deionizer: a tool for capturing and embedding I/O calls

Michael Bedford Taylor
CSAIL, Massachusetts Institute of Technology
June 8, 2004

1. ABSTRACT

In this paper, we introduce the concept of a *deionizer*. A deionizer is a special type of partial evaluator whose purpose is to create a new version of a program that can run without accessing a partial set of I/O resources. Although a deionizer can be used for *application embedding*, this short paper addresses the use of dionization for *improving benchmark accuracy*. The paper briefly discusses the key ideas and then explains the implementation and use of the MIT deionizer. This deionizer was used to produce the results for a recent conference paper that compares the Raw processor to a Pentium III.

2. INTRODUCTION

Partial evaluation or *program specialization* is a well-studied technique for program optimization [1]. A partial evaluator takes an input program, P_I, and part of its input data, D_0. The evaluator produces an output program P_O, which, when run with the remaining input D_1, produces the same result as P_I with input D_0 and D_1.

Partial evaluation is typically used for program optimization. If part of the program's input is known to be constant, all of the operations that are dependent only on those inputs can be resolved ahead-of-time. Alternatively, operations whose inputs are known in part can be transformed into easier operations; for instance, a divide-by-2 can be transformed into a shift operation.

This paper describes a specialized form of partial evaluation, *deionization*. A deionizer performs partial evaluation; however it intentionally abstains from optimizing the output program against the input. The purpose of a *deionizer* is not to optimize the performance of the program, but rather to allow it to run without accessing the original input devices.

3. USES

Application Embedding One use of deionization is for *application embedding*. For instance, an application may make use of a number of data files (for instance, bitmaps in a game). Although the developer may want to develop these files separately in source form (say, a .gif file), deionization can be used to wrap these files together into a single binary that is sent to the user. Or, in the case of an embedded system that does not have certain I/O devices, deionization allows the program to be written as if those I/O devices existed. A final binary, containing the source and data, and suitable for burning to a read-only memory, can then be rendered by the deionizer.

Benchmark Accuracy A deionizer can also be used to improve the accuracy of benchmarks in computer systems. This usage, the focus of this short paper, allows the I/O and operating system calls to be removed from benchmark applications. The deionizer described in this paper was used to improve the accuracy of collected results in a comparison between a Pentium 3 system and a Raw processor in [2].

This has two benefits:

1. Deionization removes the dependence of program running time on portions of computing systems that are not salient to the experiment being undertaken. For instance, when comparing two processor implementations, it is typical to measure the running time of the same application and data set on the two machines. However, differing implementations of operating systems, I/O architecture, motherboards, network file systems, and hard drives may cloud our ability to draw conclusions about the impact of the processor on the running time of the program.

2. Deionization eliminates many sources of variability for data collection, even for successive runs of the same application on the same machine. The running time of a program that does I/O can be dependent on the history of the operating system's file system cache, the position of disk platters as they spin, or even the temperature of the hard drive[1]. If the collecting machine uses network file storage, the timing variance can be even worse. Thus, deionization can decrease the noise margins of experiments, allowing more accurate measurements to be made.

As the reader may suspect, the choice of the term deionizer is a double play on words. First, a deionizer as de-io-nizer suggests an application that removes I/O call. Secondly, classical scientific experiments often call for water to be deionized in order to ensure reasonable accuracy of the results.

4. IMPLEMENTATION

A deionizer can be implemented in a number of ways, depending particularly upon the goal. The MIT deionizer implementation is intended for use in improving benchmark accuracy rather than for application embedding, and thus has a number of simplifying limitations. A version that supports application embedding might be closer to the implementation of a virtual file system; some I/O calls would go out to the real I/O system; while others would be intercepted and serviced through processing of internal (often memory-resident) data structures.

Operation The MIT deionizer targets POSIX-style binaries that are generated using the GNU binary utilities. As such, it should be usable with any system that supports the GNU binary utilities. This deionizer operates in four passes.

[1] Some hard drives perform thermal recalibration in response to temperature-dependent expansion or contraction of components, causing a pause in the availability of data.

1. First, an automatic tool generates a modified version of the libc library. This version of libc performs the same I/O operations as the original libc; however it also intercepts and records the results of these I/O calls in an internal data structure. The generation of this file would typically only need to be done once per system being benchmarked.

