CSE 501
Programming Assignment 3

Haichen Shen

Building

My project is written in Java. You can use "ant" to build the whole project. An ant build file is included in the tar. The default target will build the project and generate the jar file (compiler.jar).

How to use

Before you run the program, you need to first modify the "config" file. You need to input the directory of dart sdk and start. That will be used in the profiling process since I'll use the start to test run the program and collect the data.

A script run.sh is provided. The options supported are listed below:

./run.sh <input file> [-opt=<optimize>] [-profile=<profile>] [-backend=<backend>]

Optimization supported options:
ssa SSA optimization
cp Constant propagation optimization (depends on SSA)
vn Value numbering optimization (depends on SSA)

Profile supported options:
pos Basic block positioning to optimize branch prediction and icache

Backend supported options:
asm Assembly code (default)
cfg Control flow graph
ir Intermediate representation
ssa SSA code
report Report

The input file should be .start file. If you type into the profile option, the compiler would first finish all the optimization and then use the optimized program to profile. During the profiling, the compiler first instruments the program, test run it, and collect the profiling data. Next, compiler uses the data to optimize the program. At last, it outputs the code according to the backend option. All the above operations are done automatically.

Approach

Profiling

For basic block positioning, it requires the knowledge of counter for each edge, or it requires to solve Eprof problem. I choose the Eprof(Ecnt) method to solve this problem. The optimal way to profile with edge counters[1] is that:

1. Heuristically estimate the program and weight each edge.
2. According to the weight, generate the maximum spanning tree.
3. Instrument the edges that are not in the maximum spanning tree.

However, for the convenience, I instrument all the edges in the program since this is the most important part.

Basic Block Position

1. Top-down Algorithm

At first, I implement the top-down algorithm[2]. The principal idea of this algorithm is

1. First place the entry basic block for the procedure;
2. Choose the successors of current block connected with the largest counter;
3. If all successors have already been selected, pick among the unselected basic blocks the one with the largest connection to the already selected blocks;
4. Repeat step 2 and 3, until all the blocks are selected.

It's not hard to implement this algorithm. But it's surprising to find that this algorithm will deteriorate the performance and increase the branch mispredict for some program. Then I look into details that why the algorithm doesn't work.

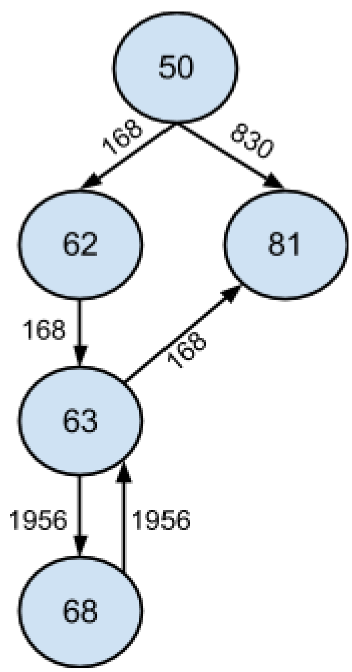

Figure 1: A partial control flow graph. The number shows the counter for each edge

Figure 1 shows an example that top-down algorithm will generate more branch mispredict. The original basic block order is 50->62->63->68->81. The branch mispredict happens in the edge 50->81 and 63->81 since those are forward edges and the predictor won't take the branch. Therefore, the total number of mispredict is 830+168=998.

However, the top-down algorithm will rearrange the block order into 50->81->62->63->68. Now the branch mispredict happens in the edge 63->68 since the 63->81 is a backward edge and the predict will take the branch. Then the mispredict increase to 1956 which turns out that the optimization makes the situation even worse. That's why I then implement bottom-up positioning algorithm and I'd like to compare the performance between them.

2. Bottom-up Positioning

The bottom-up positioning algorithm provides more freedom to place basic blocks. The basic idea is

1. Find the edge with the heaviest weight and connect this edge to an existing chain or create a new chain
2. After all edges are visited, merge the chains with a certain order.

I made a revise in merging the chains. The original algorithm[2] uses all conditional branches in a chain to imply the order between two chains. However, this method cannot promise to generate a directed acyclic graph. If there is a circle inside, you have to make a guess and this could potentially be a bad decision. Instead I give a weight for each implication. The weight is that the penalty, or the counter of mispredict, if chain C1 is after chain C2. This method could promise that there won't be a loop inside the graph of chains. Thus, we could use topology sort to generate a sequence.

