Register Pressure Scheduling in LCM

A Performance Study of SPECjvm2008

The addition of a register pressure scheduling algorithm to Local Code Motion (LCM) helps handle cases where register pressure is significant. The algorithm schedules instructions which lower register pressure for integer and/or float register classes, tracking register pressure as LCM schedules instructions. Because this is done before register allocation, the effects of spilling to one or both classes of register can be reduced or even eliminated. When LCM runs on each basic block, we calculate the entry register pressure as a starting point and schedule accordingly based on whether we initially have too much register pressure to start with in one or both register classes. We alternate on the fly in and out of scheduling for register pressure based on a rolling count of register pressure for each of the register classes which are utilized as we schedule instructions. When scheduling for register pressure, we weight the best candidate as the instruction which will alleviate the maximum amount of register pressure. This makes the choice of register pressure threshold a careful one. If we choose too low, we throw away latency value in favor of register pressure, if we chose the threshold too high, we still spill even though we are trying to prevent it. If register pressure initially is below either threshold, we start scheduling a given basic block as a topological sort, weighing candidates on latency value. This is the default algorithm when register pressure scheduling is not being utilized.

Register pressure is seen on many common applications today and can be exemplified by running performance benchmarks like SPECjvm2008. Typically loops which have a modest amount of code and are unrolled could have register pressure, or even large loops that are never unrolled. Straight line code which is not encapsulated within loops can also have significant register pressure.

This report addresses the effects of register pressure scheduling by utilizing timing statistics in the C2 compiler in Java JDK version 9 (using –XX:+CITime). We measure both with and without the new scheduling algorithm enabled so that we can do a side by side comparison. We did not disable tiered compilation, so there is some randomness in the exact method compilation list for a given metric and the amount of time collected in C2 accordingly. We wanted natural hotspot detection and elevation to C2, so leaving tiered compilation *on* was desirable. We measure the results on a reportable Base run for SPECjvm2008. We chose to utilize x86 code generation on the 32-bit JDK and compiler suite for our examples, as there is a fairly rich set of data to mine there. We note that C2 time exceeds 10% on two metrics and that in general it is well below that as a portion of application runtime. We also see that scheduling never exceeds 10% of C2, meaning the effects of running the new scheduling algorithm never exceed more than 1% of runtime for the metrics in our application suite and are generally 0.5% or less. We also note that 8 of the metrics have uplift of more than 1% performance and as much as 6% with the new algorithm and that it does not degrade any metric in this suite. The average uplift of those 8 metrics is 3.4%. We expect similar behavior on *CPU* centric applications which run on both x86 and x64 where register pressure is an issue and subsequently spilling occurs in the generated code. We ran the metrics multiple times each to establish common behavior, throwing out spikes and keeping the median scores.

Below we show a table which exemplifies all the points we make here:

metric	register pressure scheduling=on scheduling	register allocation	total c2 time	score	register pressure scheduling=off scheduling	register allocation	total c2 time	score	c2 time delta	score delta
compress	0.245	1.375	3.401	191.8	0.114	1.381	3.282	191.1	3.63%	0.34%
crypto.aes	0.392	2.82	6.988	82.7	0.182	2.65	8.487	78.1	-17.66%	5.84%
crypto.rsa	0.455	2.9	6.17	245.8	0.215	2.943	5.965	232.2	3.44%	5.87%
crypto.signverify	0.465	3.099	6.33	355.5	0.23	2.923	6.014	343.7	5.25%	3.44%
derby	1.48	9.778	19.246	332.6	0.681	9.287	17.864	326.2	7.74%	1.94%
mpegaudio	0.501	2.453	6.14	153.1	0.287	2.714	6.284	152.5	-2.29%	0.39%
scimark.fft.large	0.146	0.864	1.859	81.2	0.073	0.908	1.885	81.3	-1.38%	-0.15%
scimark.lu.large	0.114	0.649	3.26	16.9	0.059	0.689	5.33	16.9	-38.84%	0.18%
scimark.sor.large	0.115	0.665	1.45	58.2	0.057	0.709	1.45	58.3	0.00%	-0.10%
scimark.sparse.large	0.116	0.692	1.499	41.3	0.054	0.649	1.369	40.0	9.50%	3.35%
scimark.fft.small	0.27	1.74	3.681	471.0	0.139	1.738	3.577	472.7	2.91%	-0.37%
scimark.lu.small	0.327	2.052	4.721	619.2	0.156	1.888	4.388	604.0	7.59%	2.50%
scimark.sor.small	0.117	1.02	2.231	262.4	0.093	1.167	2.375	262.1	-6.06%	0.11%
scimark.sparse.small	0.161	0.897	2.427	201.9	0.083	1.011	2.467	197.3	-1.62%	2.32%
scimark.monte_carlo	0.193	1.121	2.49	282.3	0.093	1.02	2.42	280.9	2.89%	0.49%
serial	0.713	5.284	10.079	157.5	0.322	4.885	9.223	156.5	9.28%	0.60%
sunflow	0.535	3.93	8.647	86.9	0.247	3.686	8.062	85.8	7.26%	1.66%
xml.transform	3.72	21.046	46.356	377.7	1.75	21.068	44.839	375.3	3.38%	0.52%
xml.validation	1.489	8.387	19.231	659.0	0.681	8.209	18.058	654.0	6.50%	0.76%
									0.08%	3.4%

Now we can discuss the data. One will notice that there are two corner cases where the change in C2 time is significantly higher without register pressure scheduling, these two cases are prime candidates of spill code generation. If either metric had such code in a hot path, performance would have significantly been better with register scheduling *on*. The first case, the code is warm enough to notice nearly 6% of performance. We ran each metric separately so that we could see local maxima and minima issues for cases with and without register pressure scheduling. In all the cases above, scheduling time goes up with our algorithm enabled, however, we also get much of that back during register allocation on fairly regular basis. The overall effect of the overhead of our algorithm upon C2 compile time is negligible because of the averaging effect of spill code mitigation for the metrics we ran. We also expect x64 to have an average of 5% overhead on C2 because of the existence of the algorithm as there are fewer cases where register pressure is mitigated. Please also note that scheduling time is always dwarfed by time in register allocation though so we expect the overhead to always be manageable.

Our machine configuration is as follows:

Skylake Desktop (release candidate), 2.2 GHz, 8.0GB ram, Windows 8.1/x64