Profiling (computer programming)

From Wikipedia, the free encyclopedia

In software engineering, **profiling** ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

Profiling is achieved by instrumenting either the program source code or its binary executable form using a tool called a *profiler* (or *code profiler*). Profilers may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods.

Contents [hide]

Program execution

General concepts
- Code
- Translation
  - Compiler
    - Compile-time
  - Optimizing compiler
- Intermediate representation (IR)
- Execution
  - Runtime system
    - Runtime
  - Executable
  - Interpreter
  - Virtual machine

Types of code
- Source code
- Object code
- Bytecode
- Machine code
- Microcode

Compilation strategies
- Just-in-time (JIT)
  - Tracing just-in-time
- Ahead-of-time (AOT)
- Transcompilation
- Recompilation

6 more
Edit links

Print/export
Download as PDF
Printable version

Notable runtimes
- Android Runtime (ART)
- Common Language Runtime (CLR) & Mono
- crt0
- HHVM
- Java virtual machine (JVM)
- Objective-C
- V8
  - Node.js
- PyPy
- Zend Engine

Notable compilers & toolchains
- GNU Compiler Collection (GCC)
- LLVM
  - Clang

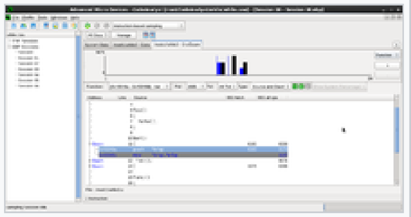
V · T · E

Gathering program events [edit]

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. Profilers are used in the performance engineering process.

Use of profilers [edit]

> Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing...
>
> — ATOM, PLDI, '94


Graphical output of the CodeAnalyst profiler.

The output of a profiler may be:

- A statistical *summary* of the events observed (a **profile**)

