HP Caliper: A Framework for Performance Analysis Tools

In recent years, the computing community has developed a strong set of tools to help analyze runtime behavior. Statistical sampling and binary instrumentation maintain positions as the two most favored techniques.

You perform statistical sampling by taking periodic snapshots of a program’s state. Statistical sampling is nonintrusive—unlike binary instrumentation, statistical sampling doesn’t add any lines of code to the application being tested—but the computing community generally regards this technique as imprecise. It imposes low overhead on a program’s runtime performance and can be used for time-critical experiments, but the measurements derived from statistical sampling can contain imprecisions. For example, without special hardware support, a sampled instruction pointer may not be related to the instruction address that caused a particular sampling event.1,2

Dynamic binary instrumentation, on the other hand, lets program instructions be changed on the fly, which can lead to more precise analysis. But because dynamic instrumentation modifies a program’s code—a program’s cache and paging behavior, for example—execution time can slow significantly, even up to four times the normal execution speed. You typically perform dynamic instrumentation at runtime, and you only test those parts of a program that will actually execute. Doing so minimizes the overhead of the instrumentation process itself.

Analysis tools based on dynamic instrumentation—see the “Related work” sidebar for information about other tools and systems—require no special executable preparation. HP Caliper, a framework for building dynamic instrumentation tools, lets you change program instructions on the fly with instrumentation probes. Caliper integrates performance measurement unit (PMU) sampling with dynamic instrumentation. This article describes Caliper’s architecture, its public interfaces, and its dynamic instrumentation algorithm.

While Caliper will eventually run on several different platforms, this article specifically describes Caliper’s implementation on HP-UX powered by the IA-64 Itanium processor. The IA-64’s instruction set architecture (ISA) offers a set of architectural features designed to create synergy between compiler and processor. Caliper’s architecture takes advantage of these features.
Related work

In "Shade: A Fast Instruction-Set Simulator for Execution Profiling," Robert Cmelik and David Keppel present a list of over 45 hardware emulators, decode-and-dispatch interpreters, pre-decode interpreters working on intermediate representations, static cross compilers, and dynamic cross compilers. These tools differ from Caliper in support for kernel code, time of instrumentation, requirements for debug information, and support for signals and multithreaded programs.

Caliper's architecture

As Figure 1 illustrates, the Caliper architecture divides developer tools based on Caliper into two parts, the user interface and the Caliper shared object called libcaliper.so. User interfaces can be stand-alone scripts or complete integrated development environments (IDEs). As Caliper’s main component, the shared object contains support code, collectors, the Caliper API, and memory management routines. It integrates a Python interpreter and provides a default C main function. Caliper’s main architectural blocks. The interfaces, although written in ANSI C, follow object-oriented design principles and form a simple object model consisting of measurement sets, event objects, process interfaces, configuration parameters, context parameters, and collector modules.

C FUNCTION INTERFACES

The Caliper API consists of a set of C function interfaces to Caliper’s main
specification and combination. Instrumentation-based measurements include function coverage and counting, basic block coverage and counting, arc counts, and call graphs. PMU-based global performance metrics include control speculation miss ratio, data speculation miss ratio, Advanced Load Address Table capacity miss ratio, data and instruction cache miss ratios, Translation Lookaside Buffer miss ratios, and more.

Event objects handle event queues and deal with application and user events. Typical program events include process creation and destruction, shared object loading and unloading, timer expiration, PMU counter overflow, and process termination.

Process interfaces handle all process-related events, such as signals, and allow the creation of or attachment to a process. Process interfaces let you control processes via the OS’s debug and performance measurement interfaces.

Configuration parameters let you set several key variables, such as the size of the shared memory blocks.

Context parameters let you scale Caliper to large applications by changing measurements in both time and space. A context’s three dimensions are:

- **address**, to include or exclude modules, functions, and address ranges;
- **time**, to schedule measurements; and
- **event**, to specify program actions for specific program events.

Using the infrastructure Caliper offers, collector modules perform special kinds of measurement, such as PMU sampling or instrumentation-based function counting. Each collector adds an individual API to the Caliper API to interact with the developer tool.

