ABSTRACT
Fine-grained binary instrumentation is a popular technique to monitor program execution. Intel’s Pin is a leading dynamic binary instrumentation framework for building program measurement and analysis tools. A key feature missing in Pin is the ability to associate call paths with instructions as they execute. The availability of calling context information enables Pin tools to provide more detailed diagnostic feedback. This paper introduces CCTLib—a call path collection library that any Pin tool can use to obtain the full calling context at any and every machine instruction that executes. CCTLib not only associates any instruction with source code along the call path, but also points to the data object accessed by the instruction if it is a memory access.

With CCTLib, we demonstrate that collecting call paths on each executed instruction is possible, even for reasonably long running programs. Prior art in call path collection for Pin has several limitations. Compared to other open-source Pin tools for call path collection, CCTLib provides richer information that is accurate even for programs with complex control flow and does so with about 30% less overhead—a difference of 14× on average. CCTLib enables attribution of metrics in Pin tools to both code and data.

Categories and Subject Descriptors

General Terms
Performance, Measurement.

Keywords
Performance analysis, Call path profiling, Data-centric attribution.

1. INTRODUCTION
Tracking program execution at every machine instruction level is a popular technique for identifying several classes of software issues. Two of the most common applications of such fine-grained execution monitoring (FEM) are execution characterization and correctness checking. For execution characterization, FEM is employed for reuse-distance analysis [19], cache simulation [14], computational and/or memory redundancy detection [8], as well as power and energy analysis [27], name just a few applications. For correctness checking, FEM is employed for taint analysis [5], concurrency bug detection [21], and malware analysis [27], among other applications. FEM is also employed in hardware emulation, reverse engineering [17], software resiliency, testing, tracing, debugging, and execution replay [25].

Two approaches for FEM are static and dynamic binary instrumentation. Several frameworks [6, 22, 29] support static and/or dynamic binary rewriting. By their very nature, techniques for FEM add non-trivial overhead. FEM overhead is often higher using dynamic rewriting. Tool-specific code invoked for instrumented instruction(s) is often called as analysis routine. Since each instruction in the original program can potentially invoke an analysis routine during monitoring, tool writers strive to limit the overhead of their analysis routines. Naturally, performing less work per analysis routine lowers a tool’s runtime overhead. However, the less measurement data a tool gathers, the less insightful its analyses and feedback are likely to be.

FEM tools can provide rich and insightful feedback about an execution by associating data collected for each monitored instruction to source code locations along the instruction’s dynamic call path. Consider an FEM tool that detects data races in parallel program executions, e.g., Helgrind—a Valgrind tool, by tracking each memory access and identifying conflicting accesses by threads. A tool that attributes a race to the source lines responsible is much more useful than one that only identifies the presence of a race. However, even attributing races to source lines isn’t enough: the same source lines could be reached by myriad program paths, not all of which may cause a data race. Having dynamic calling context information available to FEM tools, such as data race detectors, enables them to provide richer feedback that is easier to understand by tool users. Making such information available to FEM tools at an affordable overhead is one of the key contributions of our work.

Intel’s Pin [22] is a leading dynamic binary instrumentation framework for developing FEM tools. A key missing feature in Pin is the ability to provide the call path at
In this paper, we introduce CCTLib—a call path collection library for Pin that employs a shadow stack to support on-demand use of call paths instead of logging or call stack unwinding. CCTLib maintains information about call paths efficiently and compactly in a calling context tree (CCT). CCTLib is modest in overhead and usable in practice for reasonably long running programs. CCTLib collects accurate call paths even through dynamically-loaded libraries, stripped libraries, and executable code for which the compiler recorded incorrect or incomplete information about function bounds. CCTLib’s operation is largely transparent; it requires only initialization at start up and calling an interface procedure at any point in a Pin analysis routine to inspect the call path. CCTLib can be used by any Pin-based FEM tool. CCTLib is for FEM tools only and it should not inspect the call path. CCTLib can be used by any Pin-based FEM tool. CCTLib is for FEM tools only and it should not inspect the call path.

Context-centric analysis is only one facet of program monitoring. While context-centric attribution identifies problematic instructions, such instructions might be associated with different data objects. Fine-grained data-centric attribution is a technique that associates memory location(s) involved in each instruction with their corresponding data objects (e.g., static, stack, and heap variables). Pinpointing problematic data objects is an important aspect of execution analysis. Moreover, data-centric information is necessary for optimizing memory reuse distance and eliminating false sharing in multi-threaded programs. CCTLib supports both context- and data-centric attributions. CCTLib’s data-centric attribution leverages calling context information as well. Figure 1 provides a schematic view of CCTLib.

CCTLib’s call paths with instruction-level attribution can be queried in a constant time making it well suited for FEM tools. CCTLib outperforms previous call path collection schemes for Pin tools [4, 8, 19] by providing richer contextual information, being accurate under adverse circumstances (e.g., stripped code), and adding lower overhead.

The contribution of this paper is a description of the design, implementation, and evaluation of CCTLib—an open-source framework for accurate and efficient collection and attribution of context- and data-centric information in Pin tools. CCTLib enables more accurate and informative FEM tools than prior work. CCTLib outperforms a prior state-of-the-art implementation of call path collection within Pin [8] by about 30%. We demonstrate the utility of CCTLib’s context- and data-centric capabilities by collecting such information on each executed instruction for a suite of long running programs. The rest of the paper is organized as follows. Section 2 provides the necessary background for our work. Section 3 describes our methodology for call path collection. Section 4 describes the implementation of CCTLib. Section 5 evaluates our approach. Section 6 summarizes our conclusions.

2. BACKGROUND AND RELATED WORK

In this section, we discuss the state-of-the-art in call path collection, data-centric attribution, and outline challenges maintaining calling context information in Pin.

2.1 Call path collection techniques

There are two principal techniques used to collect call paths in an execution: unwinding the call stack and maintaining a shadow call stack.

