Spenser Haddad

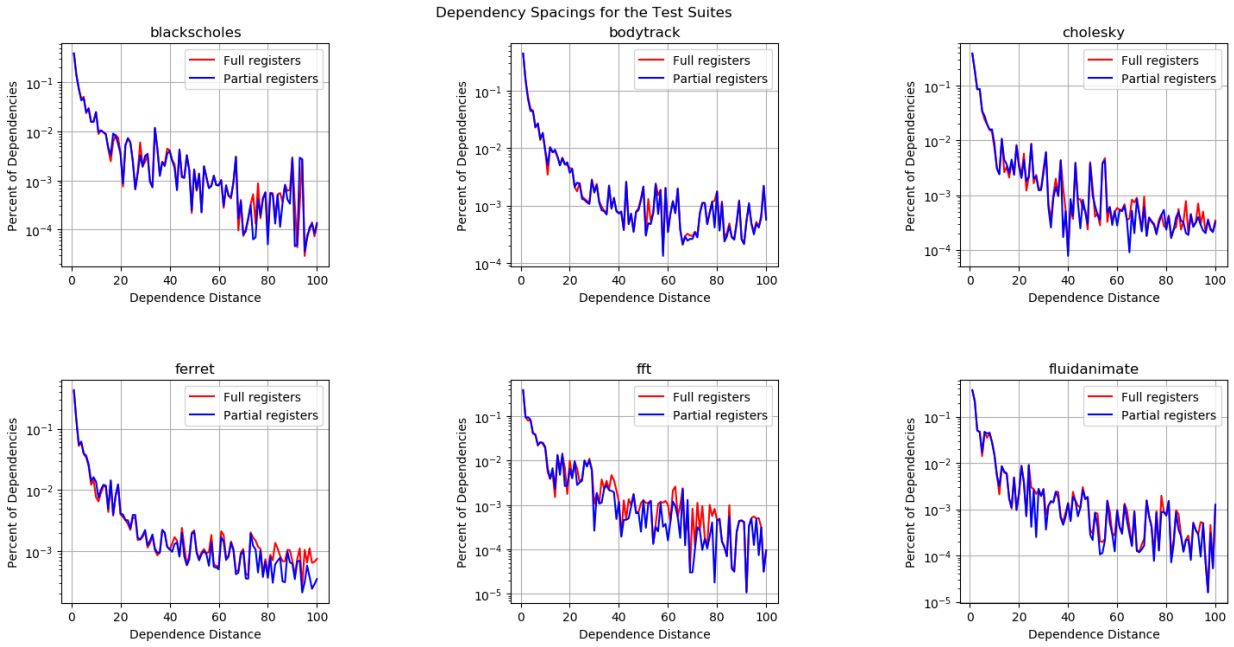
ECE 513 Computer Architecture Lab 1 Answers

1. Given that the Fetch stage is the most cycle-intensive of the pipeline, Architecture A would be the more performant choice. First of all, Architecture B uses stalling, which most likely means that replaces the instructions with bubble cycles that will lower the effective CPI of the architecture. Second, assuming that it doesn't have any mechanism to speed up the fetching of stalled instructions, Architecture B requires two fetches for every delayed instruction, one the first time the instruction is encountered and one when it fetches it after the stall. This means each of these instructions is using the most expensive stage twice per execution. If the cost is significantly higher than the cost of the other two stages, then it effectively doubles the CPI of each delayed instruction. This is a considerable performance hit that Architecture A avoids by keeping the pipeline dependencies within the cheaper Execute and Writeback stages. In addition, Architecture A does not use any bubble cycles, which will also improve the CPI compared to Architecture B.

2. Below is a compressed table of all registers used in a "grep" command, and the number of times that register had a certain dependency. For example, register RDI had a dependency of one 12,688,728 times. As can be seen, registers RBP, RSP, and YMM0 were used significantly more for longer-term dependencies. At the same time, these registers were used several orders of magnitude less for short-term dependencies. The architecture may be intentionally forwarding data it won't be using until several instructions later into these specific registers, while reserving the rest for the more common scenario where it only needs a register for the next instruction. If one could find values that are going to be used several times, and therefore should remain cached for extended periods of time, then this behavior could be used to put the values in registers that are designed to be read either repeatedly or far in the future (relatively speaking).