SUPPORT LIBRARY

The Caliper support library offers a framework of services and tools for dynamic instrumentation and sampling. These services include:

- encoding and decoding machine instructions to an intermediate representation (IR) with automatic fix-up of IP-relative branches;
- handling an executable’s ELF file, code, data segments, debug/unwind information, compiler annotations, and function tables; and
- managing data exchange between Caliper and its monitored processes for counters, events, control instructions, or other applications.

INTERPRETER AND WRAPPER

A developer tool communicates with Caliper via the Caliper API or the integrated Python interpreter, which can perform multiple tasks. The Python interpreter contains wrappers for all API functions and can be used to interpret initialization and configuration scripts. The interpreter acts as the main interface for all command-line tools and also as the main shell for the integrated debugger, called cdb. Currently, Python cannot be used to describe probe code sequences at a high level.

The API itself is defined in a set of C header files, which are processed by the Software Wrapper Interface Generator (SWIG) to generate Python wrapper code. The Caliper shared object includes the interpreter and the wrappers. Because the generated wrapper functions are too complex to be usable, we developed a Python class library based on the SWIG-generated interfaces. These more intuitive classes serve as the main scripting interface to Caliper.

PROGRAMMING MODEL

Because the Caliper programming model I’ve described so far was still too complex for simple and standardized tasks, the Caliper team developed a simplified model that requires only the most basic user control. In this model, only a few variables can be assigned before a measurement starts, and all other details are hidden.

Tools using Caliper can access the C API, the Python SWIG-generated functions, the Python class library, and the simplified layer. They can also operate Caliper from an IDE. Other language interfaces, such as C++ or Java, can be added on top of the C API.

COMMAND-LINE DRIVER

As shown in Figure 2, the Caliper shared object needs only a small, simple
driver to perform useful work. This kind of command-line driver simply runs a script file specified on the command line.

A control file in the simplified model looks like the following sample:

```
# specify application
application = "a.out"

# specify output file
pbo_out = "flow.dat"

# run collector
collect(pbo)
```

**Dynamic instrumentation**

One of the major benefits of using dynamic instrumentation, as opposed to static instrumentation, is scalability. According to the 80:20 rule—in a typical program, 80% of the runtime is spent in 20% of the code—only a small fraction of an executable system must be instrumented to detect the most significant parts of a program.

Dynamic instrumentation can be performed in several ways. The two strategies we considered for generating probe code with Caliper included making use of trampolines (out of line) and inlining and relocating probe code (inline). As an example, an out-of-line instrumentation strategy may perform code transformations like the following example. A function's entry point—in this case, foo's—would look like this in IA-64 Assembly:

```
foo::
  alloc r33=ar.pfs,0,11,1,0
  addl r9=-2944,r1
  addl r8=-2936,r1
foo'::
  ...
```

To perform function invocation counting, the out-of-line strategy would instrument foo's entry point with a long branch to a trampoline that executes the original instruction plus some additional code to update an invocation counter (see Figure 3a).

There are, of course, many ways to encode, reach, and return from the actual trampoline code. Care must be taken for code with branch instructions in the first bundle of a function. You have to use long branches when the trampoline code isn't reachable from the original code using the 25-bit encoded offset of the IP-relative branch instruction on the IA-64.

Using long branches offers several advantages. If you place the probe code out of line and branch the instrumented instructions to it, then the counting code won't cause any wrinkles in the original application's address space and all branches would reach their designated targets.

The other major strategy we considered is to inline and relocate code. The code snippet in Figure 3a for the function foo would be transformed into the instruction sequence shown in Figure 3b.

Inlining and relocating leads to more compact code, less intrusion, and better performance, but it comes at a price. Inserting probe code changes the relative offsets in a code stream and requires translating indirect branch targets that the instrumenuter cannot determine. Combining different instrumentations and probe code is not as easy as it is in the well-defined, sand-box style trampoline approach.