Call stack unwinding on x86 architectures walks procedure frames on the call stack using either compiler-recorded information (e.g., libunwind [26]) or results from binary analysis (e.g., HPCToolkit [2]). Stack unwinding does not require instrumentation and it does not maintain state information at each call and return. As a result, it adds no execution overhead, except when a stack trace is requested. This technique is well suited for coarse-grained execution monitoring, e.g., sampling-based performance tools and debuggers. However, applying call stack unwinding to gather calling context information for each machine instruction executed would frequently gather slowly changing calling context at an acceptably high overhead.

Stack shadowing involves maintaining calling context as the execution unfolds; this can be accomplished by instrumenting every function entry and exit either at compile time or using binary rewriting. Stack shadowing is used by tools including Scalasca [9] and TAU [23]. The advantage of stack shadowing is that at every instant the stack trace is ready and hence it can be collected in a constant time. Stack shadowing is well suited for FEM tools. A disadvantage of stack shadowing is that instrumentation of calls and returns adds overhead. Furthermore, it is not stateless and it requires extra space to maintain the shadow stack.

Combining call stack unwinding and stack shadowing yields hybrid call path collection techniques explored by Liu et al. [19] and Szebenyi et al. [55]. Dyninst [6] provides a stack walker API that can be queried on any instruction. Since Dyninst uses unwinding for call path collection, the overhead of doing so on each instruction makes it infeasible for FEM tools.

Valgrind [29] also provides a stack unwinding framework for use by client tools that is unsuitable for fine-grained call path collection. Callgrind, a Valgrind tool, maintains a shadow stack, which makes it possible to recover call paths.
at a finer grain. However, instead of maintaining a CCT, Callgrind maintains a directory of call paths at greater expense. Furthermore, the shadow stack maintained by Callgrind is not exported for use by other Valgrind tools.

Although at one time DynamoRio maintained a “software return stack” for the purpose of improving branch prediction \[4\], it was judged unsuccessful for this purpose and a call path abstraction was never made available as part of the tool’s interface.

### 2.2 Pin and call path collection

Intel’s Pin \[22\], a leading binary rewriting framework for FEM tools, does not provide any API for collecting call paths. Hence, we decided to implement one ourselves. Tools that use Pin are called Pin tools.

Providing access to an application’s call path while executing analysis routines in a Pin tool poses several challenges. First, unwind libraries can’t be employed out of the box since Pin Just-In-Time (JIT) compiles code from the original binary and executes the JIT-compiled code from its code cache. Second, employing call path unwinding for FEM is prohibitively expensive. Finally, even if unwinding were possible in the JITed code, transitions between the Pin framework and application code would clutter the call stack with unwanted information.

To the best of our knowledge, there are only three prior tools that collect call paths with Pin. Intel’s inspector \[12\] uses Pin for data race detection and provides full calling context for conflicting accesses. However this tool is not open source; we can neither evaluate its overhead for unwinding (full data race detection for threaded programs is expensive) nor know about its implementation. Maid \[4\] builds a shadow stack as a vector of activation records. It maintains the stack size at each level and detects unusual control transfers, e.g., exceptions, by identifying the mismatch in stack size during calls and returns. It adjusts call stacks by unwinding in the event of exceptions. However, in some cases, this approach can lead to incorrect calling contexts which persists for rest of the execution. Maid only maintains information about the current call path, and not historical information (e.g., a CCT). Furthermore, it provides only function names but not instruction/line-level information for call sites.

DeadSpy \[8\] builds calling context trees (CCT) in Pin. DeadSpy supports line-level attribution only for the leaf nodes of the call path, interior frames have only function names. DeadSpy’s approach serializes multi-threaded codes. DeadSpy does not support parent-child association between CCTs of different threads. Finally, DeadSpy does not produce correct CCTs in the event of exceptional control flows such as C++ exceptions and setjmp/longjmp. CCTLib addresses all these issues with substantially lower overhead.

Jalan et al. \[13\] developed a Pin tool for call graph construction. Call graphs provide caller-callee relations, but do not provide source line-level attribution. In CCTLib we provide a calling context tree, which is richer in information. From a CCT, one can derive a call graph if desired.

### 2.3 Data-centric attribution

Performance tools such as MemProf \[16\], Memphis \[25\], and HPCToolkit \[18\] employ data-centric attribution to diagnose memory-related bottlenecks. Memspy \[24\] monitors memory accesses using a cache simulator. Memspy attributes accesses to only heap variables. In many programs, static variables are also of interest. MACPO \[32\] uses LLVM \[20\] to instrument memory accesses and attribute accesses to variables. Though MACPO-generated code has comparatively low overhead (under 6x), it requires compilation with LLVM and it is problematic to attribute costs in dynamically loaded libraries that are not compiled with LLVM. Unlike MACPO, CCTLib works with any x86 compiler. Finally, ThreadSpotter \[33\] employs a “last write” method for data-centric attribution. It tracks store operations and uses them to identify variables at the source code level. This technique, which has an overhead of under 20%, leverages the fact that usually there is only one store (LHS value) in a source line.

### 3. CCTLib METHODOLOGY

CCTLib primarily employs stack shadowing for collecting call paths. We recognize two key aspects in collecting call paths: accuracy and efficiency.

#### Call path accuracy

Stack shadowing is typically performed by inserting instrumentation at function entries and exits. For binary instrumentation, function entries are typically discovered via symbol information and function exits are discovered via static instruction disassembly. Instrumenting function entries and exits can be inaccurate when function symbols are missing or wrong \[36\] and when machine code disassembly is incorrect due to incorrect assumptions about instruction boundaries. Despite significant effort to perform accurate static x86 disassembly \[11\], the technique is not foolproof.

