CPI: Continuous Profiling Infrastructure

Lance Berc Sanjay Ghemawat Monika Henzinger Shun-Tak Leung
Mitch Lichtenberg Dick Sites Mark Vandevoorde Carl Waldspurger
Bill Weihl

cpi@pa.dec.com
http://www.research.digital.com/SRC/cpi/

Digital Systems Research Center
Palo Alto, CA 94301 USA

We have developed a profiling system named *CPI* for Digital Alpha platforms that permits continuous profiling of entire systems, including the kernel, user programs, drivers, and shared libraries. A profile database is incrementally updated for every executable image that runs. A suite of profile analysis tools helps identify and interpret performance problems uncovered by profiling.

Our goal is to make the system efficient enough that it can be left running all the time, allowing it to be used to drive online profile-based optimizations for production systems. Our current prototype system is quite close to achieving this goal. In contrast to prior profiling systems, CPI has the following innovative features:

- **Efficiency:** The CPI system has extremely low CPU overhead – approximately 1–2%. Its memory and disk requirements are also modest. A typical profile consumes significantly less disk space than its corresponding executable image; a typical profile database consumes less than 10MB per week.
- **Transparency:** Earlier profiling tools such as *gprof*, *pixie*, and *atom* required programs to be recompiled or modified for profiling. CPI works on unmodified executables, enabling profiling on production systems.
- **Completeness:** CPI profiles entire workloads, not just single images, providing comprehensive coverage of overall system activity.
- **Accuracy:** CPI reveals where all time is being spent down to the level of individual instructions, including time spent waiting for memory accesses. Most profiling systems simply count basic block executions.

The design of the CPI system contains several interesting features. We use the Alpha performance counters to sample program counter values. On a 10-processor SMP running at 400 MHz, we get about 6100 samples per second per processor, for an overall total of 61000 samples per second. A device driver services the interrupts, and a user-mode daemon extracts raw samples from the driver, associates them with executable images, and updates disk-based profiles. The driver uses hash tables to aggregate samples, reducing the amount of information that must be communicated to the user-space daemon by a factor of 10 to 100.

The profiling system produces sample counts for every instruction in every executable that is run. In addition to the core profiling infrastructure, we have implemented several utilities to analyze the sample counts:

- *cpiprof* generates prof-style output, producing a breakdown of the time spent by image, or by procedure within each image.
- *cpicalc* uses the sample counts to compute the average number of cycles taken by each individual instruction, showing which instructions have stalled and for how long.
- Several other utilities produce basic-block flow graphs and annotate them with sample counts, CPI numbers, and source code.

We are currently working on other utilities to assign blame for wasted cycles, to determine which stalls result from memory waits, which from branch mispredicts, and so on. In addition, we are building utilities to map the profile information directly back to the source code, with the goal of identifying individual variable references as the source of memory-related stalls. Our ultimate goal is to use the detailed profile information generated by the CPI system to drive optimizations and transformations of programs, at both the source level and the object level.