In “Adaptable Binary Programs,” Susan Graham, Steven Lucco, and Robert Wahbe found the relative overhead associated with the inline and out-of-line instrumentation strategies to be 34% for in-line and 112% for out-of-line strategies. They computed transformation overhead as the runtime of all code added to the application in order to support the primary probe code, without including the probe code itself.

We also needed to minimize the use of long branches because the first versions of the IA-64 only emulated long branch, which caused additional runtime performance slow-downs. A trampoline-based instrumentation approach with out-of-line branches made heavy use of long branches; we disregarded it in favor of the current inline approach.

Preliminary measurements on HP-UX showed us that the overhead of a long call branch is approximately 100 to 300 cycles, which is a small number for an emulated instruction. It let us use the
long branch instruction occasionally in the algorithm.

The inlining relocation method is faster even without considering the extra cost of an emulated long branch instruction, which justified our decisions about Caliper’s algorithm in light of an upcoming, hardware-supported long branch instruction.

Core algorithm

Caliper works at the level of functions by inlining probes into functions and relocating the instrumented functions. The dynamic instrumentation algorithm performs the following five steps, encapsulated in the Caliper API:

1. **Attach and inject.** Caliper identifies a process and attaches to it using the HP-UX ttrace system call. When the process stops, Caliper gains control, injects code into the process to allocate shared memory, and adds runtime libraries to support dynamic instrumentation.

2. **Function discovery.** You can identify function entry points by analyzing the unwind information tables (sometimes called exception tables), the procedure lookup tables, and the symbol table. Caliper does not depend on debug information to perform this step. The analysis may still miss some function entry points because of a lack of unwind information and symbolic information. But Caliper will discover these functions dynamically (as described in Step 4).

3. **Static break insertion.** Caliper patches every function’s entry point with a break instruction.

4. **Run under dynamic instrumentation.** Caliper transfers control back to the process. The process runs until it hits one of the inserted break instructions at the entry point of a function. Because ttrace controls the process, control transfers to Caliper and the instrumentation process begins at the current function. After instrumentation, control transfers to the instrumented function, which continues to run until it hits the next break instruction. Again, control transfers to Caliper, which resumes the dynamic instrumentation process.

5. **Output.** Upon process termination or user request, control transfers to Caliper. At this point you can now retrieve statistics, counters, and other measurement results in one of the integrated, collector-specific formats or via user-defined, script-based output routines.

This dynamic instrumentation algorithm could rightfully be characterized as a lazy instrumentation algorithm. If a program were to consist of only one, presumably huge function $F$, the algorithm would instrument the whole pro-

![Figure 3](image)
program at once after reaching F's entry point, and it wouldn't perform code transformations that depend on information available at runtime.

We don't expect programs consisting of only one function to be prevalent in today’s computing environments. The instrumentation sequence depends on the dynamic control flow of a program and can be changed interactively or via the definition of context, which is why we continue to use the term *dynamic instrumentation* to describe this algorithm.

**IA-64 implementation and ongoing development**

The IA-64 processor contains a high-performance register stack engine (RSE) that helps minimize the cost of creating a call frame and a function call by maintaining a separate register stack. If a programming model requires consistent unwinding of the stack—during a C++ exception, for example—both program stack and register stack have to be unwound.

**Unwind information**

Caliper generates unwind information for every region in a program and stores it in the text segment for fast access. The IA-64 architecture requires the presence of unwind information. If code motion happens during instrumentation, the unwind information has to be dynamically updated. Dynamically updating unwind information is no easy task, because regions get modified by probe code inlining. We haven’t yet fully resolved unwind information updating, which bars Caliper from being used for analysis of C++ programs that make use of C++ exceptions.

Compilers frequently translate a C/C++ switch statement into an indirect branch based on a branch table located in the code segment. It is generally impossible for a binary code analyzer to decide whether a given address contains data (such as an entry of a branch table) or real code.

Some algorithms exist to identify branch tables. For some code-generation schemes, this problem can always be solved. Caliper uses compiler-generated annotations residing in an executable's ELF file to identify these tables. If Caliper has identified a branch table, the algorithm will modify the table entries to point to their corresponding instrumented target addresses. Caliper links annotations to the unwind information for a given region. As long as the executable contains unwind information, Caliper can find the corresponding annotations independently of whether or not the executable contains debug information.