Due to the stateful nature of stack shadowing, instrumentation of function entries and exits must match accurately. A single mismatched exit or entry can corrupt the shadow stack leading to incorrect call paths for rest of the execution. We conducted a small experiment using Pin to identify the potential impact of imprecision in binary analysis on call path collection:

1. We computed the fraction of instructions executed at runtime that did not fall under any function range discovered via Pin’s static disassembly.
2. We computed the fraction of stack-size-affecting instructions that are executed at runtime for which the disassembly was incorrect (i.e., start address of the instruction fell in the middle of another instruction during static disassembly).

Table 1 shows some applications with a non-trivial amount of imprecision in disassembly\[4\]. We did not intentionally strip any symbol from an executable. However, we are aware that Nvidia’s CUDA libraries used by LULESH-CUDA \[15\] have symbols stripped. If we stripped executables, incorrect disassembly would be more frequent. For example, for a stripped version of the SPEC CPU2006 omnetpp reference benchmark, the incorrect disassembly was about 60%. Working with stripped applications (just not stripped libraries) is a valuable use case for software security. High disassembly errors for LULESH-OpenMP \[15\] are likely due to the following reason: in optimized code generated by recent Intel’s icpc compiler version 13.0.0, machine code for

\[\text{The tests were conducted on a variety of modern 64-bit x86 Linux machines.}\]
outlined functions associated with OpenMP parallel regions and loops appears in the executable amidst machine code for the function in which the regions or loops originated. As a result, the machine code for the enclosing function is split into discontinuous pieces by the embedded outlined code. Furthermore, code for embedded outlined functions is not labeled with function symbols.

An alternate method for building shadow stack uses instrumentation of call and return instructions instead of function entries and exits. If we track every instruction, we can't possibly miss any call or return instruction and this ensures accurate shadow stacks. One can instrument and monitor every instruction in Pin via trace instrumentation. A trace in Pin jargon, is a single entry multiple exit code sequence—for example, a branch starts a new trace at the target, and a call, return, or jump ends the trace. Trace instrumentation happens immediately before a code sequence is first executed. Trace instrumentation has the advantage of not missing the instrumentation of any executed instruction. In the rest of the paper, we use the term Pin-trace to refer to Pin's traces to distinguish it from our internal representation of traces in CCTLib. By leveraging Pin traces, CCTLib avoids relying on compiler-based information for function boundaries or for machine instruction disassembly. This eliminates the possibility of missing any call and return. Consequently, CCTLib's call path collection is resilient to symbol stripping. CCTLib uses compiler-based information only to map machine instructions back to source lines. CCTLib relies on symbol information for a limited set of functions such as thread_create; however, such symbols are available in any dynamically-linked application. Data-centric attribution with CCTLib, however, relies on symbol information to attribute memory addresses back to static variables.

**Call path efficiency.**

Since our goal is to support collection of call paths on every monitored instruction in an FEM tool, our techniques need to be efficient both in space and time. CCTLib employs CCTs to store call path information. Common prefixes are shared across all call paths, which reduces space overhead dramatically. CCTLib pays particular attention to use constant-time operations within its analysis routines to keep overhead low. By employing efficient data structures, e.g., splay trees [34] for maps, our implementation achieves acceptable overhead.

### 4. DESIGN AND IMPLEMENTATION

In this section we describe the implementation of CCTLib. Section 4.1 describes how we collect a CCT for context-centric attribution. Section 4.2 describes the support for data-centric attribution within CCTLib. Table 2 lists the key APIs that CCTLib exposes to its client Pin tools. GetContextHandle obtains the current call path. GetDataObjectHandle obtains the data object accessed at an address.

### 4.1 Collecting a CCT in Pin

CCTLib builds its shadow stack by instrumenting each call and return machine instruction and stores each call path in a CCT. The CCT at a given instant has all call paths seen in the execution up to that point. The path implied by the current node in the CCT to its root represents the calling context at that point. We identify each call path with a unique 32-bit handle (ContextHandle_t) that enables us to reconstruct any call path during or after program execution.

A particularly challenging part of call path collection is source-level attribution. At each instruction, Pin provides its instruction pointer (IP) which can be mapped back to source line(s); hypothetically, one can look for this IP in the already recorded IPs under the current node of the CCT. However, such lookup on each instruction is prohibitively expensive. Variable length x86 instructions, unknown or incorrect function bounds, and tail calls compound the problem. Trace instrumentation, along with a shadow mapping from Pin-traces to their constituent instructions, enables us to solve this problem with a constant-time algorithm.

### Table 1: Impact of incorrect/incomplete static disassembly on call path collection in Pin.

<table>
<thead>
<tr>
<th>No.</th>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>typedef bool (*TrackInsCallback)(INS ins)</td>
<td>Client Pin tool callback to determine if the given instruction is to be instrumented.</td>
</tr>
<tr>
<td>2</td>
<td>typedef void (*ClientInstrumentationCallback)(INS ins, VOID *v, OpaqueHandle_t handle)</td>
<td>Callback to allow the client Pin tools to add their own instruction-level instrumentation.</td>
</tr>
<tr>
<td>3</td>
<td>bool Initialize(TrackInsCallback cb=TRACK_ALL_INS, ClientInstrumentationCallback clientInsCB=0, void * clientCallbackArg=0)</td>
<td>Initializes CCTLib.</td>
</tr>
<tr>
<td>4</td>
<td>ContextHandle_t GetContextHandle(OpaqueHandle_t handle=0)</td>
<td>Returns a handle that represents the calling context of the current thread.</td>
</tr>
<tr>
<td>5</td>
<td>DataHandle_t GetDataObjectHandle(void * address)</td>
<td>Returns a handle that represents the data object accessed by address.</td>
</tr>
</tbody>
</table>

### Table 2: APIs and data structures exposed by CCTLib.
We describe the necessary instrumentation, key internal data structures, and basic runtime actions performed by CCTLib’s analysis routines in Sections 4.1.1–4.1.3. In Section 4.1.4, we contrast CCTLib’s internals with DeadSpy’s internals. Sections 4.1.5–4.1.8 describe the details of handling complex control flows. Finally, Section 4.1.9 describes CCTLib’s current limitations.

