

Characterization of Essential Dynamic Instructions*

Steven S. Lumetta and Sanjay J. Patel
Department of Electrical and Computer Engineering
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{*steve,sjp*}@*crhc.uiuc.edu*

Categories and Subject Descriptors

B.m [Hardware]: Miscellaneous; D.m [Software]: Miscellaneous

General Terms

Measurement, Performance

Keywords

dynamic instruction stream, reverse analysis

1. INTRODUCTION

We provide a methodology for characterizing the dataflow behavior of dynamic instruction streams through the use of a tool that simplifies the process of analysis by examining instruction-level traces in reverse order of execution. Many properties of instruction streams are more amenable to processing in this direction, and the combined two-pass (creation and post-processing) analysis renders most measurements straightforward. Using this approach, we characterize the streams of the SPEC2000 integer benchmarks compiled for the Alpha ISA on the OSF operating system. We separate dynamic instructions into essential and non-essential categories. Non-essential instructions contribute neither to program output nor to control flow, and can thus be removed by an optimizer to improve performance, reduce power, *etc.* For essential instructions, we touch on redundancy through an examination of silent stores (those that write values identical to those already in memory) and their associated dataflow. We separate the remaining essential instructions into those that affect program output and those that affect only control flow. Finally, we employ the approach to examine the amount of live data in memory over the lifetime of a program. Further characterizations and additional details about the reverse trace tool appear in a technical report.

Processing consists of two phases, one in which the trace is generated, and one in which the trace is analyzed backwards, *i.e.*, from the last dynamic instruction to the first. To produce the traces, we used an extension of the Alpha instruction-level simulator in the SimpleScalar 3.0 tool set to execute versions of the SPEC2000 integer benchmarks compiled using the Compaq Alpha C compiler,

*This work was funded in part by NSF CAREER grants NSF-CCR-00-92740 and NSF-ACI-99-84492, and gracious support from AMD, Intel, and Sun. The content of the information does not necessarily reflect the position or the policy of these organizations.

Compaq C V5.9, with optimization level 4 (except for *eon*, which used *g++*). The SPEC input sets for most of the benchmarks were modified to enable the benchmarks to simulate all parts of the program in reasonable time. All benchmarks execute to completion except *vpr*, which we capped at one billion instructions; most execute for a few hundred million instructions.

The traces contain one entry (approx. 20 bits compressed) per dynamic instruction indicating the registers and memory touched by the instruction, the program counter, and the instruction class. For each store instruction, we include a bit to denote whether the store was silent. Program traces are processed in reverse order to reduce the complexity of classification. When processing a particular instruction, all consumers of the instruction's result are already known, as is the point at which any unused results are overwritten by other values. Thus, our measurements are not limited to a fixed instruction window, but can span the duration of a program. At the end of a program (the start of processing), all registers and memory locations are considered dead. We assume that applications produce output only through system calls; in particular, applications are assumed to neither share memory with other processes nor to implicitly pass information via timing behavior. As the simulator only emulates traps/system calls, we recorded production and consumption of registers and memory at the user-system interface according to the nature of the particular system call.

2. INITIAL CHARACTERIZATION

We begin our investigation by separating dynamic instructions into essential and non-essential groups. Essential instructions contribute either to program output, to control flow, or to both. Obvious examples of non-essential instructions include *nop* instructions and prefetches, which have no effect on program-visible state (registers and memory). More subtle examples include calculations for control paths not executed, production of memory values never used, and stack save/restore of unused caller registers.

A breakdown of dynamic instructions for the SPEC2000 integer suite appears in Figure 1. The proportion of non-essential instructions (top two categories) is striking; these instructions make up about a quarter of all dynamic instructions in optimized binaries. The *nop*'s account for roughly one in ten instructions, reflecting a common rule of thumb for processors that benefit from branch target alignment. Prefetches make up only an insignificant part of the dynamic instruction stream (at most 0.3% across all benchmarks), and are included with the *nop* category. The remaining 15% consists of dead instructions. These proportions are roughly the same for the unoptimized versions of the benchmarks.

The remaining categories compose the essential instructions. Within this group, it is possible to remove some redundancy by executing silent stores and the instructions that produce their addresses and values in parallel with the remaining instructions, pos-

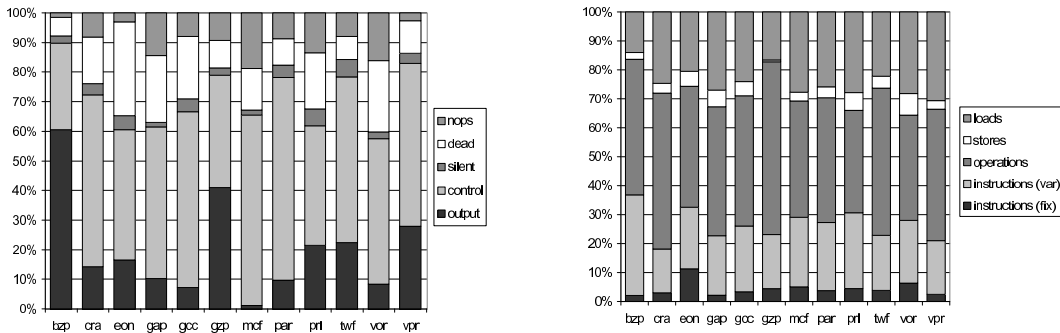


Figure 1: Breakdown of dynamic instructions and control-related dataflow in the SPEC2000 integer suite.

