Compilers - Phase 3 Analysis

Naved Rizvi
Usman Ehtesham Gul

Analysis

Benchmark 1

Algorithm	Instructions	Reads	Writes	Branches	Other
Naive	517	194	138	38	147
Intra-block	684	189	230	38	227
Briggs	339	17	134	38	150

Benchmark 2

Algorithm	Instructions	Reads	Writes	Branches	Other
Naive	1168	566	324	38	240
Intra-block	1149	282	416	38	413
Briggs	618	17	320	38	243

Benchmark 3

Algorithm	Instructions	Reads	Writes	Branches	Other
Naive	519	194	138	38	149
Intra-block	688	191	230	38	229
Briggs	344	19	135	38	152

Benchmark 4

Algorithm	Instructions	Reads	Writes	Branches	Other
Naive	1170	566	324	38	242
Intra-block	1153	284	416	38	415
Briggs	623	19	321	38	245

Comparing intra-block with naive, we see a higher number of instructions in most benchmarks for intra-block. This is possibly because of emitting many load/store instructions at each basic block entry and exit point. The number of reads for intra-block compared to naive is slightly lower than naive except for benchmar_2, where the number of reads is almost half as the number of reads for naive. This could be because, in benmark_2, we are over-writing the same variable multiple times and that variable is loaded at the beginning of each basic block. We do not see a big difference in the performance between intra-block and naive because we do not take the liveness analysis of the variables into account. Hence when entering or exiting a block, we load all variables referred to in a block without taking into account if a variable is actually live on entering a block or saving all variables referred to in a block without taking into account if a variable is actually live on exiting a block. In our implementation, we also do a few redundant move instructions which could lead to the reason for the number of writes for intra-block being higher than naive.

Comparing Briggs with the other register allocation algorithms, we see a significant improvement in the instructions count and in the number of reads and a slight improvement in the number of writes. This is because in Briggs, we take the liveness of variables into account and this allows us to re-use registers where possible since for a given pair variables, their live ranges do not overlap and hence the same register can be used to refer to both variables. We also only load on the first reference of a given variable and during the liveness of that variable, we do not have to do any additional load from memory (unlike in naive, where we would have to do this on reference to the same variable) and this leads to a significant decrease in the number of instructions and number of reads. We do see a reduction in the number of writes and that is because we would only write a register back to memory if we are about to enter a function call and then we can load the register back from memory at the end of the function call. We do this as we try to follow a function calling convention to ensure the save and temp registers do not get over-written due to a function call within a given function. We definitely see a benefit of an algorithm like Briggs to register allocation and the liveness analysis of an IR program along with the interference graph of the live ranges of variables allows us to utilize the limited number of physical registers more optimally than naive and intra-block (where intra-block currently uses a weak heuristic: frequency of the number of references of a variable in a basic block). Taking the liveness analysis into account gives a boost in performance as we avoid unnecessary loads and stores; also we take the whole function into account as opposed to taking basic blocks into

account. If a variable is not deemed to be used from the liveness analysis, then we do not have to take that variable into consideration for register allocation.