4.1.1 Instrumentation

We intercept Pin-trace creation and instrument the following places:

- entry to a Pin-trace.
- each call and return instruction in the trace, and
- any other instruction(s) the client Pin tool decides to track.

On each Pin-trace creation, we assign a unique identifier (traceId) to the trace. Further, we memorize the association between the traceId and all instruction addresses in that trace in a map (ShadowTraceMap). However, we need not maintain all instructions in the trace in the ShadowTraceMap; instead, our map contains only those instructions that are designated as “to be tracked” by the client Pin tool (Table 2, row 1) and all call and return instructions in the trace. The traceId is made known to the analysis routine added to each trace entry. Finally, we add instrumentation before each call and return instruction in the trace. We also instrument setjmp, longjmp, pthread_create, and _Unwind_SetIP functions; details about this are described in sections that follow.

4.1.2 Supporting data structures

CCTLib’s CCT is composed of TraceNodes as shown in Figure 2. A CCTLib TraceNode logically represents a Pin-trace. There is many-to-one relationship between TraceNodes and Pin-traces since the same Pin-trace can be executed from multiple calling contexts. Each TraceNode has three fields — 1) an array of INodes which we refer to as IPNodeVec, 2) traceId, and 3) a parent pointer to an IPNode. Each element of IPNodeVec logically represents an instruction in the corresponding Pin-trace. By default, the size of IPNodeVec equals the number of instructions in the corresponding Pin-trace. This enables us to associate every instruction with full calling context whenever the CCTLib client desires; supporting calling context for each instruction forms the highest overhead case. A client Pin tool may tailor CCTLib’s tracking of calling contexts for a particular task. For example, a data race detection tool, which needs to track only memory access instructions, does not require call paths for non-memory related instructions; in such cases CCTLib can be initialized to track only a client-specified “class” of instructions. CCTLib provides a callback that enables its client Pin tools to specify which instructions need CCT information at trace instrumentation time. No matter what the client Pin tool specifies, CCTLib will always include all call and return instructions in its IPNodeVec. In the best case, the number of entries in an IPNodeVec for a Pin trace equals the number of call and return instructions the trace; this is the lowest overhead case.

There is 1-1 relationship between the Nth slot in IPNodeVec in a TraceNode with traceId I and the instruction recorded in the shadow trace map at ShadowTraceMap[I][N]. This association enables us to recover the instruction pointer associated with each instruction represented in a TraceNode without having to maintain instruction addresses themselves in each IPNode. The association is both shared by different TraceNodes within a CCT as well as CCTs of different threads. Figure 3 shows the ShadowTraceMap data structure.

The details of TraceNode and IPNode are shown in Figure 4. Each element of IPNode has two fields. First, a parent field which is a pointer to the parent TraceNode. Second, a pointer to the root of a splay tree, possibly null. The splay tree represents all traces to constituent instructions.

[Figure 2: A CCTLib Calling Context Tree.]

[Table 2: A CCTLib’s TraceNode and IPNode structures.]

[Figure 3: ShadowTraceMap: CCTLib’s mapping from traces to constituent instructions.]

[Figure 4: CCTLib’s TraceNode and IPNode.]
A call to function \( C \) under execution.

4.1.3 Actions at run time

CCTLib maintains two thread-local variables curTraceNode and curIPNode. The curTraceNode points to the current TraceNode. The curIPNode points a slot in the curTraceNode's IPNodeVec that logically represents the current instruction under execution. At runtime, the following five cases arise:

1. Each time a call instruction is executed, the analysis routine sets a thread-local flag (isCall).

2. When a new Pin-trace with id traceId is entered immediately following a call instruction (inferred by inspecting the isCall flag), it represents a transition from a caller to a callee. In this case, an analysis routine at the trace entry executes the following steps:
   (a) Search for the callee TraceNode with key traceId in the splay tree rooted at curIPNode. If none is found, a new TraceNode with traceId as the key is created and inserted into the splay tree.
   (b) Set the curTraceNode to the callee TraceNode.
   (c) Reset the isCall flag.

3. When a new Pin-trace with id traceId is entered without a call instruction being executed, it represents a transition from one trace to the other within the same callee, or possibly a tail call. In this case, the analysis routine at the trace entry executes the following steps:
   (a) Search for the target TraceNode with the key traceId in the splay tree rooted at curIPNode->parent to find peer traces. If none is found, create and insert into the splay tree a TraceNode with traceId as key.
   (b) Set curTraceNode to the target TraceNode.

4. When a return instruction executes, set curTraceNode to its parent TraceNode, simulating the return from function in the shadow call stack.

5. When any instruction in a trace executes, the analysis routine inserted before that instruction sets curIPNode to the corresponding slot of IPNodeVec. At trace instrumentation time, the index of the instruction in the Pin-trace is known (which will be its offset in IPNodeVec) and hence updating the curIPNode to the correct slot within IPNodeVec can be done in a constant time even if the execution does not follow a straight line path within a trace.

It is worth mentioning that all cases except 2a and 3a are constant time operations. Our choice of data structures is driven by their amenability to such constant time operations. Cases 2a and 3a are tree lookup operations and can incur a logarithmic complexity. The choice of splay trees ensures that recently accessed TraceNodes are near the root of the tree, which makes lookup fast in practice.

As an optimization, the client Pin tool may register a callback that CCTLib calls during trace instrumentation on each instruction designated as “to be tracked” by the client (Table 3 row 2). CCTLib passes the index number in IPNodeVec that corresponds to the instruction to the client-instrumentation callback (as opaqueHandle argument), which the client’s analysis routine can pass back to CCTLib’s GetContextHandle() function when querying the calling context for that instruction. With this technique, CCTLib eliminates the runtime overhead of updating curIPNode on each instruction execution; instead, curIPNode is derived on demand when the client Pin tool queries the context. The value of curIPNode is derived via a constant time indexing operation — IPNodeVec[opaqueHandle]; this eliminates the updating of curIPNode described in case 5.