	1	2	3	4	5	6	7	8	9	10	...	91	92	93	94	95	96	97	98	99	100
RDI	1268728	82771	55204	98157	206291	210492	71448	39873	4613	3749	...	0	1	14	14	3	0	1	1	1	0
RSI	1094841	41263	14071	4280	3268	4056	774	3404	1773	818	...	0	9	2	1	1	1	9	1	1	0
RBP	6046	5115	392	3551	824	391	646	233	689	1099	...	98	91	135	224	226	142	265	232	62	71
RSP	309739	2070	6166	4557	5463	1468	1779	1650	2119	1120	...	110	38	30	191	76	267	194	260	236	76
RBX	5713	6637	945	3792	679	1959	768	1119	1277	2706	...	26	28	34	24	14	40	8	13	43	21
RDX	87360	21417	6597	15257	4419	5797	2140	4353	3299	789	...	20	0	3	0	0	200	1	0	2	21
RCX	1280193	155055	15344	9635	57701	4526	1011	587	699	38055	...	1	0	0	1	845	1	0	2	2	0
RAX	533688	338528	16160	158629	18231	21496	7600	116358	2078	1673	...	12	18	12	15	15	12	12	6	7	15
R8	8742	6090	337	1373	3011	887	362	47	55	21	...	36	0	11	1	173	12	1	32	8	17
R9	7631	8221	1544	1003	141	60	166	127	319	28	...	0	0	0	0	0	0	1	0	1	0
R10	2714	3764	279	118	79	52	27	28	18	28	...	80	0	46	0	5	77	204	43	80	0
R11	266	139	486	74	344	16	10	12	101	28	...	5	1	171	4	0	4	0	4	1	0
R12	408244	298863	407706	225	215	110271	246	545	228	135	...	43	62	17	142	159	106	22	56	190	66
R13	1098	1029	970	303	255	424	352	319	425	138	...	6	15	3	90	4	11	6	8	4	44
R14	977	679	747	791	300	138	386	836	265	209	...	17	30	17	5	6	64	17	168	118	5
R15	1107	671	417	439	120	575	142	498	361	52	...	27	22	30	84	36	46	33	7	11	29
RFLAGS	586	1054825	17120	2907	3858	197	479	88	44	37	...	17	17	17	17	17	17	17	17	17	17
RIP	1491275	8289	2585	1094	1018	356	825	569	134	370	...	0	0	0	0	0	0	0	0	0	0
YMM0	122687	95	4184	480	61268	304	52444	111	8	32	...	0	55	319	2	582	825	867	812	236	22
YMM1	15747	160527	1832	2791	94724	70	294	0	3	64	...	0	0	0	0	0	0	0	0	0	1
YMM2	9723	1929	500	97047	182	11	33	0	3	0	...	0	0	0	0	0	0	0	0	0	1
YMM3	7504	628	263	94695	49	0	33	0	3	0	...	0	0	0	0	0	0	0	0	0	1
YMM4	6927	365	94669	0	0	0	0	9	3	70	...	0	0	0	0	0	0	1	0	0	2
YMM5	138291	94669	0	0	0	0	0	1	3	69	...	0	1	0	0	0	0	0	0	0	1
YMM6	1	94669	0	0	0	0	0	0	3	0	...	0	0	0	0	0	0	0	0	0	2
YMM7	0	0	0	0	129	1	0	0	3	0	...	0	0	0	0	0	0	0	0	0	1
YMM8	327	252	450	31	7	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
YMM9	141	0	0	136	0	0	0	24	0	110	...	0	0	0	0	0	0	0	0	0	0
YMM10	272	7	129	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
YMM11	0	0	7	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

3. The biggest tradeoff to only using full register reads and writes happens when multiple partial registers in the same full register are written in successive instructions. When this happens, it will set the instruction count for measuring dependency spacing for all registers within the full register to that current instruction. For all other registers, the resulting dependency spacing when finally read will be lower than it actually is. As a result, our overall measurements will show the lower dependency spacings as being more frequent than they actually are. Interestingly, using partial register names instead of full register names does not seem to have any impact on data accuracy. The plots below compare the test suite being run treating partial registers as full registers and treating them as the actual partial register. As seen below, for each of the six tests in the test suites, the results appear similar to one another.



4. Architecture A has more general purpose registers. Most of architecture B's weight is in dependency spacing 1, which implies that it has to write over registers more frequently than Architecture A, which means that it runs out of available registers faster, and that would be the case if it had less general purpose registers to begin with. By contrast, Architecture A's dependency graph is relatively more evenly distributed, which means its able to hold data in registers for longer. This means that it is not running out of registers to load data into as frequently as Architecture B. Granted, Architecture A still has mostly single-digit dependency distances, so it still is overwriting registers every few instructions, but it is doing so at a much lower rate than Architecture B.