CS6241: Project 1 - Part 1

Zhen Li zhenli.craig@gatech.edu 2024-03-24

Contents

I did this alone.

1 Baseline

The statistics are generated with `./gen_baseline_stat.py`.

The llvm bytecode size are measured by python script `os.path.get_size()`.

The time are measured by a `time` based profiling tool `hyperfine`. It runs the program 2 times for warmup and then 10 times to take the average. (macOS seems do not have equivalent to `perf` on Linux.)

Bench		Bytecode Size (byte)	Mean User (ms)	Mean System (ms)	Mean Total (ms)
is	original	22784	16.154	0.477	17.005
	baseline	24800	20.670	0.541	21.650
	ratio	**108.8%**	**128.0%**	**113.6%**	**127.3%**
bfs	original	14384	0.230	0.308	0.747
	baseline	15728	0.301	0.366	0.848
	ratio	**109.3%**	**131.1%**	**118.6%**	**113.4%**
dither	original	31984	39.616	2.145	42.690
	baseline	34992	71.908	2.448	75.778
	ratio	**109.4%**	**181.5%**	**114.1%**	**177.5%**
jacobi-1d	original	5840	1247.679	0.969	1252.289
	baseline	6624	9595.287	1.897	9627.112
	ratio	**113.4%**	**769.1%**	**195.8%**	**768.8%**
check_elimination	original	3728	0.210	0.337	0.749
	baseline	4160	0.270	0.348	0.797
	ratio	**111.6%**	**128.3%**	**103.3%**	**106.5%**
check_modification	original	3744	0.208	0.333	0.743
	baseline	4176	0.278	0.355	0.859
	ratio	**111.5%**	**134.1%**	**106.6%**	**115.6%**

Table 1: Performance comparison between original programs and after the `check-ins` pass

2 Gupta's methods

2.1 Bytecode size and performance

The statistics are generated with `./gen_gupta_stat.py`. The measurements are the same as the baseline.

Bench		Bytecode Size (byte)		Mean User Time (ms)		Mean Total Time (ms)	
is	original	22784	100.0%	16.154	100.0%	17.005	100.0%
	baseline	24800	108.8%	20.670	128.0%	21.650	127.3%
	gupta	24272	106.5%	18.335	113.5%	19.246	113.2%
	(gupta - baseline)/original		**−2.3%**		**−14.4%**		**−14.1%**
bfs	original	14384	100.0%	0.230	100.0%	0.747	100.0%
	baseline	15728	109.3%	0.301	131.1%	0.848	113.4%
	gupta	14992	104.2%	0.295	128.5%	0.833	111.5%
	(gupta - baseline)/original		**−5.1%**		**−2.6%**		**−1.9%**
dither	original	31984	100.0%	39.616	100.0%	42.690	100.0%
	baseline	34992	109.4%	71.908	181.5%	75.778	177.5%
	gupta	32656	102.1%	50.055	126.3%	53.219	124.7%
	(gupta - baseline)/original		**−7.3%**		**−55.2%**		**−52.8%**
jacobi-1d	original	5840	100.0%	1247.679	100.0%	1252.289	100.0%
	baseline	6624	113.4%	9595.287	769.1%	9627.112	768.8%
	gupta	6368	109.0%	3063.382	245.5%	3080.304	246.0%
	(gupta - baseline)/original		**−4.4%**		**−523.5%**		**−522.8%**
check_elimination	original	3728	100.0%	0.210	100.0%	0.749	100.0%
	baseline	4160	111.6%	0.270	128.3%	0.797	106.5%
	gupta	4128	110.7%	0.289	137.7%	0.881	117.7%
	(gupta - baseline)/original		**−0.9%**		**9.4%**		**11.2%**
check_modification	original	3744	100.0%	0.208	100.0%	0.743	100.0%
	baseline	4176	111.5%	0.278	134.1%	0.859	115.6%
	gupta	4128	110.3%	0.308	148.2%	1.034	139.3%
	(gupta - baseline)/original		**−1.3%**		**14.1%**		**23.7%**

Table 2: Performance comparison between original programs and after the `check-opt` pass

2.2 Compile time check counts

Bench	After Insertion	After Modification	After Elimination	After Loop Hoisting	Percentage removed
is	56	58	34	23	**58.9%**
bfs	50	54	20	16	**68.0%**
dither	186	229	72	67	**64.0%**
jacobi-1d	22	22	8	7	**68.2%**
check_elimination	6	8	3	3	**50.0%**
check_modification	6	6	3	3	**50.0%**
malloc_1d_array	6	6	6	6	**0.0%**
static_1d_array	6	6	6	6	**0.0%**
global_1d_array	6	6	6	6	**0.0%**

Table 3: Bound check counts comparison between `check-ins` and `check-opt` pass (compile time)

2.3 Static spill code generated in each commenting on causes of performance degradations

2.4 Detailed analysis

About why certain benchmarks show a lot of removal opportunities whereas others dont, why removal is co-related to performance improvement in some cases whereas not co-related or less co-related in others and finally comparison of effectiveness