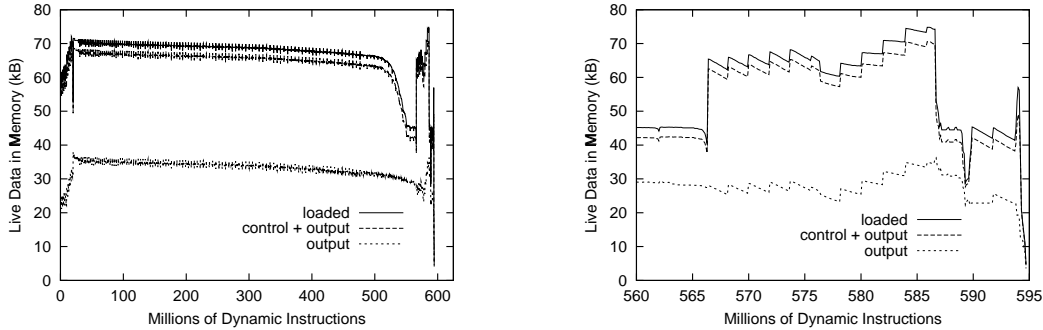


Figure 2: Live memory in `twolf`. The left graph shows the full execution, the right a phase near the end.

sibly eliminating their impact on performance entirely. As shown, the benefit of parallel silent store execution in terms of dynamic instructions accounts for another few percent of the total.

After separating out silent stores and their dataflow, the remaining essential instructions are broken into two categories: output and control. Output includes all instructions that contribute directly to a program’s output, *i.e.*, the dataflow that produces memory or register values consumed by system calls. Control contains instructions that contribute only to control flow (and thus indirectly to program output), including both the actual control instructions and all non-control instructions that contribute to conditions, indirect jump targets, *etc.* Instructions that contribute to both program output and control flow are included in the output category. The small fraction of data operations relative to control may reflect algorithmic trade-offs in the benchmarks; in particular, programmers often reduce the number of data operations at the expense of more complex control. The tiny fraction of output instructions with `mcf`, a simplex-based, schedule optimization code, supports this notion.

3. MORE DETAILED ANALYSES

Approximately half of all dynamic instructions pertain only to control flow. These instructions offer an opportunity for performance optimization, as control flow can be often be speculated correctly. For example, branches that are highly biased can be statically or dynamically speculated, and the verification (*i.e.*, the instructions to generate conditions) can happen later, at low priority. The right graph in Figure 1 divides dynamic instructions associated with control flow into loads, stores, operations (*e.g.*, addition), and control instructions (*e.g.*, branch instructions). The control instructions are further subdivided into those with variable targets (indirect jumps and conditional branches) and fixed targets (direct jumps and unconditional branches).

The distribution is fairly regular across all benchmarks. Roughly speaking, for each control instruction, there is a single load and two operations. The small fraction of stores indicates the likelihood that

control data written to memory is read many times, as is the case with entries in a linked list or hash table.

The reverse-stream analysis tool also enables a unique view of the data usage patterns of a running application. At any point during execution, we can determine the amount of live data in memory and build a profile of value bandwidth behavior.

The left graph in Figure 2 characterizes live data in memory over the execution of the `twolf` benchmark. This code performs placement and routing via simulated annealing. The values plotted are the maximum amount of live data within disjoint windows of 100,000 instructions. Three lines are plotted for each: data pertaining to output, data pertaining to either control or output, and data that are loaded to registers before the end of the program. Only the first two categories are live.

The central feature in the graph is the period over which simulated annealing occurs for cell placement. The ramps at the left and right represent dual-pass scans over the netlist, the only fairly long input file. Interestingly, roughly 3-4 kB of dead data are kept around for the duration of the program.

The right graph in the figure zooms in on the end of the execution, highlighting the file ramp. Reads of file data in 4 kB chunks are clearly visible in the control+output line. The data is consumed, producing a more compact version, some of which affects the output, and another 4 kB block is fetched until all 22 kB of the file have been read (twice). The jump in control data around the file activity represents the creation and destruction of a hash table, which serves to shorten the time required to locate matching elements, but has no effect on the program output. This behavior illustrates how programmers trade reductions in data operations for more complex control (and more control data). It also highlights the potential for using this type of profile information to locate blocks of critical control data and potentially improve cache behavior.

The figure is representative of the benchmarks, although program phases are not as easily identified in the interpreters (`gap`, `perl`).