Memory Optimization for C implementations of Whiley

Min-Hsien Weng, Dr. Bernhard Pfahringer and Dr. Mark Utting
University of Waikato
Hamilton, New Zealand
mw169@students.waikato.ac.nz utting@usc.edu.au

Call-by-value semantics[1] makes Whiley program verification easy at compile-time. But when translating Whiley program into C code, the implementation has several performance issues: a) excessive copying overheads as arrays are immutable, and are copied before each update; b) severe memory leaks as arrays are allocated on the heap and not de-allocated.

Static analysis techniques are applied to improve the efficiency of generated *C11* code. The live variable analysis is first used to determine dead variables, which will not be used/read afterwards, and then eliminate unnecessary copying of data structures. Then de-allocation analysis checks code properties and chooses suitable macro to change runtime de-allocation flag, to ensure at each program the allocated memory space can only be freed by exactly one variable.

Problem Size	N	N + D	C	C + D
100,000	4,800,256	0	1,600,248	0
1,000,000	48,000,264	0	16,000,256	0
10,000,000	480,000,272	0	160,000,264	0

Table 1: Memory Leaks of 'Reverse' Test Case using Valgrind Tool

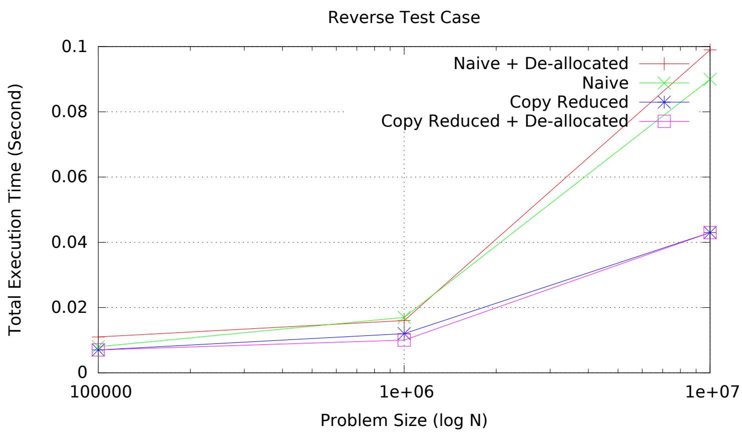

Figure 1: Execution Time of 'Reverse' Test Case

The 'Reverse' Whiley program is translated into naive (N), naive + memory de-allocation (N+D), copy reduced (C) and copy Reduced + memory de-allocation (C+D). Copy optimization increases speed up to 2.1x, and de-allocation macro reduces the leaks effectively. For other benchmark programs, the speed-up varies from 1.0x to 2.9x.

References

[1] David J Pearce and Lindsay Groves. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming*, 113:191–220, 2015.