Stijn Eyerman†  Lieven Eeckhout†  James E. Smith‡
†ELIS Department, Ghent University, Belgium
‡ECE Department, University of Wisconsin – Madison

In modern processors, both the hardware implementation and optimizing compilers are very complex, and they often interact in unpredictable ways. A high performance microarchitecture typically issues instructions out-of-order and must deal with a number of disruptive miss events such as branch mispredictions and cache misses. An optimizing compiler implements a large number of individual optimizations which not only interact with the microarchitecture, but also interact with each other. These interactions can be constructive, destructive, or neutral. Furthermore, whether there is performance gain or loss often depends on the particular program being optimized and executed.

In practice, the only way that the performance gain (or loss) for a given compiler optimization can be determined is by running optimized programs on the hardware and timing them. This method, while useful, does not provide insight regarding the underlying causes for performance gain/loss. By using the recently proposed method of interval analysis [1, 2], one can decompose total execution time into intuitively meaningful cycle components. These components include a base cycle count, which is a measure of the time required to execute the program in the absence of all disruptive miss events, along with additional cycle counts for each type of miss event. Performance gain (or loss) resulting from a compiler optimization can then be attributed to either the base cycle count or to specific miss event(s).

By analyzing the various cycle count components for a wide range of compiler optimizations one can gain insight into the underlying mechanisms by which compiler optimizations affect out-of-order processor performance. The work reported here provides and supports a number of key insights. Some of these insights provide quantitative support for conventional wisdom, while others provide a fresh view of how compiler optimizations interact with superscalar processor performance. To be more specific:

Interval Analysis. We demonstrate the use of interval analysis for studying the impact of compiler optimizations on superscalar processor performance; this is done by breaking up the total execution time into cycle components and by analyzing the effect of compiler optimizations on the various cycle components. Compiler builders can use this methodology to better understand the impact of compiler optimizations.

Evaluating Compiler Optimizations. Our analysis provides a number of interesting insights with respect to how compiler optimizations affect out-of-order processor performance. For one, the critical path leading to mispredicted branches is the only place during program execution where optimizations reducing the length of the chain of dependent operations affect overall performance on a balanced out-of-order processor — inter-operation dependencies not residing on the critical path leading to a mispredicted branch are typically hidden by out-of-order execution. Second, reducing the dynamic instruction count (an important optimization objective dating back to sequential processors) still is an important compiler optimization criterion for today’s out-of-order processors. Third, some compiler optimizations (unintentionally) bring long-latency loads closer to each other in the dynamic instruction stream, thereby exposing more memory-level parallelism (MLP) and improving performance.

Out-of-Order versus In-Order. We show that compiler optimizations have a different performance impact on inorder versus out-of-order processors. In fact, the biggest fraction of the total performance gain on inorder processors is achieved by reducing the dynamic instruction count and critical path length. For out-of-order processors on the other hand, only about half the performance gain comes from reducing the dynamic instruction count and critical path length; the other half comes from optimizations related to the I-cache, L2 D-cache and branch predictor behavior.

References