2. Second, the input program P_I is linked using the special version of libc to create an intermediate binary, P_M. Thus, P_M is an instrumented version of P_I that records the results of I/O operations.

3. Third, P_M is run, creating as a bi-product a C file that contains substitute functions for all of the I/O function calls; this is essentially a partially evaluated libc library for just that program/input set pair.

4. Fourth, the object files of P_I are relinked with the compiled C file to form the final program P_O.

Compiled into P_O is a strict list of intercepted function calls, their input parameters, and return parameters (including the value of *errno* and other globals.) When one of the substitute I/O function calls is invoked, it verifies that the call is next on the global list, and returns the appropriate return values.

Figure 1 shows the relationship between the three programs, P_I, P_M, and P_O. Note that dependencies from many levels of the original I/O hierarchy have been eliminated from use.

Required Invariants This system assumes that the user can modify the target program P_I to have two function calls. The two calls bookend the region of the program that should be deionized. The first call demarks the starting point at which all I/O calls will be recorded. The second call indicates the ending point of I/O call recording. The user must ensure that successive runs will lead to the same state when reaching the first call: for instance, 1) flushing stdout and stderr before entering the first call to ensure that the buffering on these file descriptors is always in a consistent state, 2) making sure that the pattern of allocation and deallocation of file descriptors is always the same order to ensure file descriptor numbers do not change and 3) ensuring that runs consistently run with output either piped or unpiped to ensure consistent values for *isatty*. The user must also ensure that the code after the second call do not depend on the execution of the file system operations in the region to be deionized.

Typically for our purposes, the only code that is placed before the first call is a function to record the start time via the processor cycle counter; the only code after the second call is a function to print out the end time and the difference between the two times.

5. ANALYSIS

More quantitative analysis of this technique is desirable, but will not be provided in this document. Measurements of the variance in program runs, and of the trade-offs in substituting static data for I/O accesses should be analyzed.

Current processor benchmarks like SpecInt or SpecFP avoid the use of applications that spend more than a small percentage of their time in I/O; the use of a deionizer could enlarge the selection of programs that could be used as processing benchmarks. Additionally, the input sizes of benchmarks are often set so that the I/O time is reduced to a small percentage of run time; this tool could enable more accurate data-gathering with smaller traces.

6. PORTING TO A NEW SYSTEM

This section discusses the process of porting the MIT deionizer to a new platform. GNU binutils 2.10 or later is required; objcopy must support –redefine-sym. The MIT deionizer has two system-dependent configuration files, which are used to generate the special version of libc described in part 1 of the subsection **Operation**:

1. *intercept.in*, used as input to *intercept.pl*, to rename I/O functions inside the original version of libc.a. Typically all of the POSIX layer functions will be renamed (e.g., open, read, write); however, the standard library may also have special internal names for these functions (such as __close) which will need to be renamed as well, to guarantee that they are intercepted. Finally, some libraries use weak linking to create aliases for function names; these can be specified as well. Finally, newer systems like Linux may have alternative entry points like "open64" or "read64." The objdump utility can also be used on the original libc.a library to identify routines that invoke POSIX system calls.

2. *deionize.in*, used as input to *deionize.pl*, to generate a .c file containing intercept code, as shown in Figure 1. *deionize.in* contains type signatures of the functions in *intercept.in*. It also has the annotations *remember_input_buf*, which indicates that the system should record a input memory array, and *remember_result_buf*, which indicates the system should record an output memory array. The output .c file can be compiled and archived with the output of intercept.in to form a single .a file that contains all necessary code for the modified libc.

   The UNIX utility *strace* can be used to examine P_O's execution to verify that it is no longer calling I/O routines through the OS.

These two files are linked with P_I (in substitution of the original libc) to generate P_M.

7. ACKNOWLEDGMENTS

The author would like to thank Walter Lee, David Wentzlaff, and Karen Zee for discussions in conjunction with the development of the MIT deionizer. This research was funded by DARPA, NSF, the MIT Oxygen Alliance, and an Intel Graduate Fellowship.

REFERENCES

[1] N. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys 28*, 3 (September 1996).