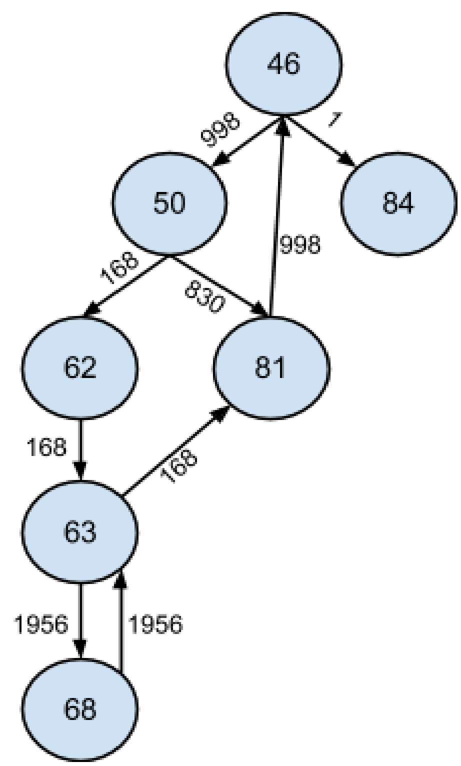

Figure 2: A partial control flow graph

However, this method still won't work for some case. Let's take the previous example with a little extension of the CFG (Figure 2). The current order is 46->50->62->63->68->81->84. The mispredict happens in 46->84, 50->81 and 63->81, 999 in total. Bottom-up positioning could generate the following two chains: {81->46->50->62->63->68}, {84}. Take a look at the first chain. Inside it has a mispredict 62->63 since 81 is in front of 63. This single mispredict has the weight of 1956, which is already greater the 999.

3. Bottom-up Positioning Improvement 1

I analyze the reason that the bottom-up positioning still fails is because it uses a backward edge 81->46 and makes 81 in front of 63. What if we ignore all the backward edges. This also makes sense for two reasons: (i) First, if we make the backward edges forward, it will break the origin order of program and make it difficult to understand; (ii) those backward edges won't be mispredicted since the predictor will assume the program will take this branch.
Actually this improvement works pretty well after I compare it between the above two algorithms. The result of measurement is displayed in the next section.

4. Bottom-up Positioning Improvement 2

This improvement also concerns about the backward edges. I think about if we make backward edges forward, it is very likely that we could remove a branch instruction and in result save the dynamic cycles. Besides the backward edges always have a greater weight in the loop. This might potentially save a great amount of time in branch. Therefore, I made another improvement of bottom-up positioning by increase the weight of backward edges by 1.

Performance Test

Table 1 shows the statistics of comparison among the above four algorithms. First we take a look at the program "sieve". We can find that both top-down and bottom-up without improvement make more branch mispredict. In compare, bottom-up with improvement 1 and 2 both decrease mispredict and dynamic cycles. Especially the improvement 2 has a significant improvement. However, focusing on the program "prime", the improvement 2 has more branch mispredict than improvement 1. But it still has a smaller dynamic cycles. So we can tell that removing branch instruction in the loop does save a lot of time, though it also depends on the branch mispredict penalty (I think here the start interpreter assigns too few penalty on the mispredict).

Program	Optimization	Dynamic cycles	Instruction count	icache miss	Branch mispredicts
points	origin	1737	455	15	2
	top-down	1737	455	15	2
	bottom-up	1737	455	15	2
	bottom-up(improve1)	1737	455	15	2
	bottom-up(improve2)	1719	437	15	2
mmm	origin	7371	4791	83	38
	top-down	7371	4791	83	38
	bottom-up	7435	4771	85	102
	bottom-up(improve1)	7371	4791	83	38
	bottom-up(improve2)	7390	4700	94	38
gcd	origin	300	178	12	2
	top-down	300	178	12	2
	bottom-up	300	178	12	2
	bottom-up(improve1)	300	178	12	2
	bottom-up(improve2)	290	168	12	2
hanoifibfac	origin	4548	4045	35	153
	top-down	4688	4191	35	147
	bottom-up	4540	4043	35	147
	bottom-up(improve1)	4540	4043	35	147
	bottom-up(improve2)	4540	4043	35	147
prime	origin	500595	405322	26	17161
	top-down	487954	406692	27	3140
	bottom-up	486758	398000	27	10636
	bottom-up(improve1)	487954	406692	27	3140
	bottom-up(improve2)	486758	398000	27	10636
rational	origin	3536	464	56	9
	top-down	3549	468	57	8
	bottom-up	3533	464	56	8
	bottom-up(improve1)	3533	464	56	8
	bottom-up(improve2)	3528	459	56	8
cproptest	origin	227	106	12	1
	top-down	227	106	12	1
	bottom-up	227	106	12	1
	bottom-up(improve1)	227	106	12	1
	bottom-up(improve2)	220	99	12	1
struct	origin	2950	394	67	2
	top-down	2950	394	67	2
	bottom-up	2950	394	67	2
	bottom-up(improve1)	2950	394	67	2
	bottom-up(improve2)	2946	390	67	2
sieve	origin	130914	90319	26	1831
	top-down	131546	90487	26	2295
	bottom-up	129552	88493	26	2295
	bottom-up(improve1)	130420	90487	26	1169
	bottom-up(improve2)	124979	85708	26	507
link	origin	638	388	14	20
	top-down	620	388	14	2
	bottom-up	620	388	14	2
	bottom-up(improve1)	620	388	14	2
	bottom-up(improve2)	620	388	14	2

