CSC 766 PR3 – Design Space Exploration for Speculative Optimizations

Objective

The goal of the project is to perform a design space exploration to find speculative optimizations which provide the most benefit on a given piece of code.

Abstract

Speedups from traditional optimizations have dried up. Hence, there is a need to explore other ways of achieving speedup. One such technique is speculative optimization. It relaxes *certain* constraints within the compiler, so that the code can be more aggressively optimized. The constraints existed, in the first place, to prevent loss of semantics under *any* input. Relaxing the constraints requires that there is a semantic-correctness checking mechanism. In order to get performance from the system, the aggressively optimized should provide speedup, while taking into account the occasional mis-speculation, and the associated rollback and recovery costs. In this project, I created such a system to explore the optimization space to find one or more n-tuples of optimizations which could give a performance benefit.

The main takeaways from the project are:

1. Speculation sites are crucial to the performance of the overall system.
2. The semantic-correctness checking mechanism needs to be light-weight, since it can inhibit application performance, even when there is no violation.

The Idea

The granularity of speculation chosen for this project is the function. This design choice was made because most optimizations within the compiler work at the global (function) level. It is easier for a compiler to optimize, versus at the module level, since each function can (and typically does) have multiple call sites. Performing inter-procedural optimization would require validating the transformation holds for all the sites, in terms of correctness and performance.

A granularity smaller than a function could have been chosen for this project, but a smaller region would restrict the optimizations that the compiler can perform on the code. Moreover, choosing a region which has multiple entrances and multiple exits would restrict the compiler.

The sites of speculation were given by the alias analysis queries of the optimizations. The response to these queries, in the non-speculative case, would either be No-Alias, May-Alias or Must-Alias. Since the compiler cannot prove/disprove the existence of a dependence statically (due to control and data flow of the program, known only at runtime), it has to respond to the queries, with a May-Alias answer. However, previous works have shown that many of these *May-Alias* queries, do not actually occur at runtime, a significant portion of the time. Techniques which have statically tried to predict the probability of occurrence of these dependences have been inaccurate. Hence, a profiling technique is required, in order to obtain the probabilities of these dependences.

The probability needs to be first gathered, before any speculation can happen, because without such probabilities, marking pairs for speculation is not likely to yield consistent results. In my test with the SPEC2000 benchmark suite, I found the mis-speculation percentage was very quite when I tried to speculation on all of the dependences. Since the functions might have hundreds of queries each, performing an exhaustive search for the low probability ones would be very expensive. The profiler used to find the relationships is described in the next section.

Once the infrequently occurring relationships are known, the compiler can now speculate on these dependences. The original design was to perform speculation on these dependences using various optimizations. However, when implementing the system, I encountered a large road block, in the form of a semantic-checking system. The system is required to ensure correctness when the program is in a speculative region. All the sites at which the compiler *could* have speculated, should be checked each time their parent function is speculated upon. The current mechanism for instrumenting the pairs involves inserting a function at each of the two instructions involved, and checking for an alias at the end of the function. However, this mechanism proved to be too heavy for the benchmarks, and the instrumentation code itself was many times the size of the actual application code. This negated any impact that the speculative optimizations could have had on the code. In the future work section, I have listed ways to make the instrumentation cheap, so that the optimizations have a chance at providing speedup.

The Profiler

The profiler has been implemented using signatures (bloom-filter based structures). Bloom-filters are inaccurate means of storage, providing quick insertion and disambiguation. Their storage contains a super set of the inserted data. Hence, on any given disambiguation, they suffer from false positive rate. However, this rate can be managed to within 10% of all the answers. The signatures implemented in this profiler have been used previously, and reported a 10% false positive rate.

The profiler has been implemented to find the probability of dependence between a pair of instructions. The structure used to maintain the signatures is a hash-map, containing one signature for each of the instructions forming the pair. The current scheme of assigning signatures, gives each unique LHS of a pair, a PrimeID. Hence, multiple RHS pairs with the same LHS, share a single signature. This technique allows for fast disambiguation, with a tradeoff in terms of accuracy and space. The problem with this scheme is that multiple RHS entries share the same signature, which can cause pollution, and hence a higher false positive rate. A better technique is being investigated for future implementations.

The Simulator

In order to provide timing results, and perform runtime-checking cheaply, an execution driven cache simulator was implemented. The simulator uses a single-level cache model to calculate the Time-Per-Instruction (TPI) and Instructions-Per-Cycle (IPC). The sematic-checking system is very similar to the profiling system above, and uses signatures to check for any conflicts. A safe memory checkpoint is taken before speculation begins, and in case of a conflict, execution is rolled back to the checkpoint.

The Complete System

The system implemented consists of a profiler, a compiler pass for cloning functions, and an execution-driven timing simulator. The profiler and simulator are implemented in PIN, whereas the compiler pass has been written in LLVM. The target architecture for the compiler is x86, since PIN can decode x86 instructions. The pass for cloning function was described in PR2.

