the SPEC95 benchmarks, excepting 099.go and 126.gcc, 1–22 (avg. 12) dense paths account for 51–86% (avg. 63%) of the L1 cache misses. Considering all hot paths (dense + sparse) increases the number of paths to 3–28, but also increase their coverage to 59–98% (avg. 83%) of the L1 misses.

The programs 099.go and 126.gcc have a well-known reputation for differing from the rest of the SPEC95 benchmarks. Our numbers corroborate this observation. They execute roughly an order of magnitude more paths than the other programs and each path makes a less significant contribution to the program's miss rate. It is therefore necessary to reduce the threshold for hot paths to 0.1%. This change finds 139 hot paths in 099.go that account for 55% of its misses and 172 hot paths in 126.gcc that account for 27% of its misses. In both cases, the number of hot paths is still around 1% of executed paths and a miniscule fraction of potential paths.

6.4.2 Cache Misses, By Procedure

Another way to apportion cache misses is by procedure. Table 5 reports hot procedures in the SPEC95 benchmarks, for the L1 data cache. A *hot procedure* incurs at least 1% of the cache misses. A *dense procedure* is a hot procedure that has an above-average miss ratio. A *sparse procedure* is a hot procedure with a below-average miss ratio.

Again, cache misses are heavily concentrated in a small portion of the program. From 1–24 (avg. 11.7) procedures account for 44–99% (avg. 91%) of the cache misses. This time, 099.go, 126.gcc, and 147.vortex have significantly lower coverage than the other programs. Again, lowering the threshold to 0.1% improves coverage, so that 82 procedures cover 97% of the misses in 099.go, 157 procedures cover 89% of the misses in 126.gcc, and 75 procedures cover 96% of the misses in 147.vortex.

6.4.3 Implications for Profiling

Table 5 also demonstrates that reporting cache misses by procedure may not help isolate the aspects of a program's behavior that caused cache misses. Hot procedures execute many paths (an average of 34 and 63, for dense and sparse procedures, respectively), so that knowing a procedure incurs many cache misses may not help isolate the path along which they occur. Moreover, collecting and reporting cache misses measurements at the statement level, in addition to being far more expensive than path profiling, does not alleviate this problem. In these benchmarks, the basic blocks along hot paths execute along an average of 16 different paths (of all type). Path profiling offers a low-cost way to provide insight into a program's dynamic behavior.

7 Related Work

This section describes previous work related to flow and context sensitive profiling.

7.1 Call-related Performance Measurement

Many profiling tools approximate context sensitive profiling information with heuristics. Tools such as *gprof* [GKM83] and *qpt* [BL94] use counts of the number of times that a caller invokes a callee to approximate the time in a procedure attributable to different callers. Because these profiling tools do not label procedure timings by context, information is lost that cannot be accurately recovered. Furthermore, the tools also make naive assumptions when propagating timing information in the presence of recursion, which results in further inaccuracy.

Abnormalities that result from approximating context sensitive information are well known [PF88]. Ponder and Fateman propose several instrumentation schemes to solve these problems. Their preferred solution associates procedure timing with a (caller, callee) pair rather than with a single procedure. This results in one level of context sensitive profiling. Our work generalizes this to complete contexts.

Pure Atria's commercial profiling system Quantify uses a representation similar to a CCT to record instruction counts in procedures and time spent in system calls [Ben96]. Details and overheads of this system are unpublished. This work goes further by incorporating path profiling and hardware performance metrics.

7.2 Context Sensitive Measurement Mechanisms

Call path profiling is another approach to context sensitive profiling [Hal92, HG93]. It, however, differs substantially in its implementation and overhead. Hall's scheme re-instruments and re-executes programs to collect call path profiling of LISP programs in a top-down manner [Hal92]. Initially, the call-sites in the "main" function are instrumented to record the time spent in each callee. Once measurements have been made, the system re-instruments the program at the next deepest level of the call graph and re-executes it to examine particular behavior of interest, and so on. Since the amount of instrumentation is small, overhead introduced in a run can be quite low. However, iterative re-instrumentation and re-execution can be impractically expensive and does not work for programs with non-reproducible behavior. By contrast, our technique requires only one instrumentation and execution phase to record complete information for all calling contexts.

