A Comparison of Allocators

David Koes
dkoes@cs.cmu.edu

March 11, 2006

In this document we compare the graph-coloring based allocator and the local/global non-iterative default allocator of version 3.4.3 of the GNU compiler `gcc`*.*

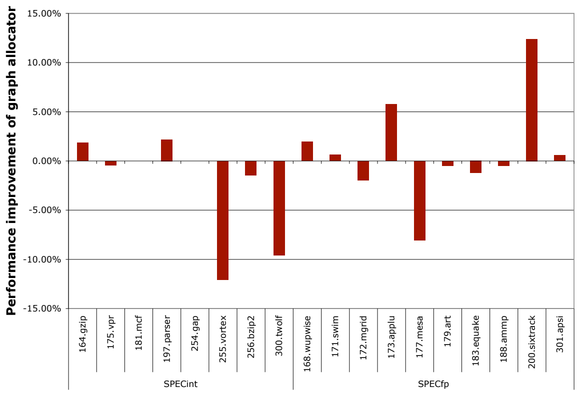

Figure 1: Performance improvement of the graph allocator compared to the default allocator. The code was compiled with the flags `-O3 -funroll-loops` and executed on a RedHat Linux 9.1 workstation with a 1.8Ghz Pentium 4 and 1.5GB of RAM.

The performance of selected SPEC benchmarks compiled with the graph allocator compared to code compiled with the default allocator when targeting x86 is shown in Figure 1. The results are mixed with most benchmarks demonstrating no significant change (as the SPEC benchmark suite uses wall clock timing and the benchmarks were not run multiple times in generating this data, a certain amount of measurement data should be expected – sorry).

The results for code size when targeting x86, 68k, and PPC, which have 8, 16, and 32 registers respectively, are shown in Figure 2. In this case,

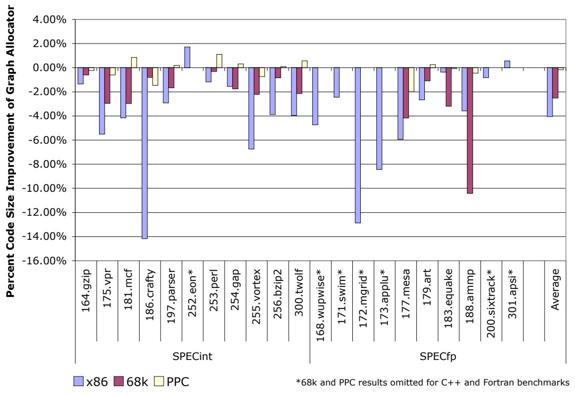

Figure 2: Code size improvement of the graph allocator compared to the default allocator. The code was compiled with the flags `-Os`. Only the size of the resulting `.text` section is measured.

the graph-coloring based allocator does almost uniformly worse than the default allocator suggesting that it generates substantially more spill code (although probably not within critical loops given the performance results). As expected, the difference in code size quality decreases as more registers are available for allocation.

We compare the ability of both allocators to avoid spilling by compiling over 9,000 functions assembled from several benchmark suites (include SPEC, MediaBench, and MiBench). The results are shown in Figure 3. The graph allocator is more successful than the default allocator at avoiding spills when the interference graph is colorable. However, when it is necessary to generate spill code, the graph coloring allocator spills more variables on average than the default allocator. In cases where both the

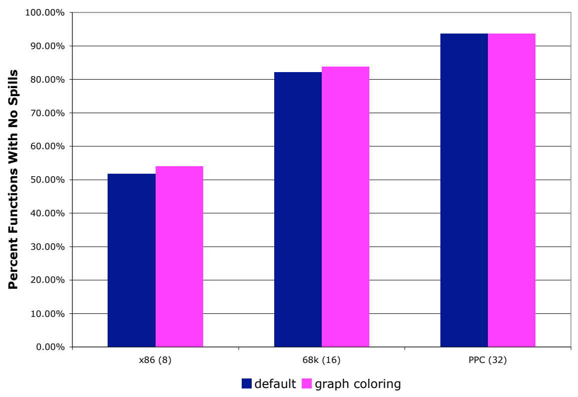

Figure 3: The percent of functions successfully allocated without generating any spill code for both the graph allocator and default allocator. The graph allocator is more successful at coloring the interference graph, especially when resources are scarce.

	Register Allocation	Compilation
x86	4.17x	1.31x
68k	2.5x	1.17x
PPC	4.8x	1.33x

Table 1: Total cumulative slowdown of just the register allocation pass and total compilation as reported by the `-ftime-report` flag when compiling with `-O3 -funroll-loops`.

graph allocator and default allocator are forced to spill, the graph allocator spills 53.6%, 14.2%, and 11.5% more variables than the default allocator for the x86, 68k, and PPC architectures, respectively. The distribution of the spill ratio is shown in Figure 4. As the number of available registers decreases and the irregularity of the architecture increases, the distribution shifts to the detriment of the graph allocator. These figures should be treated with some skepticism since the different allocators have differing notions of exactly what a spilled variable is. However, the fact that the most common case (20% to 30% of spilling functions) is for both allocators to spill exactly the same number of variables indicates these numbers may not be completely irrelevant.

The graph coloring allocator is significantly

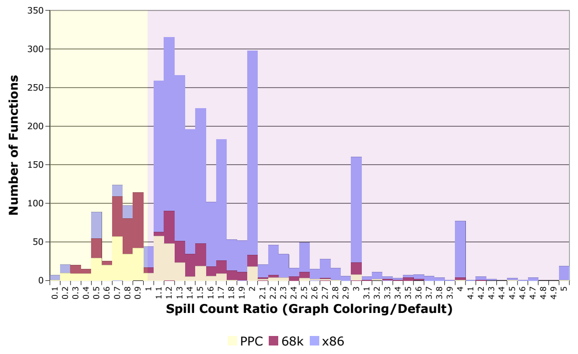

Figure 4: A histogram of the spill count ratio of the graph allocator and the default allocator. Values to the right of 1.0 indicate functions where the graph allocator spilled more variables than the default allocator. The most common case, where both allocators spilled the same number of variables, is omitted to improve the readability of the graph.

slower than the default allocator as shown in Figure 1. The graph allocator is also much buggier than the default allocator. It failed to compile four functions in our benchmark suite when targeting x86 and one function when targeting 68k (the default allocator had no problems). When a function fails to compile, all remaining functions in that file are not compiled. In performing our comparison we only present data for the functions that were successfully compiled by both architectures resulting in some inter-architectural discrepancies.

The graph allocator was removed from `gcc` in version 4.0.