Table 1: Performance comparison among four algorithms

Program	Optimization	Dynamic cycles	Instruction count	icache miss	Branch mispredicts
loop	top-down	37407894	37005550	11	402234
	bottom-up	41492118	36634258	11	4857750
	bottom-up(improve1)	37407894	37005550	11	402234
	bottom-up(improve2)	32581086	32178742	11	402234
richards	origin	26933770	16716459	574219	367278
	top-down	26159228	16373694	561209	83421
	bottom-up	26259236	16473702	561209	83421
	bottom-up(improve1)	26439098	16730588	552687	80946
	bottom-up(improve2)	26159228	16373694	561209	83421
test	origin	9050	8009	4	1001
	top-down	8050	8009	4	1
	bottom-up	8050	8009	4	1
	bottom-up(improve1)	8050	8009	4	1
	bottom-up(improve2)	9051	8010	4	1001
vnumtest	origin	537	144	39	3
	top-down	534	144	39	0
	bottom-up	534	144	39	0
	bottom-up(improve1)	534	144	39	0
	bottom-up(improve2)	534	144	39	0
class	origin	718	84	19	0
	top-down	718	84	19	0
	bottom-up	718	84	19	0
	bottom-up(improve1)	718	84	19	0
	bottom-up(improve2)	718	84	19	0
sort	origin	4950	3346	34	14
	top-down	4950	3346	34	14
	bottom-up	4976	3337	34	49
	bottom-up(improve1)	4950	3346	34	14
	bottom-up(improve2)	4979	3320	35	59
regslarge	origin	57325	16374	4095	1
	top-down	57325	16374	4095	1
	bottom-up	57325	16374	4095	1
	bottom-up(improve)	57325	16374	4095	1
	bottom-up(improve2)	57325	16374	4095	1

Table 1 (continued): Performance comparison among four algorithms

To make the comparison between four algorithms more direct, I selected several programs and compare the performance improvement refer to the original programs. In Figure 3, the value is calculated in the way: *improve(%) = (mispredict - mispredict') / mispredict,* where the *mispredict* is the number of origin mispredict, *mispredict'* is the improved result. From this chart, we can tell that the bottom-up with improvement 1 generally provide the most accurate branch prediction. In a few cases, they could make almost all branch prediction correct based on the profiling result.

In Figure 4, the value is calculated in the way: *improve(%)=cycle / cycle'*, where the *cycle* is the origin dynamic cycles and the *cycle'* is the improved result. From the char, we can find that the bottom-up with improvement 2 make the program run fastest in most cases. Especially for those loop incentive programs, the improvement 2 could have a significant improvement.