**Preserved semantics**

The foremost requirement for binary instrumentation is, of course, to preserve the program semantics at any given time. Caliper uses a staged method to find free registers. First, Caliper identifies free registers with the assistance of compiler-generated annotations. If it doesn’t find annotations, it creates free registers by increasing the number of a function’s stacked output registers.

This process will fail if the function has multiple alloc instructions or has already allocated all stacked registers. In these cases, you have to explicitly spill to or fill the program stack.

**Multithreaded applications**

Multithreaded applications presented a new kind of challenge for Caliper. Inserting an instruction (like a long branch) turned out to be more complicated than we expected. The IA-64 bundle size is 16 bytes, but load and store instructions only operate on a maximum of 8 bytes. This meant that two store instructions would be necessary to update a bundle.

Multithreaded applications present two potentially hazardous scenarios. First, it is possible that a thread could hit a bundle in the middle of its update process. This would cause a half-deployed instruction with an invalid instruction template field, which results in a signal. Second, a thread could stall on slot one or slot two of a bundle, waking up on a changed instruction, and ultimately resulting in incorrect program behavior.

We solved the latter scenario in Caliper using a sequence of update steps. The first problem requires the installation of a signal handler for invalid template exceptions. We haven’t yet implemented this feature. To date, Caliper simply halts all threads in the target application while performing an instruction update. While guaranteeing program correctness, this method slows down execution speed, especially on multiprocessor systems. For this reason, we plan to add full support for multithreading.

**Call shadows**

The IA-64 supports call shadows where two branches are located in one bundle, as in the following example:

```
nop.m 0
(p6)br.call.dptk.few -0x150
br.call.sptk.few -0x410;
```

If the predicate register p6 is set to one, the first branch instruction executes. Because branch targets and return address are always full bundle addresses on the IA-64, the second branch won’t be executed.

If you instrument this instruction sequence and insert counting code, the original instructions may get dispersed across multiple bundles, changing the implicit logic of the call shadow that suppressed the second branch. This is why Caliper performs an additional search for call shadows and alters the instrumentation sequences accordingly.

**Integrated debugging**

Because of operating system limitations and because of conflicting break instruction use, you cannot use standard debuggers to debug Caliper-controlled applications. We integrated debugging functionality into Caliper and named it cdb (Caliper debugger). Debugging became invaluable for identifying program flaws, invalid probe code sequences, kernel bugs, and missing stop bits in the instrumented code.

Cdb makes use of the integrated Python interpreter to display a prompt, to parse commands, and to perform actions accordingly. It supports insertion of break points, single stepping, disassembly of original and instrumented code, dumping of data and registers, and
more. cdb also uses Python’s support for socket communication to enable remote debugging. Caliper allows falling back to a cdb prompt whenever an unexpected situation (or signal) occurs.

LIMITATIONS
Caliper does have some limitations. It doesn’t instrument dynamically generated code and cannot handle programs that internally change between little-endian and big-endian or that make use of IP-aware signal handlers. It is also possible to create assembler code sequences where instrumentation will fail—code performing label arithmetic, for example—but you would use such sequences only rarely, if at all.

TIMING MEASUREMENTS
Currently, Caliper can instrument the first 18 Spec2000 benchmark programs and some commercial applications. Tools built on Caliper include the profiling tool gprof, a PBO tool that generates input files for the optimizing compiler, a function coverage tool, and a predicate hazard checker. Predicate hazard checking is an interesting application of the Caliper framework. The HP compiler optimization group developed an algorithm that places instructions in the same issue group, although they may have a resource conflict, as in the following bundle with a read-after-write conflict:

\[
\text{nop.m (p35) addl } r14=0x40784634, r0 \\
\text{ld4.s } r15=[r14]
\]

The instructions, however, are predicted. If you can guarantee that the predicates are never one at the same time, then predicate hazard checking becomes a powerful optimization technique.