At any instruction, its calling context identifier is simply the address of IPNode at curTraceNode->IPNodeVec[opaqueHandle]. We allocate all IPNodes from a fixed memory pool, hence, instead of pointer sized (8 bytes on 64-bit machines) handles, we can manage with just 32-bit handles. The handle uniquely represents a call path in the CCT; by traversing the parents pointers, the entire call path can be constructed. We allow for a maximum of \( 2^{32} \) unique call paths in a program. CCTLib provides the option to serialize the entire CCT during program termination and call path identifiers can be used to extract call paths during postmortem analysis. CCTLib also provides the option to serialize CCTs as a DOT file for visualization.

4.1.4 Contrast with DeadSpy’s CCT

Our technique of building CCTs extends our prior work on CCT construction in DeadSpy [3]. The CCTs in DeadSpy can only provide source line mapping information for the leaf node of a path; for interior frames DeadSpy can recover only function names. As a consequence, the code in Figure 5 which has multiple call sites to the same callee, leads to the ambiguous CCT shown in Figure 7. It is unclear from the call path in Figure 7 whether \( A \) was called from line 2 or line 3 of main. In contrast, builds CCT as shown in Figure 6. CCTLib has call site level attribution, which disambiguates two calls to \( A \) from two different call sites in the same callee. The Maid stack trace tool does not have call site level attribution and hence suffers from the same problem.

Another difference between DeadSpy’s and CCTLib’s CCT construction is the elimination of DeadSpy’s notion of ContextNode in CCTLib. DeadSpy represents function calls in a CCT using ContextNode, which serves as an umbrella for—all callees (ContextNode) and traces (TraceNode) belonging to a function. On a function call, DeadSpy’s scheme employs a two-level search. First, DeadSpy searches the target function under all callees of the current ContextNode represented by childContexts field, then DeadSpy searches the traces pointed to by childTraces in the callee ContextNode. DeadSpy uses hash tables as maps for these

---

1. If the client Pin tool decides to track only a subset of instructions, curIPNode points to the most recent tracked instruction.
lookups. In contrast, CCTLib consolidates both ContextNodes and TraceNodes into one TraceNode. CCTLib eliminates the two-level lookup with a single-level lookup during a call instruction as mentioned before in step 2a. Replacing two-levels of lookup with one-level lookup and employing splay trees, which are better suited for the lookup of frequently accessed items, improves performance as we show in Section 5. Experiments with the bzip2 SPEC CPU2006 reference benchmark showed that execution time of monitored programs using splay trees in CCTLib was 50% faster than using google’s sparsehash tables [10] and 24% faster than GNU C++’s std::hash_map.

4.1.5 Signal handling

When a signal is delivered, control asynchronously transfers to a registered signal handler. With CCTLib, the signal handler appears as a callee under the instruction where the signal was delivered. This is accomplished by simply setting the isCall flag in the analysis routine added using PIN_AddContextChangeFunction for signals. All callees of the signal handler will be treated as normal callees. The return from signal handler executes a return instruction enabling CCTLib to resume from the position in the CCT where the signal was delivered. If the client tool opts to not track all instructions, the signal handler appears as if it was called from the last tracked instruction that executed.

4.1.6 Handling Setjmp/Longjmp

The setjmp and longjmp APIs allow programs to bypass the normal call/return flow of control. With setjmp, the program memorizes the current architectural state of the program in a user provided buffer jmp_buf and with longjmp it restores the state stored in the given buffer. A longjmp may unwind multiple frames of the stack without executing a return. To ensure CCTLib produces correct call paths even after longjmp calls, CCTLib memorizes the association between the jmp_buf and the calling context where setjmp was called, and restores the same calling context after a longjmp, thus simulating the effect of longjmp in its shadow stack.

4.1.7 Shadow stack during C++ exceptions

When a C++ program throws an exception, the runtime performs stack unwinding to look for an exception handler. Once a handler is found, execution resumes at the handler, having unwound zero or more stack frames. The C++ unwinder calls _Unwind_SetIP as the last step of jumping to the handler function. The _Unwind_SetIP has the following signature: void _Unwind_SetIP(struct _Unwind_Context * context, uint value). The first argument (context) among other things contains information about the stack frame that can handle the current exception. It also includes information such as the instruction pointer in the handler’s frame that called a chain of functions which eventually led to the exception. The _Unwind_SetIP function overwrites its return address with the second argument value, which contains the address of the exception handler. Upon return from _Unwind_SetIP, control resumes in the exception handler instead of the original caller.

To mimic the effect of stack unwinding due to exception in the CCTLib’s shadow stack, CCTLib instruments the _Unwind_SetIP function and uses the information present in the context argument to set curTraceNode appropriately. On entering _Unwind_SetIP, CCTLib captures the first argument i.e., context. CCTLib calls the _Unwind_GetIP(context) to obtain the instruction pointer of the call site, exceptionCallerIP, in the ancestor frame where the exception handler is present. Now, CCTLib walks up the call chain, starting at curTraceNode, looking for the first TraceNode that contains exceptionCallerIP. CCTLib stops on the first found TraceNode t and records a pointer to it. On the return path from _Unwind_SetIP, CCTLib sets its curTraceNode to t resulting in the shadow call stack unwinding exactly the same number of frames as the original stack. Subsequent entry to the handler Pin-trace will follow the earlier described strategy under case 3.

This unwinding technique needs tailoring for systems that employ other strategies for exception handling.

4.1.8 CCT in multithreaded codes

CCTLib produces independent CCTs for each thread in a program execution. Most of CCTLib’s data structures are thread local except IPNodes, which are allocated from a shared memory pool. The allocation of IPNodes uses atomic operations. Consequently, multiple threads can build/manage their own CCTs concurrently. Since threads can have parent-child relationships, we associate the root of the child thread’s CCT to the creation point of its parent’s CCT.