  Summary profile information is often shown annotated against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program.

```
/* ------------ source------------------------- count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

- A stream of recorded events (a **trace**)

  For sequential programs, a summary profile is usually sufficient, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening.

  The size of a (full) trace is linear to the program's instruction path length, making it somewhat impractical. A trace may therefore be initiated at one point in a program and terminated at another point to limit the output.

- An ongoing interaction with the hypervisor (continuous or periodic monitoring via on-screen display for instance)

  This provides the opportunity to switch a trace on or off at any desired point during execution in addition to viewing on-going metrics about the (still executing) program. It also provides the opportunity to suspend asynchronous processes at critical points to examine interactions with other parallel processes in more detail.

A profiler can be applied to an individual method or at the scale of a module or program, to identify performance bottlenecks by making long-running code obvious.[1] A profiler can be used to understand code from a timing point of view, with the objective of optimizing it to handle various runtime conditions[2] or various loads.[3] Profiling results can be ingested by a compiler that provides profile-guided optimization.[4] Profiling results can be used to guide the design and optimization of an individual algorithm; the Krauss matching wildcards algorithm is an example.[5] Profilers are built into some application performance management systems that aggregate profiling data to provide insight into transaction workloads in distributed applications.[6]

History [edit]

Performance-analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the program status word (PSW) at set timer-intervals to detect "hot spots" in executing code.[*citation needed*] This was an early example of sampling (see below). In early 1974 instruction-set simulators permitted full trace and other performance-monitoring features.[*citation needed*]

Profiler-driven program analysis on Unix dates back to 1973,[7] when Unix systems included a basic tool, `prof`, which listed each function and how much of program execution time it used. In 1982 `gprof` extended the concept to a complete call graph analysis.[8]

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM[9](Analysis Tools with OM). The ATOM platform converts a program into its own profiler: at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "instrumentation".

In 2004 both the `gprof` and ATOM papers appeared on the list of the 50 most influential PLDI papers for the 20-year period ending in 1999.[10]

Profiler types based on output [edit]

Flat profiler [edit]

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph profiler [edit]

Call graph profilers[8] show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. In some tools full context is not preserved.

Input-sensitive profiler [edit]

Input-sensitive profilers[11][12][13] add a further dimension to flat or call-graph profilers by relating performance measures to features of the input workloads, such as input size or input values. They generate charts that characterize how an application's performance scales as a function of its input.

Data granularity in profiler types [edit]

Profilers, which are also programs themselves, analyze target programs by collecting information on their execution. Based on their data granularity, on how profilers collect information, they are classified into event based or statistical profilers. Profilers interrupt program execution to collect information, which may result in a limited resolution in the time measurements, which should be taken with a grain of salt. Basic block profilers report a number of machine clock cycles devoted to executing each line of code, or a timing based on adding these together; the timings reported per basic block may not reflect a difference between cache hits and misses.[14][15]

Event-based profilers [edit]

The programming languages listed here have event-based profilers:

- Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
- .NET: Can attach a profiling agent as a *COM* server to the *CLR* using Profiling *API*. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can rewrite the target application's bytecode in arbitrary ways.
- Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile' function to trap events like c_{call,return,exception}, python_{call,return,exception}.
- Ruby: Ruby also uses a similar interface to Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical profilers [edit]

Some profilers operate by sampling. A sampling profiler probes the target program's call stack at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.

The resulting data are not exact, but a statistical approximation. "The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods." [16]

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of CPU cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

Dedicated hardware can go beyond this: ARM Cortex-M3 and some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the program counter in a truly undetectable manner, allowing non-intrusive collection of a flat profile.

Some commonly used[17] statistical profilers for Java/managed code are SmartBear Software's AQtime[18] and Microsoft's CLR Profiler.[19] Those profilers also support native code profiling, along with Apple Inc.'s Shark (OSX),[20] OProfile (Linux),[21] Intel VTune and Parallel Amplifier (part of Intel Parallel Studio), and Oracle Performance Analyzer,[22] among others.

Instrumentation [edit]

This technique effectively adds instructions to the target program to collect the required information. Note that instrumenting a program can cause performance changes, and may in some cases lead to inaccurate results and/or heisenbugs. The effect will depend on what information is being collected, on the level of timing details reported, and on whether basic block profiling is used in conjunction with instrumentation.[23] For example, adding code to count every procedure/routine call will probably have less effect than counting how many times each statement is obeyed. A few computers have special hardware to collect information; in this case the impact on the program is minimal.

Instrumentation is key to determining the level of control and amount of time resolution available to the profilers.

- **Manual**: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- **Automatic source level**: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- **Intermediate language**: instrumentation added to assembly or decompiled bytecodes giving support for multiple higher-level source languages and avoiding (non-symbolic) binary offset re-writing issues.
- **Compiler assisted**
- **Binary translation**: The tool adds instrumentation to a compiled executable.
- **Runtime instrumentation**: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool.
- **Runtime injection**: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions.

Interpreter instrumentation [edit]

- **Interpreter debug** options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator [edit]

- **Hypervisor**: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
- **Simulator** and **Hypervisor**: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator.

See also [edit]

- Algorithmic efficiency
- Benchmark
- Java performance
- List of performance analysis tools
- PAPI is a portable interface (in the form of a library) to hardware performance counters on modern microprocessors.
- Performance engineering
- Performance prediction
- Performance tuning
- Runtime verification
- Profile-guided optimization
- Static code analysis
- Software archaeology
- Worst-case execution time (WCET)

References [edit]

1. ^ "How to find the performance bottleneck in C# desktop application?". Stack Overflow. 2012.
2. ^ Krauss, Kirk J (2017). "Performance Profiling with a Focus". Develop for Performance.

3. ^ "What is code profiling? Learn the 3 Types of Code Profilers". *Stackify Developer Tips, Tricks and Resources*. Disqus. 2016.
4. ^ Lawrence, Eric (2016). "Getting Started with Profile Guided Optimization". *testslashplain*. WordPress.
5. ^ Krauss, Kirk (2018). "Matching Wildcards: An Improved Algorithm for Big Data". Develop for Performance.
6. ^ "List of .Net Profilers: 3 Different Types and Why You Need All of Them". *Stackify Developer Tips, Tricks and Resources*. Disqus. 2016.
7. ^ Unix Programmer's Manual, 4th Edition
8. ^ *a b* S.L. Graham, P.B. Kessler, and M.K. McKusick, *gprof: a Call Graph Execution Profiler*, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, *SIGPLAN Notices*, Vol. 17, No 6, pp. 120-126; doi:10.1145/800230.806987
9. ^ A. Srivastava and A. Eustace, *ATOM: A system for building customized program analysis tools*, Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI '94), pp. 196-205, 1994; ACM *SIGPLAN Notices* - Best of PLDI 1979-1999 Homepage archive, Vol. 39, No. 4, pp. 528-539; doi:10.1145/989393.989446
10. ^ 20 Years of PLDI (1979–1999): A Selection, Kathryn S. McKinley, Editor
11. ^ E. Coppa, C. Demetrescu, and I. Finocchi, *Input-Sensitive Profiling*, IEEE Trans. Software Eng. 40(12): 1185-1205 (2014); doi:10.1109/TSE.2014.2339825
12. ^ D. Zaparanuks and M. Hauswirth, *Algorithmic Profiling*, Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), ACM SIGPLAN Notices, Vol. 47, No. 6, pp. 67-76, 2012; doi:10.1145/2254064.2254074
13. ^ T. Kustner, J. Weidendorfer, and T. Weinzierl, *Argument Controlled Profiling*, Proceedings of Euro-Par 2009 – Parallel Processing Workshops, Lecture Notes in Computer Science, Vol. 6043, pp. 177-184, 2010; doi:10.1007/978-3-642-14122-5 22
14. ^ "Timing and Profiling - Basic Block Profilers". *OpenStax CNX Archive*.
15. ^ Ball, Thomas; Larus, James R. (1994). "Optimally profiling and tracing programs" (PDF). *ACM Transactions on Programming Languages and Systems*. ACM Digital Library. **16** (4): 1319–1360. doi:10.1145/183432.183527. Archived from the original (PDF) on 2018-05-18. Retrieved 2018-05-18.
16. ^ Statistical Inaccuracy of `gprof` Output Archived 2012-05-29 at the Wayback Machine
17. ^ "Popular C# Profilers". Gingtage. 2014.
18. ^ "Sampling Profiler - Overview". *AQTime 8 Reference*. SmartBear Software. 2018.
19. ^ Wenzal, Maira; et al. (2017). "Profiling Overview". *Microsoft .NET Framework Unmanaged API Reference*. Microsoft.
20. ^ "Performance Tools". *Apple Developer Tools*. Apple, Inc. 2013.
21. ^ Netto, Zanella; Arnold, Ryan S. (2012). "Evaluate performance for Linux on Power". *IBM DeveloperWorks*.
22. ^ Schmidl, Dirk; Terboven, Christian; an Mey, Dieter; Müller, Matthias S. (2013). *Suitability of Performance Tools for OpenMP Task-Parallel Programs*. Proc. 7th Int'l Workshop on Parallel Tools for High Performance Computing. pp. 25–37.
23. ^ Carleton, Gary; Kirkegaard, Knud; Sehr, David (1998). "Profile-Guided Optimizations". *Dr. Dobb's Journal*.

External links [edit]

- Article "Need for speed — Eliminating performance bottlenecks" on doing execution time analysis of Java applications using IBM Rational Application Developer.
- Profiling Runtime Generated and Interpreted Code using the VTune Performance Analyzer

Categories: Software optimization | Profilers