To verify the algorithm, the optimization group uses a static tool to read ELF executables and to output potential hazards. The static tool can indicate hundreds and thousands of potential hazards. You can then check this information manually against disassembled code and run it through other static analyzers. Still, there has to be some form of dynamic verification. If we found one single occurring hazard, we could prove that the algorithm had a flaw for a given input stream.

To support our compiler optimization team, we wrote a collector that reads in the output of the static hazard analyzer and instruments functions containing potential hazards. The probe code sequences check whether or not two indicated predicate registers are both set to one at a questionable address. If a single counter has a value of one or greater, the hazard exposes itself.

Table 1 presents timing measurements for Caliper performing function counting on several Spec2000 programs and on a commercial, nonlinear finite-element solver (FinElem). The program consists of roughly 970,000 lines of C, Fortran, and Assembly.

The overhead caused by additional probe code instructions executed for function counting is relatively small but grows for very short functions. Table 1 illustrates the combined relative cost of startup, analysis, instrumentation, and execution of probe code and long

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>SRC</th>
<th>DATA</th>
<th>RUN</th>
<th>INSTR</th>
<th>OVER (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec-mgrid</td>
<td>F77</td>
<td>train</td>
<td>327.2</td>
<td>330.8</td>
<td>1.1</td>
</tr>
<tr>
<td>Spec-facerc</td>
<td>F90</td>
<td>test</td>
<td>248.2</td>
<td>254.4</td>
<td>2.5</td>
</tr>
<tr>
<td>Spec-gzip</td>
<td>C</td>
<td>train</td>
<td>374.4</td>
<td>397.5</td>
<td>6.2</td>
</tr>
<tr>
<td>Spec-vpr l</td>
<td>ref</td>
<td></td>
<td>652.3</td>
<td>711.9</td>
<td>9.1</td>
</tr>
<tr>
<td>Spec-swim</td>
<td>F77</td>
<td>train</td>
<td>139.6</td>
<td>156.5</td>
<td>12.1</td>
</tr>
<tr>
<td>Spec-ammp</td>
<td>C</td>
<td>train</td>
<td>425.1</td>
<td>476.5</td>
<td>12.1</td>
</tr>
<tr>
<td>Spec-galgel</td>
<td>F90</td>
<td>train</td>
<td>272.8</td>
<td>307.0</td>
<td>12.5</td>
</tr>
<tr>
<td>Spec-vpr l</td>
<td>C</td>
<td>train</td>
<td>73.9</td>
<td>84.3</td>
<td>14.1</td>
</tr>
<tr>
<td>FinElem</td>
<td>C/F/A</td>
<td>exampl9</td>
<td>7,334.7</td>
<td>9,019.6</td>
<td>23.0</td>
</tr>
<tr>
<td>Spec-twolf</td>
<td>C</td>
<td>train</td>
<td>51.8</td>
<td>78.3</td>
<td>51.2</td>
</tr>
<tr>
<td>Spec-mesa</td>
<td>C</td>
<td>train</td>
<td>11.9</td>
<td>21.5</td>
<td>80.7</td>
</tr>
<tr>
<td>Spec-eon</td>
<td>C++</td>
<td>train</td>
<td>727.7</td>
<td>2,651.7</td>
<td>264.4</td>
</tr>
</tbody>
</table>

Table 2. Instrumentation overhead statistics for several Spec2000 programs and FinElem, a commercial finite-element solver. Overhead comparison is sorted by total number of functions (Funcs), the number of executed functions (Reached), the percentage of reached functions out of the total number of functions (average is 26%), the number of functions not instrumented (NI), and the percentage of NI functions out of the total number of functions.

