10/22/2018

VLIW Schedule Report

Project 1, Part 1

Andrew Tam, Victor Keyes, Chris Church
TEAM 29

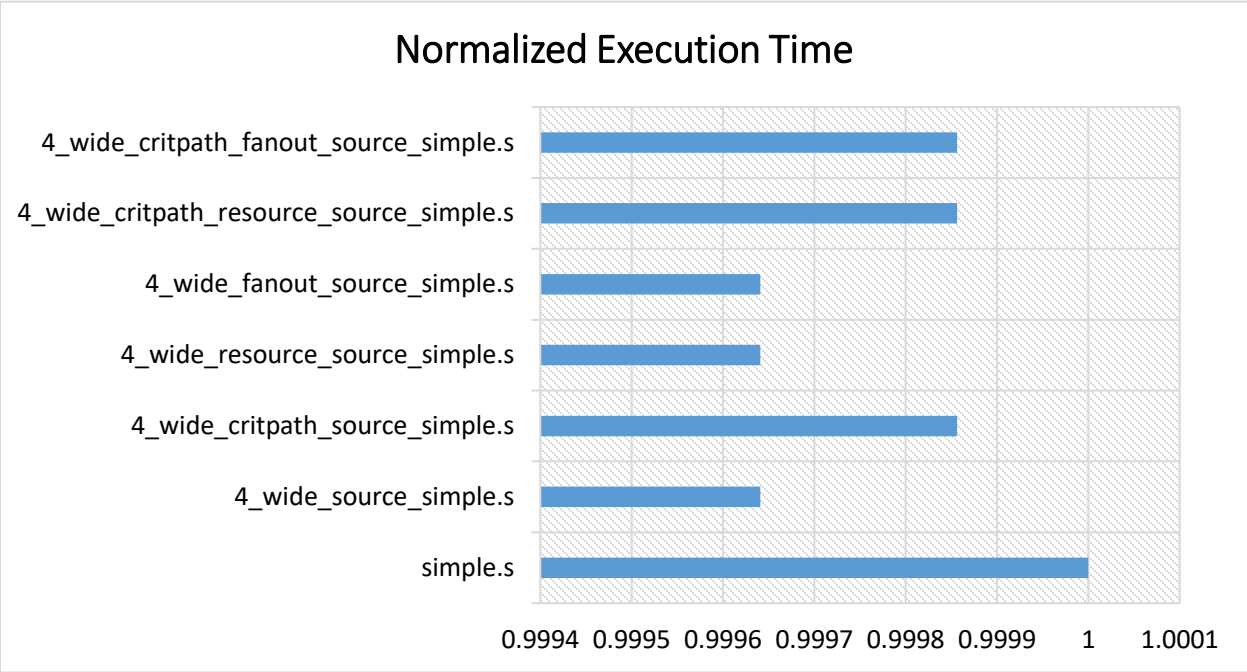
Critical Path for Simple.s

Simple.s Order	Simple.s Instruction	Delay
1	ldw $r0.2 = 0x24[$r0.1]	3
5	mpylu $r0.8 = $r0.2, $r0.5	2
8	add $r0.8 = $r0.8, $r0.9	1
10	ldw $r0.9 = 0x1c[$r0.1]	3
13	mpylu $r0.11 = $r0.9, $r0.6	2
14	mpyhs $r0.6 = $r0.9, $r0.6	2
16	add $r0.11 = $r0.11, $r0.6	1
17	add $r0.8 = $r0.8, $r0.11	1
19	add $r0.5 = $r0.5, $r0.8	1
20	add $r0.2 = $r0.2, $r0.5	1
21	ldw $r0.5 = 0x30[$r0.1]	3
23	add $r0.6 = $r0.5, $r0.3	1
24	add $r0.10 = $r0.10, $r0.6	1
25	add $r0.10 = $r0.10, $r0.2	1
28	add $r0.4 = $r0.10, $r0.9	1
Critical Path Length		**24**

Schedule Length for Different Heuristics

	Schedule Length (Instructions)
Simple.s	31
4_wide_source_simple.s	26
4_wide_critpath_source_simple.s	29
4_wide_resource_source_simple.s	26
4_wide_fanout_source_simple.s	26
4_wide_critpath_resource_simple.s	29
4_wide_critpath_fanout_simple.s	29

Performance Results



	cycles	ms	Normalized Time
simple.s	13918	0.027836	1
4_wide_source_simple.s	13913	0.027826	0.999640753
4_wide_critpath_source_simple.s	13916	0.027832	0.999856301
4_wide_resource_source_simple.s	13913	0.027826	0.999640753
4_wide_fanout_source_simple.s	13913	0.027826	0.999640753
4_wide_critpath_resource_source_simple.s	13916	0.027832	0.999856301
4_wide_critpath_fanout_source_simple.s	13916	0.027832	0.999856301

Discussion

As the raw data show above, the 4-wide program with source tiebreaking saved 5 cycles compared to the original 1-wide program. Tiebreaking using resource-source and fan-out-source heuristics also saved 5 cycles, which indicates the original source program was written efficiently. Surprisingly, the critical path-source heuristic only saved 2 cycles compared to the 1-wide program. This was attributable to the fact that the function to create critical paths does not in fact generate every possible delay chain, but only the one it expects to be the longest one ending with each node. The original version of the critical path heuristic function added a new path whenever the dependencies branched out, but because the dependency graph included implied as well as direct dependencies (e.g. instruction #11 has only a direct dependence on #9, but the dependence vector for instruction #11 also lists some of instruction #9's dependencies which are implied for #11 by its dependence on #9), thousands of delay chains were created for this short section of code in the complete critical path finding solution.

In order to keep memory usage reasonable, it was gambled that whenever multiple dependencies existed for a node, the longest critical path would include the dependency that came later in the program and this was the only delay path saved. Because of this incompleteness, the critical path ordering was less effective than source, resource-source, or (direct dependency) fan-out-source. It is worth noting that the critical path calculation, even in its abbreviated form as described here, was significantly more complex to implement than any of the other tiebreaking heuristics but performed worse. Also worth noting was that the critical path-resource-source and critical path-fan-out-source heuristics performed the same as the critical path-source heuristic, which indicated that secondary tiebreaking was not critical given the uniformity of performance among the secondary heuristics of source/resource/fan-out.