Since Pin does not provide information about parent-child relationship between threads, CCTLib performs extra work to establish this relationship. We override pthread_create to accomplish this. Immediately after a parent thread returns from pthread_create, CCTLib publishes its call path and waits for the analysis routine added as part of a new thread creation to execute. During this interval all threads attempting to spawn children thread(s) are paused from making progress. The analysis routine executed as part of the newly spawned thread attaches the published call path as its parent, after which all threads resume.

4.1.9 Current limitations

Our current implementation does not support attaching to a running process. While we can determine partial call paths seen after attaching, parent-child thread relationships, and
handling exceptional flow of control are problematic. We wish to address this in future work.

4.2 Data-centric attribution in CCTLib

CCTLib supports attribution of memory addresses to heap and static data objects. For stack data objects, CCTLib marks them as on stack but does not associate them with individual stack variables. It is straightforward to capture the memory range for each thread’s stack and eliminate addresses falling in that range from further analysis.

4.2.1 Instrumentation for heap data objects

CCTLib instruments memory management functions such as malloc, calloc, realloc, and free. CCTLib uses the calling context of the dynamic memory allocation site such as malloc to uniquely identify a heap data object. The memory range and its associated allocation call path identifiers are maintained in a map for future attribution of metrics to data objects.

4.2.2 Instrumentation for static data objects

CCTLib uses symbol names to uniquely identify static data objects. CCTLib reads the symbol table from each loaded module and extracts the name and memory range allocated for each static data object. It records this information in a map for each load module.

4.2.3 Associating address to data object at runtime

A CCTLib client Pin tool queries GetDataObjectHandle, passing it an effective address. GetDataObjectHandle returns a DataHandle_t, which is a 40-bit handle that uniquely represents the data object. Eight bits are reserved to distinguish between stack, heap or static variables. The remaining 32 bits represent the object. In the case of a heap object, the 32-bit handle is a CCT handle (ContextHandle_t). In the case of static variables, the handle is an index into a string pool of symbol names.

Aforementioned maps for associating addresses to objects need to be scalable to allow multiple threads to simultaneously query them. We provide two implementations, one is parsimonious in memory but less efficient in performance, another one requires large memory but delivers efficient, constant-time lookup.

The first approach uses a novel balanced-tree-based map data structure that allows concurrent reads while the map is being updated. There is one map per load module and one map for heap addresses. <Address range, object handle> pairs are recorded in sorted order in these maps. Lookup in the tree has logarithmic cost. While map lookups are frequent, insertion and deletion are not. We devised a data structure where insertion and deletion are serialized, but address lookups are fully concurrent. The result delivers scalable high performance for threaded program.

The second approach employs a page-table based shadow memory analogous to [8]. For every allocated byte in the program, the shadow memory holds its 40-bit data handle. On each dynamic allocation, the corresponding range of shadow memory is populated with the call path handle. For each symbol for static variables, the corresponding range of shadow memory is initialized with the string pool handle for the symbol. At runtime, for every accessed memory address, CCTLib maps the address to its shadow location and fetches the corresponding data handle in constant time.

The shadow memory lookups and updates need no locking; hence the approach scales well. Naturally, shadow memory introduces at least 5x memory bloat. CCTLib provides both balanced-tree-based and shadow-memory-based methods for its client tools to tradeoff memory consumption vs. runtime overhead. In the next section, we will compare the performance of these different approaches.

5. EVALUATION

In this section, we evaluate CCTLib’s runtime and memory overhead on serial applications, as well as its scalability on parallel programs. We conducted our single-threaded experiments on 16-core Intel Sandy Bridge machines clocked at 2.2GHz, with 128GB of 1333MHz DDR3 running Red Hat Enterprise Linux Server v6.2 and GNU 4.4.6 tool chain. We conducted our parallel scaling experiments on a quad-socket 48-core system with four AMD Opteron 6168 processors clocked at 1.9 GHz with 128GB of 1333MHz DDR3 running CentOS 5.5 and GNU 4.4.5 tool chain. All applications were compiled with -O2 optimization. We chose a subset of the SPEC CPU2006 integer reference benchmarks along with three other applications for evaluation. Each application was chosen to represent a variety.

- ROSE compiler [31]: ROSE source-to-source compiler has close to 3 million lines of C++ code. We applied CCTLib when ROSE was compiling 80K lines of its own header files. The code does not show any spatial or temporal data locality. It has a large code footprint and deep call chains.
- LAMMPS [30]: LAMMPS is a molecular dynamics code with 500K lines of C++ code. We applied CCTLib on the in.rhodo input, which simulates Rhodo spin model. The code is compute intensive and has deep call chains.
- LULESH [15]: is a shock hydrodynamics mini-app; it solves the Sedov Blast Wave problem, which is one of the five challenge problems in the DARPA UHPC program. The code is memory intensive and does not have good parallel efficiency. It has frequent memory allocations and deallocations, which makes it an interesting use case for data-centric analysis.
- Parm悠闲 [18]: is a molecular dynamics code with 500K lines of C++ code. We applied CCTLib on the in.rhodo input, which simulates Rhodo spin model. The code is compute intensive and has deep call chains.

5.1 Runtime overhead on serial codes

Table 3 shows runtime overhead for executions of our benchmark codes. For SPEC CPU2006 benchmarks that have multiple inputs, we show the mean values (arithmetic mean for raw values and geometric mean for relative quantities) across all inputs. Column 2 shows the running time for each program in seconds. h264ref for each program in seconds. Columns 3–6 focus on CCTLib’s call path collection overhead.

The shadow memory lookups and updates need no locking; hence the approach scales well. Naturally, shadow memory introduces at least 5x memory bloat. CCTLib provides both balanced-tree-based and shadow-memory-based methods for its client tools to tradeoff memory consumption vs. runtime overhead. In the next section, we will compare the performance of these different approaches.
Table 3: Runtime overhead of CCTLib.