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>FUNCS</th>
<th>REACHED</th>
<th>%</th>
<th>NI</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec-mgrid</td>
<td>601</td>
<td>144</td>
<td>24.0</td>
<td>4</td>
<td>2.8</td>
</tr>
<tr>
<td>Spec-facerc</td>
<td>683</td>
<td>209</td>
<td>30.6</td>
<td>8</td>
<td>3.8</td>
</tr>
<tr>
<td>Spec-gzip</td>
<td>439</td>
<td>107</td>
<td>24.4</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>Spec-vpr l</td>
<td>ref</td>
<td>565</td>
<td>210</td>
<td>37.2</td>
<td>1</td>
</tr>
<tr>
<td>Spec-swim</td>
<td>598</td>
<td>139</td>
<td>23.2</td>
<td>4</td>
<td>2.9</td>
</tr>
<tr>
<td>Spec-ammp</td>
<td>483</td>
<td>138</td>
<td>28.6</td>
<td>1</td>
<td>0.7</td>
</tr>
<tr>
<td>Spec-galgel</td>
<td>765</td>
<td>269</td>
<td>35.2</td>
<td>6</td>
<td>2.2</td>
</tr>
<tr>
<td>Spec-vpr l</td>
<td>train</td>
<td>565</td>
<td>209</td>
<td>37.0</td>
<td>1</td>
</tr>
<tr>
<td>FinElem</td>
<td>5,073</td>
<td>583</td>
<td>11.5</td>
<td>27</td>
<td>4.6</td>
</tr>
<tr>
<td>Spec-twolf</td>
<td>496</td>
<td>210</td>
<td>42.3</td>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>Spec-mesa</td>
<td>1,391</td>
<td>243</td>
<td>17.5</td>
<td>11</td>
<td>4.5</td>
</tr>
<tr>
<td>Spec-eon</td>
<td>2,889</td>
<td>613</td>
<td>21.2</td>
<td>302</td>
<td>49.3</td>
</tr>
</tbody>
</table>
branch instructions. The cost of the instrumentation process is significant and long-running applications like Spec-mgrid show a smaller relative overhead compared to short-running applications like Spec-mesa.

A notable example is the C++ application Spec-con, where instrumentation results in a huge overhead. This behavior can be explained by looking at Table 2, which lists the total number of functions, the number of executed functions, and the number of noninstrumented functions for each of the test cases.

Because of a temporary Caliper limitation, we didn’t instrument nearly half of Spec-con’s functions. This resulted in a large number of long branches between instrumented and original code.

**WE PLAN TO PORT**

Caliper to HP-UX on PA-RISC. PA-RISC’s ISA is fairly similar to the IA-64’s. The success of HP’s Aries emulator running PA-RISC applications on IA-64 demonstrates this. But there will be two main problems with porting Caliper to PA-RISC:

- There is no PMU or equivalent hardware on PA-RISC, which means that Caliper for PA-RISC will focus on binary code instrumentation.
- Caliper exploits two instructions unique to the IA-64: the long branch instruction (brl) and the memory access synchronizing instruction (fetchadd) for counter updates. For both instructions there is no equivalent on PA-RISC, which means we must develop workarounds.

To support analysis of C++ programs that make use of exceptions, we also plan to continue developing the dynamic updating of unwind information. We plan to develop more tools on top of the Caliper instrumentation framework. These will include basic block-related tools and API checkers such as a memory-leak detection tool and a pthread correctness checker.

Today, Caliper is still in beta testing, and is mainly used within HP. But with the launch of HP’s IA-64 program in the near future, we will officially release Caliper in its first full-point version. We anticipate that Caliper’s role in the future of development technology will grow because of the complexity of the IA-64 processor, the need for proper runtime analysis, and the growth and evolution of the IA-64 program itself.

**Acknowledgments**

Caliper is the result of a strong team effort within HP. I thank Dave Babcock, Eric Gouvier and German Rivera for their contributions. Thanks to Umesh Krishnaswamy and Thomas Lofgren for their additional help with this article. I especially thank Vinodha Ramasamy for major work on design and implementation of the dynamic instrumentation algorithm. Finally, I thank the anonymous reviewers for their invaluable feedback.

**References**


Robert Hundt is a software development engineer at Hewlett Packard. He has a diploma in computer science from Technical University, Munich. His research interests include code transformation, instrumentation, and optimization. He is a member of the IEEE and Usenix. Contact him at Hewlett-Packard Company, 19111 Pruneridge Ave., ms42u4, Cupertino, Calif. 95014; robert_hundt@hp.com.