Goldberg and Hall used process sampling to record context sensitive metrics for Unix processes [HG93]. By interrupting a process and tracing the call stack, they constructed a context for the performance metric. Beyond the inaccuracy introduced by sampling, their approach has two disadvantages. Every sample requires walking the call stack to establish the context. Also, the size of their data structure is unbounded, since each sample is recorded along with its call stack.

7.3 Calling Context Trees

CCTs are related to Sharir and Pnueli's call strings, which are sequences of calls used to label values for interprocedural flow analysis [SP81]. In interprocedural analysis, the need to bound the size of the representation of recursive programs and to define a distributive meet operator, made call strings impractical. CCTs, because they only need to capture a single execution behavior, rather than all possible behaviors, are a practical data structure.

Jerding, Stasko and Ball describe another approach for compacting dynamic call trees that proceeds in a bottom-up fashion [JSB97]. Using hash consing, they create a dag structure in which identical subtrees from the call tree are represented exactly once in the dag. In this approach, two activations with identical contexts may be represented by different nodes in the dag, since node equivalence is defined by the subtree rooted at a node rather than the path to a node.

8 Summary

Flow and context sensitivity are used in data-flow analysis to increase the precision of static program analysis. This paper applied these techniques to a dynamic program analysis—program profiling—where they improve the precision of reporting hardware performance metrics. Previous tools associated metrics with programs' syntactic components, which missed spatial or temporal interactions between statements and procedures. Paths through a procedure or call graph capture much of a program's temporal behavior and provide the context to interpret hardware metrics.

This paper showed how to extend our efficient path profiling algorithm to track hardware metrics along every path through a procedure. In addition, it described a simple data structure that can associate these metrics with paths through a program's call graph. Measurements of the SPEC95 benchmarks showed that the run-time overhead of flow and context sensitive profiling is reasonable and that these techniques effectively identify a small number of hot paths that are the profitable candidates for optimization.

The program paths identified by profiling have close ties to compilation. Many compiler optimizations attempt to reduce the number of instructions executed on paths through a procedure [MR81, MW95, BGS97]. In order to ensure profitability, these optimizations must not increase the total instructions executed along all paths through a procedure or, more strongly, over any path through a procedure. In addition, these optimizations duplicate paths to customize them, which increases code size, to the detriment of high issue rate processors. Much of the optimization's complexity, and many of their heuristics, stems from the assumption that all paths are equally likely. Compilers can use path profiles to identify portions of a program that would benefit from optimization, and as an empirical basis for making optimization tradeoffs.

Acknowledgements

Many thanks to Ashok Singhal of Sun Microsystems for assistance in understanding and using the UltraSPARC hardware counters. Trishul Chilimbi, Mark Hill, and Chris Lukas provided helpful comments on this paper.

References

[Bal94] Thomas Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.

[Ben96] Jim Bennett (PureAtria, Inc.). Personal communication, November 1996.

[BGS97] R. Bodik, R. Gupta, and M. L. Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[BL94] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(3):1319–1360, July 1994.

[BL96] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.

[CMH91] P. P. Chang, S. A. Mahlke, and W-M. W. Hwu. Using profile information to assist classic code optimizations. *Software–Practice and Experience*, 21(12):1301–1321, December 1991.

[GKM83] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software–Practice and Experience*, 13:671–685, 1983.

[Hal92] R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE92)*, pages 296–306, 1992.

[HG93] R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 1–14., Cincinnati, OH, 1993.

[JSB97] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of 1997 International Conference on Software Engineering (to appear)*, May 1997.

[Knu71] D. E. Knuth. An empirical study of FORTRAN programs. *Software–Practice and Experience*, 1(2):105–133, June 1971.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[LW94] Alvin R. Lebeck and David A. Wood. Cache profiling and the spec benchmarks: A case study. *IEEE Computer*, 27(10):15–26, October 1994.

[MR81] E. Morel and C. Renvoise. Interprocedural elimination of partial redundancies. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[MRW92] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. Performance measurement instrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–450, July 1992.

[MW95] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 56–66, June 1995.

[PF88] C. Ponder and R. J. Fateman. Inaccuracies in program profilers. *Software–Practice and Experience*, 18:459–467, May 1988.

[RBDL97] T. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Technical Report 1335, Computer Sciences Department, University of Wisconsin, Madison, WI*, January 1997.

[Sof93] Pure Software. *Quantify User's Guide*. 1993.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[Sun96] Sun Microelectronics. *UltraSPARC User's Manual*, 1996.

[WHH80] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6(3):278–286, May 1980.