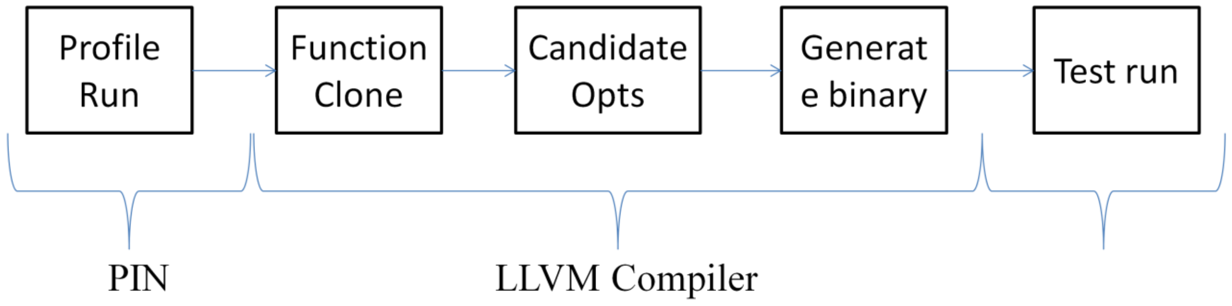

Figure 1. The Complete System

The benchmark suite chosen for testing the system was the SPEC2000. The benchmarks and the functions chosen for speculation are listed in Table 1.

Table 1. Benchmarks and Functions chosen

Benchmark	Function	Percentage of Execution Time
Ammp	Eval	5.7%
Gcc	Propagate_block	13.75% (Largest)
Gzip	Inflate_codes	9.69%
Parser	Prune_match	6.8%
Twolf	New_dbox_a	32.48% (Largest)
Vpr	Update_bb	23.27%
Microbench	functionOne	48.67%

The largest function was not chosen in some benchmarks because it either directly or indirectly called itself. In the current system, recursion is not supported, since supporting it would require an exploration of the depth to which speculation should be performed, and that is beyond the scope of this project.

The main problem that I encountered with the project was the overhead of the instrumentation. Although I was able to find pairs which had low aliasing probabilities, the speedup from the optimization was not visible due to the high cost of instrumentation. The figures below highlight this point.

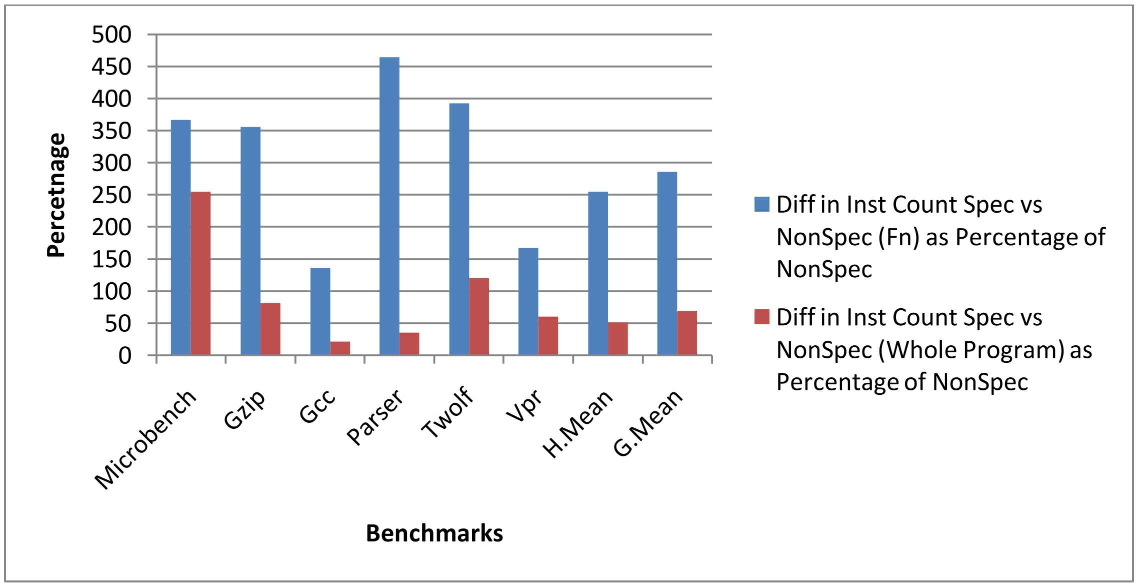

Figure 2. Overhead due to Instrumentation

Figure 2 above shows the overhead, in terms of percentage, the *difference* in instructions contributed by the *instrumentation only* to the overall benchmark. The harmonic mean of the overhead is 250%. This means that for every single instruction in the original program, twenty five instructions have been executed in the instrumented program. Considering that the instrumentation was only done on *one function per program*, this overhead is huge. Without addressing this point, further exploration would mostly likely be, pointless.

