Summary Part 1

An Infrastructure for Adaptive Dynamic Optimizations

Mrinal Nath (ID: 3307043)
February 28, 2005

What are the problems solved by this paper?

Modern software heavily uses dynamic class loading, shared libraries and runtime binding. Due to such techniques, static compilers can not analyze the whole program in sufficient detail. Also, their predictions of runtime program behavior have limited accuracy. Profiling can help only for the common or frequently used cases, but does not provide information about dynamically changing program behavior.

Therefore, dynamic optimizations of the code can be very useful, since they have access to runtime information about the program behavior. However, an important limitation of dynamic optimization is that it can incur a significant overhead, which can actually result in a slowdown instead of a speed up of the program being dynamically optimized.

This paper proposes a dynamic optimization technique which is highly efficient. Additionally, it is flexible (different types of optimizations can be done), easy-to-use (a good API is provided), and general in scope (so it can be used for non-optimization related purposes). An important feature of the proposed techniques is that it is adaptive – code can be re-optimized easily even after it is placed in the code cache.

What are the approaches attempted by this paper?

The proposed optimization infrastructure is built on a dynamic optimizer called DynamoRIO. It copies basic blocks into code caches for inspection and optimization. Frequently executed sequences of basic blocks are stitched to form a ‘trace’. Private code caches are used for each thread, which helps avoid inter-thread synchronization overhead. To achieve efficiency, two techniques are used: (1) limiting the optimization units to linear streams of code (basic blocks or traces) so that simpler optimization strategies can be used and (2) using adaptive levels of detail for representing instructions (so that minimum space is used and unnecessary conversions to extract details are avoided).

The API provides a rich set of functions and data structures. Instruction generation is simplified by providing macros for each IA-32 instruction. It also provides a ‘clean-up’ mechanism in case some of the predictions used in the optimizations are proven wrong.

Re-optimization of code is possible by using information provided by runtime profiling. Thus, adaptive optimizations can be made. It is possible for the client to build custom traces, thereby increasing the flexibility and power of the DynamoRIO tool.

What are the main conclusions of this paper?

The authors performed different types optimizations and the results shows that there can be significant performance gain if the right type of optimizations are done on a particular program. A ‘traditional’ optimization was the removal of redundant loads. This optimization gave good speedup for FP programs. An example of architecture-specific optimization was the conversion of inc/dec instructions to add/sub 1 instructions for Pentium 4. This can be considered as a strength reduction optimization and it gave speedups on a number of benchmarks. An adaptive optimization was the profiling of
indirect branch target instructions and using conditional jumps to avoid the hash table look-up. This optimization performed well for integer programs. Similarly, generating custom traces for optimization helped integer programs.

Overall, they reported 12% improvement (w.r.t the native machine) in runtime for FP programs when all optimizations are used. But when integer programs are included, there is no improvement w.r.t the native machine – which is still a 12% improvement w.r.t the base DynamoRIO scheme.

Thus we can conclude that the proposed techniques are efficient (there is room for improvement), flexible, adaptive and general. The techniques do not rely on any special hardware, OS or compiler support.