Dynamic Compilation-I

Lecture #13: Tuesday, 13 May 2003
Lecturer: Navneet Aron, Sorav Bansal
Scribe: Varun Malhotra, Janani Ravi

1 Introduction

With the modern software heavily utilizing shared libraries, dynamic class loading (for instance in Java) and runtime binding, the scope of static compiler analysis is becoming restrictive. The optimizations by the static compiler are limited by the information available at static compile time. Using profiling information may improve the accuracy of the information of run-time program behavior but this approach doesn't hold promise for all general purpose programs. This motivates shifting of optimizations from static compilation time to runtime.

Additionally, these days software is being shipped as a collection of DLLs rather than a single executable and complexity is shifting from hardware to software. Dynamic compilation exploits program staging by postponing certain optimizations until initial stages of execution have been completed which permits accurate prediction of program behavior. Dynamic Compiler can also optimize binaries, that have not been compiled by the vendors with high levels of static optimization in order to make debugging easy.

Several optimizations are suited to a dynamic optimization framework like:

- *Adaptive* optimizations require instant responses to changes in program behavior, for instance, online profiling techniques can be used to identify *hot* regions allowing optimizations to concentrate on areas where they can be most effective.
- *Architecture-specific* code transformations can be performed at runtime without restricting the executable to target a single processor and thus, allowing the executable to remain generic.
- Inter-module optimizations for programs using shared and dynamically loaded libraries can be done by a dynamic compiler as all the code is available to a dynamic optimizer.

However, since the dynamic compiler uses cycles and other resources (like memory) at run time, the overhead of the optimization must be optimized before any improvements can be seen. This limits the scope of optimizations that can be done at runtime, and makes the efficiency of the optimizations extremely critical.

2 Summary of the papers

2.1 The JRPM System for Dynamically Parallelizing Java Programs

Java Runtime Parallelizing Machine (JRPM)[1] is a system for parallelizing sequential programs automatically. JRPM consists of following components:

- Hydra Chip Multiprocessor - provides low latency inter-processor communication
- Test profiler: provides the mechanism for obtaining statistical data about dynamic dependencies, thread sizes and amount of buffering space needed for speculative state.
- Thread Level Speculation(TLS) support: Allows converting an otherwise sequential program into parallel execution model while ensuring that the memory accesses between threads maintain original sequential program order.
- Java Virtual Machine(VM): Provides an environment where dynamic optimizations can be performed without modifying the source binaries.

Hydra Chip multiprocessor(CMP) provides following hardware support for thread level (or loop level) speculation. It has a co-processor attached to each main processor for controlling speculative state. Apart from that, it has speculative tag bits for the purpose of memory disambiguation between threads and has a L2 cache to store speculative data. In order to tackle hazards caused by various data-dependencies, JRPM employs various techniques. It uses forwarding of data to threads in sequential order and restarting threads to deal with RAW (read after write) hazards. For WAW (write after write) violations, buffered writes are made available only to sequentially later threads. For the case of WAR (write after read) hazards, buffering and committing the speculative writes is done in program order.

An important consideration in JRPM is to choose speculative thread loops(STL), a consequence of which is to identify the thread size. Small loops are good because they have lesser chances of speculative buffer overflow and they have lesser penalties if RAW hazards occur later in execution. On the other hand, larger threads are benefiacial as they have lesser overheads. Apart from looking at the thread size, load dependency, state overflow analysis are also done to identify best loops to be chosen as STL. Certain control routines are inserted by the compiler for speculative execution after the selection of loops (STL). A faster compilation is achieved by interleaving compilation stages and by minimizing compiler passes. To enhance the performance of this speculative code, further optimizations are done. Non communication loop inductors are used by assigning loop indices to processors at static time and incrementing them by 4 after every iteration (# processors in Hydra = 4). Several other optimizations like loop invariant register allocation

, resettable non-communicating loop inductors, thread synchronization locks, reduction operator optimization, multilevel STL decomposition, hoisted startup and shutdown are used.

While handling exceptions, only the exceptions caused by a non speculative thread need to be handled. For the purpose of garbage collection, mark and sweep is used. An optimization used in JRPM is to split the free list of objects into several free lists, one for each processor. Synchronization mechanism in original sequential code are modified for speculative execution to avoid unnecessary synchronization. JRPM claims to achieve a speedup of 3-4 for floating point applications, 2-3 for multimedia applications and 1.5-2.5 for integer applications with Hydra (4 processors). Although, JRPM achieves good overall performance with low overheads of profiling and dynamic compilation but the target applications are restricted to a subset of programs. For instance, it is not possible to parallelize programs with system calls in the critical code regions.