[2] M. B. Taylor, et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. *Proceedings of the International Symposium on Computer Architecture, 2004*.

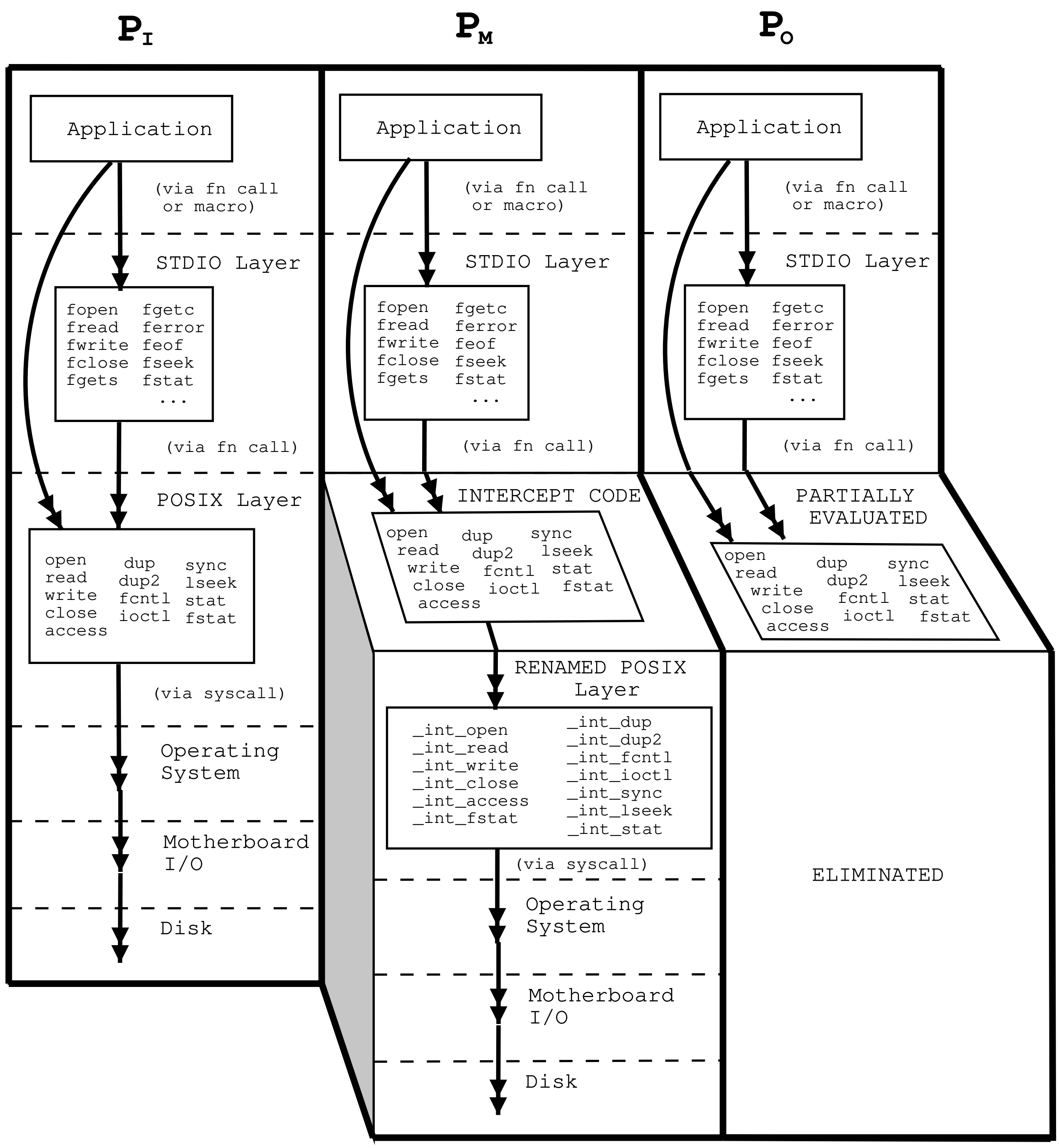

Figure 1: The I/O abstraction layers of the initial input program P_I, the intermediate program P_M and the final output program P_O.