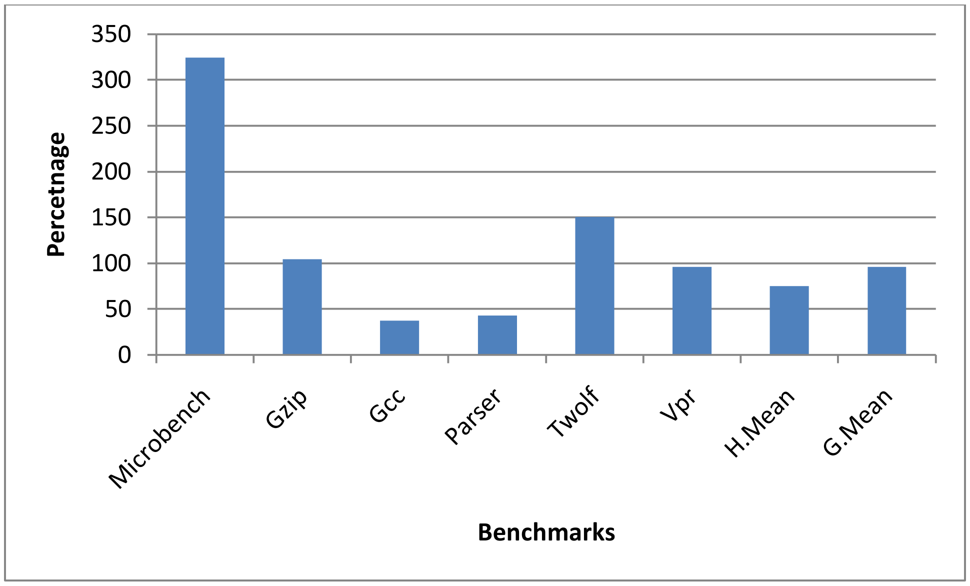

Figure 3. Speculative Insts as a percentage of the all dynamic instructions

Figure 3 shows the number of instructions in the speculative version of the function as a percentage of the total instructions in the program. Here also, we see that the number of instructions in the function exceeds those of the original benchmark in four out of six benchmarks.

The figure below compares the IPC for the case where GVN is applied to the benchmark after cloning, with one where it is not optimized. The IPC is the same even after applying GVN. This can be attributed to the function calls inserted in the benchmark, which probably inhibit GVN from performing the necessary transformations. Another possible reason is that the current system only captures the first degree queries made by the optimizations. Based on the initial query results, many more queries are made by the optimizations. These are not profiled, and hence, not speculated upon, by the current system.

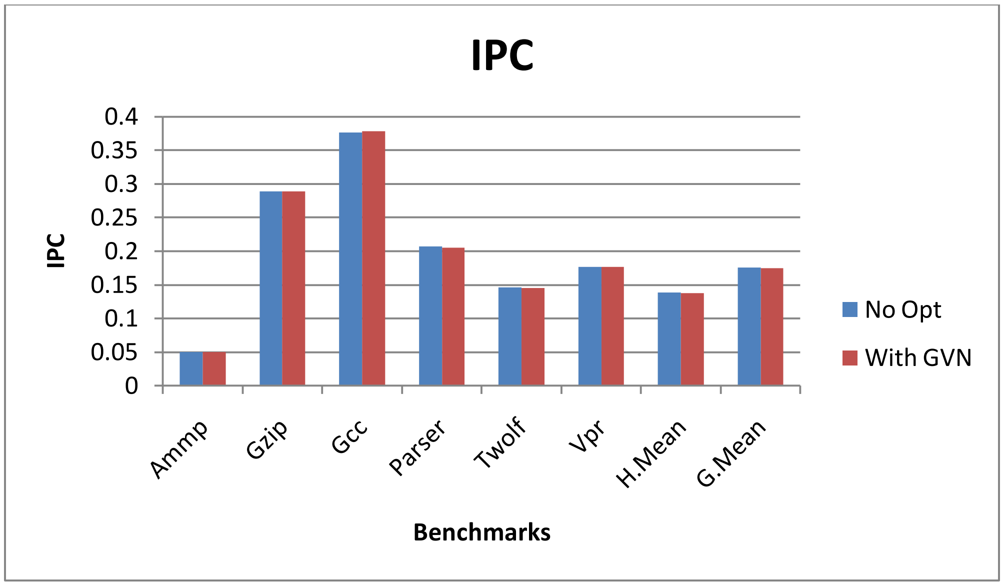

Figure 4. IPC with and without GVN enabled

Problems and Future Directions

The problems and possible solutions to each of them are listed below.

1. The instrumentation overhead for semantic-checking is large, and needs to be tackled to bring the instruction counts for the speculative version to the same range as the non-speculative one, before any optimization is performed. Using instructions, like those proposed in SoftSig could decrease the overhead for the benchmarks. I am currently exploring the possibility of inserting a special instruction, or an unused instruction in the x86 ISA, to schedule the insertion and disambiguation operations. Other schemes which can decrease the number of signatures used, and/or the number of comparisons made can also improve the performance of the system.
2. Capturing higher order alias analysis queries from optimizations may be crucial to getting the optimizations to perform aggressive optimizations. Currently, the system only tracks the first order queries. Responses to the first order queries cause another set of queries to be posed by the optimizations. Answers to these, and even higher order queries might hold the key to gaining performance from the system.
3. The optimizations could be making queries for certain pairs, but might not eventually, perform any transformation on them. Such queries could cause needless mis-speculation, since there was not transformation performed based on the answer to the queries. This can be rectified by tracking the pairs which actually contribute to transformations, and removing instrumentation for those that don’t.