2.2 Dynamo: A Transparent Dynamic Optimization System

Heavyweight static compiler optimization techniques are limited in their scope and capability. This is particularly true with the growing trend towards Object-Oriented languages (using dynamic linking) and dynamic code generation environments (for instance, JAVA JITs and dynamic binary translators).

Dynamo [2] is a software dynamic optimization system that is capable of transparently improving the performance of a native instruction stream as it executes on the processor. It requires no user input, no special compiler or OS or any other special hardware support, and can be used on even legacy native binaries.

Any runtime scheme needs not only to improve the executing native program but also recoup the overhead of its own operation. The paper shows that Dynamo is able to recoup the overhead even in extreme situations when the binary has been statically optimized by the compiler. They demonstrate not just the potential, but the limits of the Dynamo scheme.

The optimizations targeted by Dynamo are:

- Redundancies across cross-static program boundaries (e.g. Procedure Calls)

- Instruction Cache Utilization

These objectives cannot be achieved by a static compiler, especially if the procedures are linked *late*. Dynamo achieves this by identifying frequently executing traces in the program and optimizing the traces. Since, the traces could span procedure-call boundaries, this opens up a vast array of possible optimizations. Dynamo takes special care to ensure

that the traces are *Join-Free*. This opens up more opportunities for optimizations inside the trace since the optimizer does not need to worry about crossing join-points. They perform conventional optimizations, such as, copy propagation, strength reduction, etc. Furthermore, join-free property of traces introduces new redundancies which is exploited by Dynamo.

The main operations of Dynamo are:

- Trace Selection - Employs a very lightweight scheme to profile only tails of loops
- Trace Optimization - Both conventional and aggressive optimizations are performed on the trace
- Fragment Code Generation and Linking - Fragments are made out of traces and they are linked into the Fragment Cache. Linking can be an expensive process.
- Fragment Cache Management - Since the cache needs to be small, the cache is flushed on observing heavy workload changes.
- Signal Handling - Signals need to be handled in the Interpreter mode. Asynchronous signals are queued until next switch to the interpreter mode. Dealing with synchronous signals is hard. They split the optimizations into *conservative* and *aggressive*. Aggressive optimizations do not guarantee precise exception-handling.

The main overheads of the scheme are:

- Startup cost due to interpretation
- Counter updates and storage
- Linking Fragments
- Redundant copies of code
- Context Switching between Interpreter and Fragment Cache
- Cache Replacement

The paper does a very good job at demonstrating the limits of the Dynamo system. It shows that even in extreme situations, Dynamo can help improve performance. We would have liked to see their experiments to be run on different machines since the performance improvement depends heavily on factors like branch-misprediction penalty, I-cache size, TLB size, etc. Moreover, we would have liked to see them experiment with different design choices. In particular, we would like to see how the performance is effected by using different profiling mechanisms and cache replacement schemes.

3 Discussion

Having looked at the details of the JRPM and the Dynamo system, and the various benefits of dynamic compilation, we will now discuss the various problems that are hard to tackle in a dynamic compilation environment.

3.1 What's hard in a dynamic compilation environment

One of the first problems that most systems need to deal with is *self-modifying* code, which can be exceedingly difficult to detect at run-time. Then, there could be *self-referential* code, which refers to itself as data. This is a problem for any dynamic compilation system because the system needs to make its own private copy of the code on which various optimizations are carried out. On top of that, the system has to detect and ensure that when the code references itself as data the original code needs to be fetched, otherwise the optimized version needs to accessed. Another problem is that of handling precise exceptions. Different systems use different strategies to counter this problem, for instance,

- In JRPM, the exceptions generated in any thread are serviced only when the thread is no longer speculative.

- IBM's Daisy allows two kind of load instructions: *real* and *speculative*. They use speculative load instructions instead of loads when they are prefetching. Speculative loads function just like binding, non-blocking loads, but they do not cause exceptions if they cannot be satisfied; instead, if an exception is raised, it is serviced at the point of the real load instruction in the program.

- Transmeta's Crusoe again uses the speculative load instruction for prefetching data but the difference is that the speculative load sets a flag (or poison bit) as a side-effect in case of an exception. This flag is checked for an exception at the point of the actual load instruction.

- Dynamo has some checkpoints in the code where it saves the state of the system and it recreates the state using the original code in case an exception occurred. It has some shadow registers and memory to store the state of the system.

One of the problems that is usually overlooked while discussing dynamic compilation is that it may reveal *non-determinism* in an application which could be really hard to debug. This can force the compiler to operate in a conservative mode.

