CS 6290 Project 2: Tomasulo Algorithm

Sam Britt

April 6, 2012

Experimental Approach

To find the optimal design for the runs, a mostly brute-force approach was taken. However, some assumptions and insights reduced the solution space to something more manageable.

First, since increasing the fetch rate does not increase the hardware cost for our simulation, it was left at 8 and not altered. It was assumed that reducing the fetch rate would not increase the IPC, as a reduced fetch rate would only starve the scheduler for instructions to choose from.

Second, the total number of result buses can be at most the sum of the functional units. If there are f total functional units, then only f instructions can complete on any given cycle, and so any more than f result buses would be a waste.

Finally, through ad-hoc experimenting with the parameters of the simulation, it seems like the traces have threshold parameters, below which the IPC drops off rapidly. For example, perhaps the instructions of a trace use functional unit type 0 much more often than the others. In this case, IPC is not changed much when altering the number of functional units of types 1 and 2, but will change drastically when reducing the number of type 0 functional units. Or, often, a trace will exhibit little change in IPC when reducing the number of result buses up to a threshold, but once that point is passed the result bus becomes the bottleneck and IPC drops rapidly.

These observations drove the style of the optimization algorithm, which follows. First, a cutoff IPC value of 90 % of the maximum IPC was employed—any combination of hardware that resulted in an IPC less than this cutoff was not considered. This is an attempt to quantify a configuration that is "good enough." Other approaches are also valid; e.g., approaches that attempt to minimize the relative change between two configuration. However, the 90 % rule seemed to give reasonable results and was simple to implement. In the following optimization algorithm, N is the fetch rate, R is the number of result buses, and k_i is the number of functional units of type i.

1 Set their parameters to their maximums:
$N \leftarrow 8$, $R \leftarrow 6$, and $k_i \leftarrow 2$ for $i = 0, 1, 2$.
2 With these parameters, determine IPC_{max} and $\text{IPC}_{\text{cutoff}} = 0.9\text{IPC}_{\text{max}}$.
3 Leaving $R = 6$, exhaustively search the k_i space to find the minimum $\sum_i k_i$ that still results in an IPC less than the cutoff.
4 With this set of k_i, vary R from 1 to $\sum_i k_i$ to find the minimum R that still results in an IPC less than the cutoff.
5 This set of N, R and k_i is the optimal hardware configuration.

Since the algorithm optimizes the set k_i first, the largest part of the solution space that this algorithm does not cover is perhaps larger k_i and smaller R. However, some ad-hoc searches of this part of the solution space showed no better results than what this algorithm could produce.

Results

Table shows the final results of the optimization algorithm for each trace. Clearly, the most traces favor more functional units of type 0 over those of type 1 and 2. To investigate this further, a script was written to calculate the percentage of each type of instruction for each trace. Indeed, each of the "real" traces (`barnes`, `gcc`, `ocean`, and `perl`) had mostly type 0 instructions, anywhere from 65 to 90 % (results not shown, but the script is available in the source directory). The `test3` trace, which benefited from an extra functional unit of type 2, had 100 % type 2 instructions.

Table 1: Optimization results

Trace	IPC_{max}	IPC_{opt}	R	k_0	k_1	k_2	N
barnes.txt	2.74	2.74	3	2	1	1	8
gcc.txt	2.83	2.81	3	2	1	1	8
ocean.txt	2.23	2.23	3	2	1	1	8
perl.txt	3.03	2.97	3	2	1	1	8
test1.txt	0.33	0.33	1	1	1	1	8
test2.txt	0.67	0.67	2	2	1	1	8
test3.txt	0.50	0.50	2	1	1	2	8
test4.txt	0.80	0.75	2	1	1	1	8