May 4, 1995

Wisconsin Architectural Research Tool Set

Mark D. Hill, James R. Larus, Alvin R. Lebeck,
Madhusudhan Talluri, and David A. Wood

Computer Sciences Department
University of Wisconsin
1210 West Dayton St.
Madison, WI 53706
warts@cs.wisc.edu

Wisconsin Architectural Research Tool Set (WARTS) is a collection of tools for profiling and tracing programs and analyzing program traces. WARTS currently contains:

- QPT, a program profiler and tracing system.
- CPROF, a cache performance profiler.
- Tycho and dineroIII, cache simulators.
- EEL, a library for editing executable files.

WARTS is distributed with the full source and a small amount of documentation. The tools in WARTS are copyrighted and distributed under license. A copy of the license is available on `ftp.cs.wisc.edu` in `~ftp/pub/warts/license.ps` or it can be obtained by writing to the address above.WARTS is available without charge for university researchers and is available to other researchers for a modest research donation. Contact `warts@cs.wisc.edu` for more details or to be added to the mailing list to hear of future releases.

QPT: A Quick Program Profiling and Tracing System

QPT is an exact and efficient program profiler and tracing system written by James Larus. The `qpt` tool rewrites a program's executable file (`a.out`) by inserting code to record the execution frequency or sequence of every basic block (straight-line sequence of instructions) or control-flow edge. From this information, another program `qpt_stats` can calculate the execution cost of procedures in the program. Unlike the Unix tools `prof` and `gprof`, QPT records exact execution frequency, not a statistical sample. When tracing a program, QPT produces a trace regeneration program that reads the highly compressed trace file and regenerates a full program trace.

When profiling, QPT operates in two modes. In "slow" mode, it places a counter in each basic block in a program—in the same manner as the MIPS tool `pixie`. In "quick" mode, QPT places counters on an infrequently-executed subset of the edges in the program's control-flow graph. This placement can reduce the cost of profiling by 3–4 times. Since there is no such thing as a free lunch, quick profiling requires more program analysis and consequently slows QPT. The additional cost to instrument a program and report results, however, is small and is quickly gained back when profiling long-running programs.

QPT offers a number of advantages over existing systems:

- In quick mode, QPT can record edge execution frequencies as well as basic block execution frequencies.

- QPT also records a number of semantics events—such as loop entry and iteration, function entry and exit, and memory allocation and deallocation.
- On MIPS systems, QPT typically requires less overhead than `pixie` and can profile programs that use signals. QPT can also produces hierarchical profiles (like `gprof`).
- On SPARC-based systems, QPT offers exact instruction counts, instead of a PC-histogram that is subject to quantization errors. QPT also does not require recompilation of the program.

QPT currently runs on MIPS and SPARC-based systems. However, QPT is obsolete and has been succeeded by QPT2, which provides roughly the same functionality and is based on EEL (see below).

The profiling and control-tracing algorithms in QPT are described in:

[1] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, July 1994, pp. 1319–1360.

Abstract execution is described in:

[2] James R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs," *Software Practices & Experience*, 20, 12, December 1990, pp 1241-1258.

QPT is described in:

[3] James R. Larus, "Efficient Program Tracing," IEEE Computer, 26, 5, May 1993, pp 52-61.

CPROF: A Cache Performance Profiler

The CPROF system is a cache performance profiler written by Alvin R. Lebeck and David A. Wood that annotates source listings to identify the source lines and data structures that cause frequent cache misses. The CPROF system consists of two programs: `Cprof`, a uniprocessor cache simulator, and `Xcprof`, an X windows user interface. `Cprof` processes program traces generated by QPT (see above) and annotates source lines and data structures with the appropriate cache miss statistics. `Xcprof` provides a generalized X windows interface for easy viewing of annotated source files.

The performance of current RISC processors is very sensitive to cache miss ratios. Programmers, compiler writers, and language designers must explicitly consider cache behavior to fully exploit a program's performance potential. CPROF provides detailed information about a program's cache behavior through full cache simulation. By annotating lines of source code and data structures with the corresponding number of cache misses, CPROF helps the user focus on problematic data structures and algorithms. CPROF aids the programmer in identifying types of transformations that can improve program cache behavior by classifying cache misses as: compulsory, capacity, or conflict.

Note: This product contains software developed by the University of California, Berkeley and its contributors.

Our experience using CPROF to tune this subset of the SPEC benchmarks is detailed in:

[1] Alvin R. Lebeck and David A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," IEEE Computer, vol. 27, no. 10, Oct. 1994, pp. 15–26.

Tycho and DineroIII: Cache Simulators

Tycho and *dineroIII* are uniprocessor cache simulators written by Mark Hill. The simulators report the behavior of one or more alternative cache designs in response to an input trace provided by the user (e.g., with QPT). A trace is a list of the memory references that a program or workload makes while it is executing. Both simulators are written in C, use the same ASCII trace format, and have been distributed to dozens of companies and universities.

The first simulator, *tycho*, simultaneously evaluates many alternative uniprocessor caches, but severely restricts the design options that may be varied [1]. Specifically, with one pass through an address trace, *tycho* will produce a table of miss ratios for caches of many sizes and associativities, provided that all caches have the same block (line) size, do no prefetching, and use LRU replacement. *Tycho* is used, for example, with the SPEC benchmark suite to examine numerous caches [2]. Tycho is most useful for reducing the size of a large cache design space. A second version of *tycho*—*tychoII*—provides higher performance with the option of binary trace input and several other optimizations by Madhusudhan Talluri. *TychoII*, however, is more complex than *tycho* and has not been widely used.

The second simulator, *dineroIII*, evaluates only one uniprocessor cache at a time, but produces more performance metrics (e.g., traffic to and from memory) and allows more cache design options to be varied (e.g., write-back vs. write-through, LRU vs. random replacement, demand fetching vs. prefetching). *DineroIII* is distributed for instructional use with a popular graduate computer architecture textbook [3]. *DineroIII* is most useful for carefully studying a few alternative cache designs.

[1] Mark D. Hill and Alan Jay Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. on Computers,* C-38, 12, December 1989, p.1612-1630.

[2] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, Alan Jay Smith, "Cache Performance of the SPEC Benchmark Suite," to appear, IEEE Micro, August 1993, 3, 2.

[3] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, California, 1990

EEL: Executable Editing Library

EEL (Executable Editing Library) is a library for building tools to analyze and modify an executable (compiled) program. The systems and languages communities have built many tools for error detection, fault isolation, architecture translation, performance measurement, simulation, and optimization using this approach of modifying executables. Currently, however, tools of this sort are difficult and time-consuming to write and are usually closely tied to a particular machine and operating system. EEL supports a machine- and system-independent editing model that enables tool builders to modify an executable without being aware of the details of the underlying architecture or operating system or being concerned with the consequences of deleting instructions or adding foreign code.

EEL is a new C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools. EEL differs from other systems in two major ways: it can edit fully-linked executables, not just object files, and it emphasizes portability across a wide range of systems.

EEL comes with a new implement of QPT (QPT2). It is described in:

[1] James R. Larus and Eric S. Schnarr, "EEL: Machine-Independent Executable Editing," To appear: *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.