3.2 Translation or Optimized code

Dynamic compilation systems could function as interpreters without applying any optimizations. This scenario is encountered in most of the simulation environments where the

simulators simply translate the input code without actually optimizing the code since the process of translation is very light-weight. However, most of the systems (like Dynamo) use a combination of translation and optimizing techniques. Dynamo uses translation to do online profiling and only when some piece of code has been identified as *hot*, Dynamo actually compiles an optimized version of that code segment. On the other hand, JRPM doesn't use translation but uses dynamic compilation to optimize the code segments which have been identified as threads. One of the differences between Dynamo and the JRPM system is that JRPM *instruments* the code to speculate on the regions which are prospective/potential threads while Dynamo uses counters to keep a check if some code is used frequently or not. *Instrumentation* could lead to bloating of code and sometimes could hinder the performance gains but instrumenting code in *rare* cases can prove to be beneficial.

It is important to understand the trade-offs of translation and optimizing the code. The interpretation and profiling of the code before actual optimizations are carried out could have a performance impact, as this adds an additional step to the entire compilation process. The optimizations made to the code have to be good enough to compensate for this overhead incurred. Moreover, the system should look for optimizations in those code segments which take up most of the running time (target Amdahl's Law).

Dynamo uses a separate buffer/cache to store the dynamically compiled code which introduces a question of what cache replacement policies to choose from. The problem with replacing a cache block in this scenario is that whenever a code block gets replaced there can be lot of other code segments linked to that block. Unlinking of components in a cache has a very high overhead and moreover, it is difficult to flush strongly connected (or linked) components. Other fancy schemes could be devised wherein we identify the *hot* segments and don't replace them. This overhead of keeping account of what is *hot* and what is not is smaller for larger fragments but the disadvantage is that if the fragment is large subparts of the fragment may become cold. Having a fine-grain control over the cache fragments can improve the efficiency of the cache. To counter all these problems, Dynamo uses a simple scheme where it flushes this cache periodically. One might be tempted to design large caches in this scenario, so that we don't have to replace very often but the flip side is that it takes more time to flush if the cache is large.

Another question to think about is the granularity at which the various compiler optimizations should work (say parallelization) and the trade-offs associated with each approach. For instance, the JRPM system works the granularity of loops and targets loops as potential candidates of parallelization. However, there is a trade-off while determining the size of the loops on which we are speculating. Larger loops would mean more code being executed in a single thread and there would be greater chance of dependencies between loops while on the other hand, smaller loops would mean greater independence but greater overhead of thread synchronizations. In the Dynamo system, optimizations

are carried out on superblocks which are identified by merging basic blocks on the most frequently traversed paths in the control flow graphs. This allows optimizations like redundancy checking, copy propagation, code elimination across basic block boundaries. Dynamo uses fairly simple optimization techniques, and thus, looking at a larger code window proves useful. However, since the complexity of most of the compiler optimizations is super-linear, large code window greatly increases complexity and compile time grows extremely fast increasing the overhead of the runtime system.

3.2.1 Why only loops why not function calls

Most of the papers about exploiting parallelism in the programs target loops, however, one could argue that we could parallelize function calls/invocations. What is it that makes it easier to parallelize (or even speculate) loops rather than function calls ?

- Characteristics of loops:
  - The is more inherent parallelism in loops.
  - Execution time for loops is the same for every iteration so there is no need to worry about load-balancing.
- Characteristics of functions:
  - There are too many types of functions to care of.
  - There can be trouble dealing with return types in functions.
  - Functions do not have well-defined boundaries and recursive functions are very hard to deal with.
  - Functions have smaller regions for speculation and moreover, the behavior of the functions is highly dependent on the application and the input dataset.

After having said that, the driving factor of whether we should speculate on functions or not is highly dependent on the type of information we obtain at runtime. Techniques like constant propagation and partial method evaluation targeting *hot* methods can be very effective in getting performance gains.

4 Conclusions

Dynamic compilation has its own set of benefits and overheads. However, the ideal situation is where the static and the dynamic compiler complement each other rather than using one or the other. The static compiler has the luxury of looking at the source code (which the dynamic compiler might not have) and it can give useful hints to the dynamic compiler in the form of annotations in the code making the job of dynamic compiler easier.

After discussing various issues about dynamic compilation, the main conclusions are as follows:

- Optimize as per hints provided by the static compiler
- Follow Amdahl's law and look for those optimizations in a program, which take up the most execution time.
- Use profiling to better understand the source code and thus improve optimizations performed

References

[1] The JRPM System for Dynamically Parallelizing Java Programs, *M. Chen* and *K. Olukotun*, Proceedings of the 30^{th} International Symposium on Computer Architecture, San Diego, CA, June 2003

[2] Dynamo: A Transparent Dynamic Optimization System, *V. Bala*, *E. Duesterwald* and *S. Banerjia*, Proceedings of the Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000