

Ease: An Environment for Architecture Study and Experimentation†

JACK W. DAVIDSON AND DAVID B. WHALLEY

Department of Computer Science

University of Virginia

Charlottesville, VA 22903, U. S. A

Gathering detailed measurements of the execution behavior of an instruction set architecture is difficult. There are two major problems that must be solved. First, for meaningful measurements to be obtained, programs that represent typical work load and instruction mixes must be used. This means that high-level language compilers for the target architecture are required. This problem is further compounded as most architectures require an optimizing compiler to exploit their capabilities. Building such a compiler can be a formidable task.

The second problem is that gathering detailed dynamic measurements of an architecture using typical user programs reading typical data sets can consume significant computation resources. For example, a popular way to gather execution measurements is to simulate the architecture. This technique is often used when the architecture in question does not yet exist, or is not yet stable and available for production use. Depending on the level of the simulation, programs can run 100 to 500 times slower than directly-executed code [HUGU87]. Tracing is another alternative one can use if the architecture being measured exists, is accessible, and tracing is possible on that machine. Tracing can be even slower than simulation [HUGU87]. Because of the large performance penalties with these methods, the tendency is to use small programs with small data sets. The relevance of measures collected this way is always subject to question.

This paper describes an environment called *ease* (Environment for Architecture Study and Experimentation) that solves both these problems. It consists of an easily retargetable optimizing compiler that produces production-quality code. The compiler also supports the generation of instrumented code that gathers very fine-grained execution statistics with little overhead. Typically, instrumented code runs 10 to 15 percent slower than code that is not instrumented. Similarly, because information about instructions are collected as a side effect of the compiler generating code, compilation time is only increased by 15 to 20 percent. The combination of an easily retargetable compiler and an efficient method of observing the run-time behavior of real programs provides an environment that is useful

†This work was supported in part by the National Science Foundation under Grant CCR-8611653.

in a number of contexts.

ease logically consists of two parts; the set of tools for building optimizing compilers quickly and the tools that produce and analyze the measurements of the execution behavior of the instruction set architecture. The compiler technology is known as *vpo* [BENI88, DAVI84, DAVI86]. An efficient way to collect measurements for subsequent analysis is to modify the back end of the compiler to store the characteristics of the instructions to be executed and to produce code that will count the number of times that each instruction is executed. These modifications have been implemented in *vpo* and are shown in Figure 1.

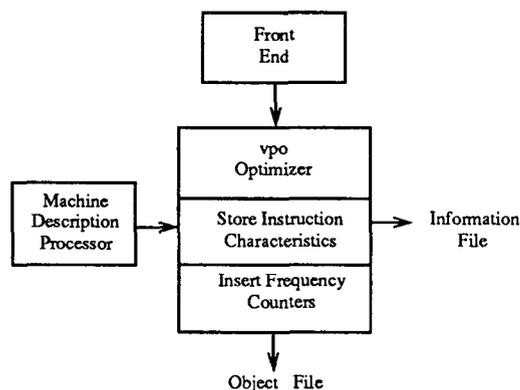


Figure 1. Schematic of *ease*.

The first modification necessary to collect measurements is to have *vpo* save the characteristics of the instructions that will be executed. During code selection, information about the characteristics of the instructions are gathered and used for semantic checks. The semantic checks are extended to store these characteristics with the instruction by invoking a machine-independent routine. After all optimizations have been completed, the information about each instruction is then written to a file for subsequent processing.

The second modification is to have *vpo* generate code to count the number of times each instruction is executed. Again this is accomplished after all optimizations have been performed. Within each function there are groups of instructions, basic blocks, that are always executed the same number of times. There are also groups or classes of basic blocks that are

executed the same number of times and these are denoted as execution classes. Using the dataflow information collected by the optimizer, the execution classes are determined and code to count the number of times that each execution class is executed is inserted at the beginning of the first basic block in the execution class.

At the end of the execution of the program, the number of times that each execution class is executed is written to a file. The execution counts and the characteristics of the instructions can then both be used to produce dynamic measurements. The characteristics of the instructions can also be used to produce static measurements.

ease has been ported to ten different machines to compare current architectures. Measurements from the execution of a test set of nineteen C programs were obtained for each of the architectures. The detail and accuracy of the reports produced by *ease* allowed insights to be drawn when analyzing the measurements. The measurements collected include:

- instruction path length
- instruction path size
- instruction type distribution
- addressing mode distribution
- memory reference size distribution
- memory reference address distribution
- register usage
- condition code usage
- conditional branches taken
- average number of instruction between branches
- data type distribution

The measurements are sufficiently detailed to determine the number of times each combination of addressing mode and data type is used for each field of each type of instruction. Results comparing the ten architectures analyzed appears in WHAL89.

In addition to using *ease* to evaluate and analyze existing instruction set architectures, it can be used to help design new machines [DAVI89b]. In this case, *vpo* emits code for an existing host machine that emulates the instruction set of the machine being designed. *vpo*'s organization permits this to be done quickly and easily as follows. The last step in the compilation process is the conversion of an machine-independent representation of an instruction to assembly language for the target machine and its emission to a file that will be processed by the system's assembler. In order to evaluate an architecture that does not exist, rather than emit assembly code for the target machine, assembly code for an existing architecture is emitted. Information about the effects of the instruction are emitted as if the target architecture existed.

ease has also been used to analyze different code generation strategies. For instance, by recompiling the source files from the C run-time library, different calling sequence conventions have been investigated [DAVI89a]. By extracting measurements of the behavior of the code, the effect of any change can be easily observed.

This environment for the collection of architectural measurements has been designed to require little effort when retargeting for a new architecture. Since the code selector and

other optimizations are constructed automatically, a *vpo*-based compiler is easy to retarget. Because the optimizer stores information about instructions using a machine-independent representation, it is easy to produce assembly code for both existing and proposed architectures and to store instruction information for the collection of measurements. Most of the code to perform the extraction of measurements is also machine-independent. A *vpo*-based C compiler for ten different machines was modified to collect measurements as specified above. For each machine, it typically took three to four hours to make the necessary machine-dependent modifications to the compiler.

The *ease* environment has been shown to be an efficient tool for architectural evaluation and design. Since accurate and detailed reports can be produced for a variety of measurements, the impact of each modification to the compiler or architecture can easily be determined. This allows one to use an iterative design method for evaluation of performance in a quantitative manner.

ACKNOWLEDGEMENTS

Manuel Benitez helped implement the machine-independent portion of *vpo*.

REFERENCES

- [BENI88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation, Atlanta, GA, June 1988*, 329-338.
- [DAVI84] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems* 6,4 (October 1984), 7-32.
- [DAVI86] J. W. Davidson, A Retargetable Instruction Reorganizer, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction, Palo Alto, CA, June 1986*, 234-241.
- [DAVI89a] J. W. Davidson and D. B. Whalley, Methods for Saving and Restoring Register Values across Function Calls, Tech. Rep. 89-11, University of Virginia, November 1989.
- [DAVI89b] J. W. Davidson and D. B. Whalley, Reducing the Cost of Branches by Using Registers, TR89-14, University of Virginia, November 1989.
- [HUGU87] M. Huguet, T. Lang and Y. Tamir, A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements, *Proceedings of the SIGPLAN Notices '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, MN, June 1987*, 14-25.
- [WHAL89] D. B. Whalley, A Study of High-Level Language Architectures, Ph.D. Dissertation Proposal, University of Virginia, Charlottesville, VA, June 1989.