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Original run time in sec</td>
<td>CCTLib overhead in MB</td>
<td>Context-centric analysis</td>
<td>Data-centric analysis</td>
<td>Overhead (tree-based)</td>
<td>Overhead (shadow memory)</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>astar</td>
<td>276.26</td>
<td>3.04E+05</td>
<td>1.17x</td>
<td>1.17x</td>
<td>1.34x</td>
<td>8.65x</td>
</tr>
<tr>
<td>bzip2</td>
<td>111.91</td>
<td>1.02E+05</td>
<td>0.59x</td>
<td>0.94x</td>
<td>1.12x</td>
<td>8.03x</td>
</tr>
<tr>
<td>gcc</td>
<td>44.61</td>
<td>8.15E+04</td>
<td>0.60x</td>
<td>0.75x</td>
<td>1.37x</td>
<td>8.64x</td>
</tr>
<tr>
<td>h264ref</td>
<td>309.12</td>
<td>1.50E+05</td>
<td>0.56x</td>
<td>0.79x</td>
<td>1.36x</td>
<td>11.3x</td>
</tr>
<tr>
<td>hmmer</td>
<td>323.32</td>
<td>1.06E+05</td>
<td>0.87x</td>
<td>1.03x</td>
<td>1.32x</td>
<td>29.4x</td>
</tr>
<tr>
<td>libquantum</td>
<td>402.38</td>
<td>2.15E+05</td>
<td>1.19x</td>
<td>1.22x</td>
<td>1.35x</td>
<td>11.9x</td>
</tr>
<tr>
<td>mcf</td>
<td>315.37</td>
<td>6.58E+04</td>
<td>0.50x</td>
<td>0.65x</td>
<td>1.19x</td>
<td>11.0x</td>
</tr>
<tr>
<td>Xalan</td>
<td>352.29</td>
<td>4.25E+04</td>
<td>0.53x</td>
<td>0.69x</td>
<td>1.18x</td>
<td>11.1x</td>
</tr>
<tr>
<td>ROSE</td>
<td>319.97</td>
<td>6.58E+04</td>
<td>0.50x</td>
<td>0.65x</td>
<td>1.19x</td>
<td>11.0x</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>79.28</td>
<td>1.16E+05</td>
<td>0.94x</td>
<td>1.03x</td>
<td>1.32x</td>
<td>29.4x</td>
</tr>
<tr>
<td>LULESH</td>
<td>67.29</td>
<td>2.09E+04</td>
<td>0.46x</td>
<td>0.55x</td>
<td>1.19x</td>
<td>11.1x</td>
</tr>
<tr>
<td>GeoMean</td>
<td>219.97</td>
<td>3.05E+04</td>
<td>0.54x</td>
<td>0.68x</td>
<td>1.16x</td>
<td>10.9x</td>
</tr>
</tbody>
</table>

Table 4: Memory overhead of CCTLib.

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Original resident memory in MB</td>
<td>Max call paths</td>
<td>Context-centric analysis</td>
<td>Data-centric analysis</td>
<td>Overhead (tree-based)</td>
<td>Overhead (shadow memory)</td>
</tr>
<tr>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>astar</td>
<td>230</td>
<td>2.64E+05</td>
<td>0.13x</td>
<td>0.13x</td>
<td>0.14x</td>
<td>0.10x</td>
</tr>
<tr>
<td>bzip2</td>
<td>591</td>
<td>9.12E+04</td>
<td>0.13x</td>
<td>0.13x</td>
<td>0.13x</td>
<td>0.10x</td>
</tr>
<tr>
<td>gcc</td>
<td>456</td>
<td>8.10E+08</td>
<td>0.16x</td>
<td>0.16x</td>
<td>0.16x</td>
<td>0.10x</td>
</tr>
<tr>
<td>h264ref</td>
<td>31</td>
<td>1.30E+06</td>
<td>0.23x</td>
<td>0.23x</td>
<td>0.23x</td>
<td>0.10x</td>
</tr>
<tr>
<td>hmmer</td>
<td>15</td>
<td>2.50E+05</td>
<td>0.38x</td>
<td>0.38x</td>
<td>0.38x</td>
<td>0.10x</td>
</tr>
<tr>
<td>libquantum</td>
<td>96</td>
<td>1.02E+05</td>
<td>0.18x</td>
<td>0.18x</td>
<td>0.18x</td>
<td>0.10x</td>
</tr>
<tr>
<td>mcf</td>
<td>167</td>
<td>7.90E+05</td>
<td>0.48x</td>
<td>0.48x</td>
<td>0.48x</td>
<td>0.10x</td>
</tr>
<tr>
<td>Xalan</td>
<td>170</td>
<td>3.56E+06</td>
<td>0.98x</td>
<td>0.98x</td>
<td>0.98x</td>
<td>0.10x</td>
</tr>
<tr>
<td>ROSE</td>
<td>149</td>
<td>8.58E+08</td>
<td>0.28x</td>
<td>0.28x</td>
<td>0.28x</td>
<td>0.10x</td>
</tr>
<tr>
<td>LAMMPS</td>
<td>101</td>
<td>1.01E+06</td>
<td>0.90x</td>
<td>0.90x</td>
<td>0.90x</td>
<td>0.10x</td>
</tr>
<tr>
<td>LULESH</td>
<td>26</td>
<td>2.84E+05</td>
<td>0.27x</td>
<td>0.27x</td>
<td>0.27x</td>
<td>0.10x</td>
</tr>
<tr>
<td>GeoMean</td>
<td>-</td>
<td>-</td>
<td>0.49x</td>
<td>0.49x</td>
<td>0.49x</td>
<td>0.10x</td>
</tr>
</tbody>
</table>

5.2 Memory overhead on serial codes

Table 4 shows the results of memory overhead introduced by CCTLib. Column 3 shows the maximum number of call paths collected in each application. CCTLib collected ~1.49 billion call paths during an execution of the ROSE compiler. When tracking memory access instructions, CCTLib requires 3.49× extra memory on average for storing the CCTs. When tracking all instructions, CCTLib requires 4× extra memory on average. When performing data-centric analysis via balanced-tree-based technique, CCTLib requires 4.38× extra memory on average. When performing data-centric analysis via shadow-memory-based technique, CCTLib requires 16.9× extra memory on average. Column 6 and Column 7 subsume the overhead in Column 5 since they include the overhead of CCT construction for each instruction.