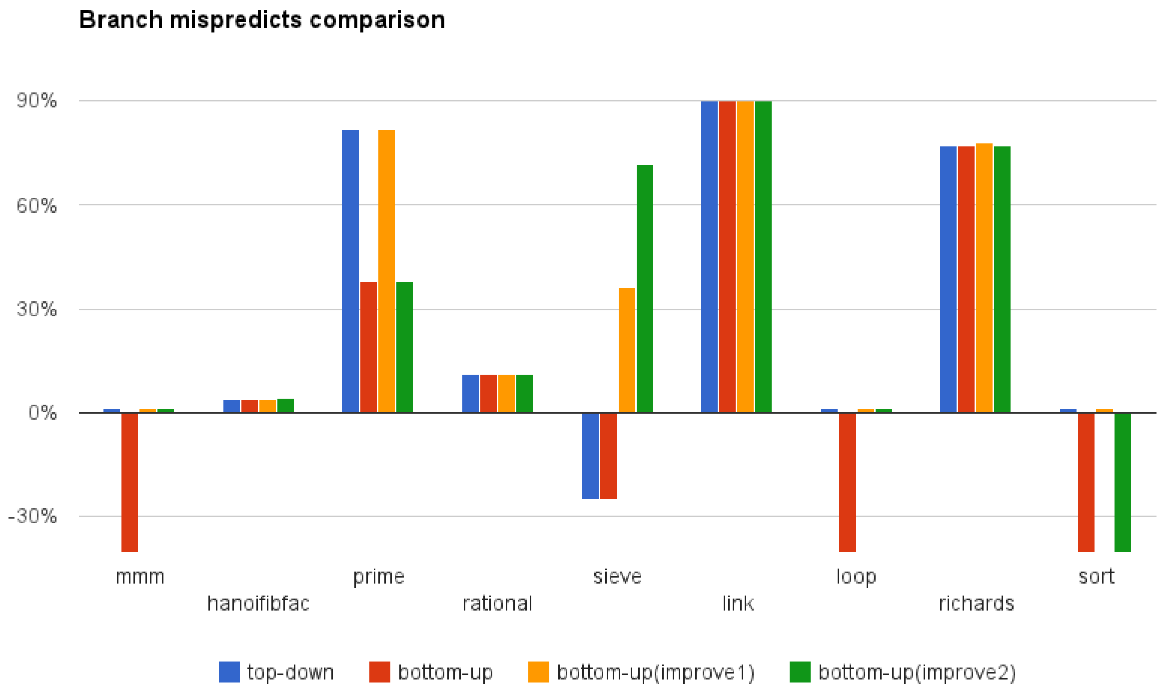

Figure 3: Branch mispredict comparison. The value refers to the original program.

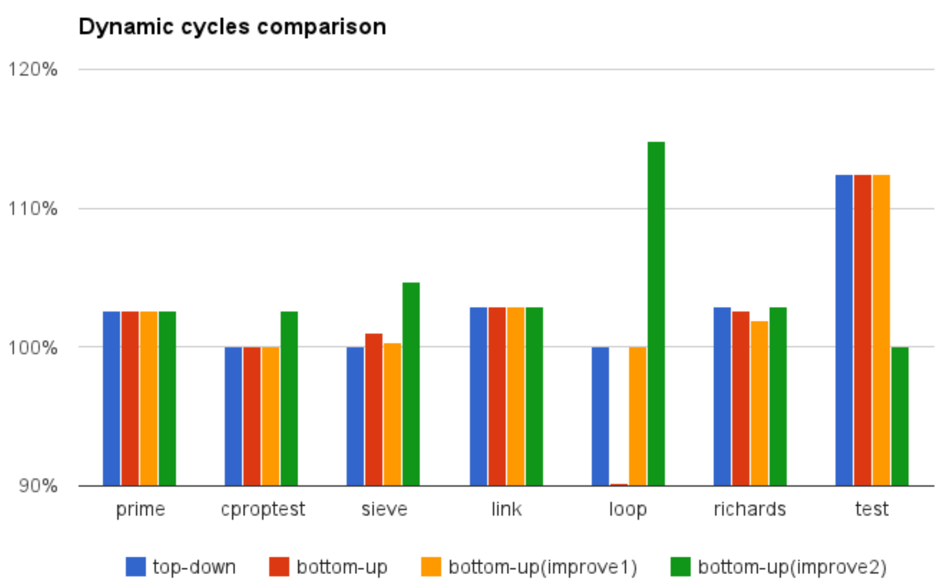

Figure 4: Dynamic cycles comparison. The value refers to the original program.

Summary

- Implement the profiling based code positioning optimization.
- Analyze the reason that the original two algorithms in [2] don’t work for some cases.
- Make two different improvement to the bottom-up positioning algorithm.
- Have elaborate performance tests of 4 algorithms and make implication from it.

* The final submitted version of code positioning uses the bottom-up positioning algorithm with improvement 2.

Reference

[1] Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. ACM Trans. Program. Lang. Syst. 16, 4 (July 1994), 1319-1360

[2] Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation (PLDI '90). ACM, New York, NY, USA, 16-27.