Kaivalya Mishra CS415

Project 1 Report

Overview of Implemented Algorithms

For this project, there were four different algorithms for local register allocation that were to be implemented. The following lists each of the methods alongside their symbols which will be referenced later on:

- `b` – Bottom-Up Allocation
- `s` – Simple Top-Down Allocation
- `t` – Live Range Comparision Top-Down Allocation
- `o` – Consecutive Usage Comparision Top-Down Allocation

For the custom allocation method, `o`, its principles were similar to that of the `s` method, in which method `o` assigned physical registers to the highest priority virtual registers for the duration of their live ranges and utilized the number of occurences for ranking the priorities. However, the tiebreaker for `o` looks at which register had greater consecutive usage throughout the block instead of which register had the shorter live range.
To demonstrate this idea, here is an example segment of ILOC code:

```
add r5, r6 => r7
mult r5, r7 => r8
sub r8, r6 => r9
loadI 4 => r10
lshift r6, r10 => r11
add r11, r5 => r12
```

In the example, the two registers with the greatest number of occurences are `r5` and `r6`. Method `s` would give higher priority to `r6` due to its shorter live range. Method `o` however,

would instead give higher priority to `r5` due to it being called upon twice in a row while `r6` had calls which were seperated by other lines of instructions.

For every algorithm, if it was found that the number of virtual registers was less than or equal to the number of physical registers given, no modifications would be made to the original instructions and the code would be passed along as is.

All top-down methods shared the same function for assigning physical registers and inserting spill code once their unique priority rankings were determined. Method `b` was the only one with a completely seperate function due to its fundamentally different approach to allocation.

Report Results

Number of Instructions/Operations Executed							
Method	Registers	Report 1	Report 2	Report 3	Report 4	Report 5	Report 6
b	5	134	50	65	36	86	47
	10	79	50	55	36	67	47
	15	63	50	49	36	53	47
s	5	192	138	120	101	133	139
	10	172	123	100	86	118	124
	15	155	108	83	72	103	109
t	5	134	84	87	60	113	81
	10	103	50	65	36	86	47
	15	64	50	49	36	65	47
o	5	143	82	88	60	113	81
	10	107	50	68	36	80	47
	15	64	50	49	36	61	47

Number of Cycles for Execution							
Method	Registers	Report 1	Report 2	Report 3	Report 4	Report 5	Report 6
b	5	273	62	100	46	143	52
	10	114	62	70	46	100	52
	15	65	62	52	46	70	52
s	5	497	383	318	252	364	357
	10	457	327	263	217	297	322
	15	399	279	206	184	230	287
t	5	336	220	203	150	292	201
	10	207	62	115	46	201	52
	15	66	62	52	46	116	52
o	5	363	221	209	150	288	201
	10	235	62	129	46	179	52
	15	66	62	52	46	108	52

Execution Times of Allocator Invocation

Average Times over 5 Executions (milliseconds)							
Method	Registers	Report 1	Report 2	Report 3	Report 4	Report 5	Report 6
b	5	3.6	2.6	3.4	3	3	3.2
	10	2.8	2	2.6	2.2	2.8	3
	15	2.8	2	2.6	2	2.8	3
s	5	3.6	3	2.8	3.2	3	3.6
	10	3.4	2.6	2.2	2.6	3.2	3.4
	15	3.2	2.4	2.6	2.2	3	3.4
t	5	3.4	2.8	3.4	3.8	3.4	3.8
	10	3.2	2.4	2.8	2.8	3.4	3.6
	15	2.8	2.2	3.4	3	3	3
o	5	3	3	4.2	3.6	3.6	3.8
	10	2.8	2.4	3.6	3	3.4	3.4
	15	2.6	2	3.4	3	3	3.4

(The times were recorded via the `time` command, using the `real` output values. The virtual machine used was ilab1.)

Discussion of Results

Firstly, viewing the number of instructions added and the cycle times of the different methods, method `b` had the fewest additions to the base code and the lowest cycle times by a wide margin. Method s had the greatest number of modifications to the base code and thr largest cycles times, always having to make additions to the code even when given 15 registers. Method `t` and `o` had a roughly even performance overall, being better than `s` but behind `b`, sometimes not needing to insert spill operations in certain cases.

The following sections will mainly discuss methods `b`, `t`, and `o` due to method `s` not having substantial differences in regards to performance across the reports.

Report 1

In Report 1, method `b` saw substantial improvement for instructions used and cycle time when moving from 5 to 10 registers and achieved optimal performance with 15 registers. Method `t` appeared to have better performance than `s`, using fewer instructions and having even lower cycle times. Particularily, `o` used a greater number of cycles than `t` for 10 registers, despite not having that many more instructions.

There is a specific instance of unecessary spill code being added for the top-down algorithms in this report, due to there being dead code in the form of virtual register `r35` never being utilized despite having a value loaded into it.

Report 2,4, and 6

Regarding Reports 2, 4, and 6, method `b` had optimal performance for each category of registers given. Methods `t` and `o` achieved optimal performance when given 10 and 15 registers and had very close performances when given 5 registers.

Looking closer at Report 2, it should be noted that when using 5 registers, `o` had slightly greater cycle time than `t` despite using fewer instructions.

Report 3

Similar to Report 1, in Report 3, method `b` achieved optimal performance when given 15 registers but did not see as substantial of an improvement when moving from 5 to 10 registers. Method `t` performed slighlty better than `o` with fewer instructions used and lesser cycle time when using 5 and 10 registers. Both methods attained optimal performance when using 15 registers.

Report 5

Unlike in the other reports, method `b` never achieved optimal performance even when given 15 registers. When using 5 registers, `o` and `t` utilized and equal number of instructions,

but `o` had lesser cycle time. For 10 and 15 registers, `o` had fewer instructions and lesser cycle time.

Average Execution Times

Turning attention to the time elapsed for the executables to process, the averages ranged from 2-3.8 milliseconds. Interestingly, method `b` appeared to have the shortest average execution times, followed by method `s`, with `t` and `o` having the longest. This may just be due to when the commands were executed and what other processes the machine may have been handling, as there were some interruptions and connection issues during testing. When comparing moving to different categories of registers given, average execution times evidently decrease when given a greater number of registers.

Conclusions

Based on the analysis of the results previously mentioned, it can be stated that the allocator executes faster when given more registers to use, most likely due to needing to insert less spill code into the program. Comparing the different methods, it can be stated that allocations made by `b` have the best performances, given the that it focuses more on replacement than allocation, greatly limiting the spill code that would need to be inserted. Method `s` consequently has the worst performance, since it assigns only one virtual register to a physical register for the entire block, ignoring live ranges completely and thus needing a much greater amount of spill code. Since `t` and `o` pay attention to the live ranges, they see a performance improvement over `o`, but not quite to the level of `b` because of looking over the input code with a broader perspective rather than a case by case basis. Specifically comparing `t` and `o`, it may be the case that `o` performs better than `t` when there are more densely overlapping register live ranges. Referencing Report 5, the performance difference could be due to the fact that the virtual registers had closer live range lengths, thus leading to the tiebreaker portion of `t`'s heuristic not being as effective as `o`'s which instead looks at consecutive usage. The opposite case can be made when the live ranges have greater discrepancies, such as in Reports 1 and 3.