There is a correlation between the number of call paths collected and the memory overhead of call path collection. Applications such as **gcc**, **Xalan**, and **ROSE** have deep recursion leading to same instructions being repeated in multiple calling contexts, consequently they have a higher memory overhead. If we ignore **gcc**, **Xalan**, and **ROSE**, the geometric mean for columns 4, 5, 6, and 7 are respectively 1.71×, 1.77×, 1.99×, and 11.4×.

Our 40-bit **DataHandle_t** structures are word aligned to 64 bits, causing about 8× memory overhead for shadow-memory-based data-centric analysis. One can use packed structures and trade off memory for runtime overhead. We did not explore that option in our evaluation. For the **hmmer** benchmark, the memory overhead appears to be higher for shadow-memory-based data-centric analysis; this is because the application has a very small memory footprint (15MB), and the preallocated page tables (8MB) used in the shadow memory technique [8] skew the numbers.

5.3 Scalability on parallel applications

To evaluate CCTLib’s scalability, we perform strong scaling experiments with the OpenMP versions of LULESH and LAMMPS. We varied the number of threads from 1 to 32 and evaluated the slowdown due to CCTLib when com-
Table 5: Parallel efficiency of CCTLib.

<table>
<thead>
<tr>
<th></th>
<th>LAMMPS no. threads</th>
<th>LULESH no. threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Original running time in seconds</td>
<td>174.75</td>
<td>95.32</td>
</tr>
<tr>
<td>Original prog. parallel efficiency</td>
<td>100%</td>
<td>91%</td>
</tr>
<tr>
<td>Call path analysis</td>
<td>Overhead</td>
<td>22×</td>
</tr>
<tr>
<td></td>
<td>Scalability</td>
<td>100%</td>
</tr>
<tr>
<td>balanced-tree-based</td>
<td>Overhead</td>
<td>80×</td>
</tr>
<tr>
<td>data-centric analysis</td>
<td>Scalability</td>
<td>100%</td>
</tr>
<tr>
<td>Shadow-memory-based</td>
<td>Overhead</td>
<td>32×</td>
</tr>
<tr>
<td>data-centric analysis</td>
<td>Scalability</td>
<td>100%</td>
</tr>
</tbody>
</table>

We define $O(n)$—the overhead in an $n$-thread execution, as the ratio of $R_c(n)$—the running time of CCTLib in an $n$-thread execution, to $R_o(n)$—the running time of the original $n$-thread execution. We define $S(n)$—the percent parallel scalability of $n$-thread execution of CCTLib, as a scaled ratio of the overhead of a 1-thread execution to the overhead of an $n$-thread execution.

$$O(n) = \frac{R_c(n)}{R_o(n)}$$

$$S(n) = \frac{O(1)}{O(n)} \times 100$$

High scalability is best. The results are tabulated in Table 5.

We make the following observation about CCTLib:

- For call path collection, the overhead remains fairly stable (22×–27×) for LAMMPS and achieves 96% scalability. For LULESH, the overhead decreases quite dramatically with increased parallelism. This is because of the Amdahl’s law, since LULESH has poor parallel efficiency, injecting the scalable CCTLib component into it causes CCTLib’s overhead to scale superlinearly (586%).

- Data-centric attribution using our concurrent balanced-tree-based technique has higher overhead for LAMMPS (about 80×); however, the overhead remains stable and achieves perfect scaling (96%). For LULESH, the overhead steadily reduces with the increased level of parallelism reaching 60% scalability at 32 threads. Since the balanced-tree-based technique has a lower memory footprint, it might be preferred over our shadow-memory-based technique when the concurrency level is higher and the application has poor scalability.

- Data-centric attribution using our shadow-memory-based technique is significantly faster than our balanced-tree-based technique and it scales well. On LAMMPS, the technique introduces $31 \times 38 \times$ slowdown and has 95% scalability. For LULESH, the trend is similar to that of the call path collection, lowering overhead to 8× with the increase in the number of threads causing it to attain 408% scalability.

### 6. CONCLUSIONS AND FUTURE WORK

Fine-grained instruction-level execution monitoring is frequently employed in performance analysis, software correctness, security and other domains. Intel’s Pin is a leading tool for developing FEM tools. Pin does not have any built-in capability for call path collection. The availability of calling contexts enriches a tool by enabling it to provide more detailed diagnostic feedback. In this paper, we introduced CCTLib, an open-source library for Pin tools that collects call paths attributing each instruction to full calling contexts. CCTLib not only collects call paths but also attributes each memory access to the corresponding data object. Compared to state-of-the-art open-source Pin tools for call path collection, CCTLib is richer in information, more accurate even for programs with complex control flow and does so with about 30% less overhead—a difference of 14× on average. We demonstrated that, with CCTLib, one can collect call paths and attribute memory accesses to data objects on every executed machine instruction even for reasonably long running program. CCTLib has modest runtime overhead and scales well when used on multithreaded codes.

Our choice of algorithms and data structures within CCTLib makes such a heavy-weight task affordable, efficient, and scalable.

In the future, we would like to support attaching CCTLib to a running process to gather call paths. We would like to employ CCTLib on state-of-the-art Pin tools to enhance their capabilities with context- and data-centric information. Ultimately, we would like call path collection to be an integral part of Pin.

### 7. ACKNOWLEDGMENTS

This work was supported in part by the DOE Office of Science through cooperative agreements DE-FC02-13ER26144 and DE-FC02-12ER26119. This research used the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097.

### 8. REFERENCES


