EE 392c Advanced Topic on Computer Architecture
Lecture 14: Dynamic Compilation 2
Stanford University May 24, 2003

Introduction

Dynamic compilation has the potential for performance that surpasses that what is possible with static compilation because a dynamic compiler has access to dynamic information. Some examples of dynamic information are runtime constants, which are variables that have a single value, or relatively few values, during program execution, and trace information, which are the sequences of branches that are taken at run time. We cover two systems that take advantage of these two types of information: DyC, which is an extension to C that allows programmers to annotate their programs to take advantage of runtime constants, and DELI, which provides an interface for fine-grained control of dynamic execution and traces.

DyC Summary

DyC is a selective dynamic compilation system that is based on standard C. It includes support for partial evaluation at run time. Programmers annotate their source code to identify static variables (variables that have a single value, or relatively few values, during program execution) on which many calculations depend. DyC then automatically determines which parts of the program downstream of the annotations can be optimized based on the static variables' values.

DyC uses the technique of staged compilation to efficiently support dynamic code generation. In staged compilation, most of the planning for dynamic compilation and optimization is performed at static compile time. The static compiler generates a specialized code generator for the dynamically compiled region. This code generator takes the values of the static variables and generates a specialized version of the code based on those values.

DyC is powerful because it provides an easy mechanism to selectively add dynamic compilation to standard C programs. The annotations are straightforward to add to the program, and much of the work about whether values are static or dynamic is done by the compiler. The overhead for dynamic compilation is very low due to the use of staged compilation. The breakeven points (the time during the program execution at which the benefit of dynamic compilation outweighs the cost) are reasonable, and they have shown actual performance improvements on real benchmarks.

On the other hand, it is sometimes difficult for the programmer to know where to put the annotations. Furthermore, in many cases putting the annotations in the wrong place will hurt the performance of the program significantly. The process of optimization is mostly a process of trial-and-error - the programmer guesses where to put annotations and then executes the program to see if any speedup was achieved. There is no notion of dynamic feedback or adaptive dynamic compilation in DyC. All decisions are hard coded.

Deli Summary

The traditional methods to improve the performance of code in different environments have been to cache, link, and execute a private copy of the code. Caching and linking has been employed to avoid the repetition of work done to produce the executable itself (dynamic binary translation) or to allow performance enhancing translations to be made to the private copy of the code (dynamic optimization).

The DELI extracts the underlying control functionality for caching and linking and opens it up to the OS and higher layers of system software and applications through an explicit programming interface. This allows the clients fine-grained control over the execution of programs and the opportunity to observe and manipulate every single instruction just before it runs.

The DELI transparently or explicitly (via the interface) takes over control of the running application and transports the executing code from the loaded application image into its code cache, from where it eventually executes. By way of this transformation, the DELI enables services such as translation, optimization, sandboxing, code patching, safety checking, hardware virtualization, remote code, and data streaming, and more. Despite the great variety of these capabilities, the DELI leverages the investment of building a single well-designed caching and linking mechanism across these many different uses.

DELI provides dynamic code optimization as a service to emulators so that these systems can be designed and implemented independent of the hardware they run on. The dynamic optimization is also provided as a function to the client applications and under client instructions can extract optimizations not available to optimizers which rely purely on binary text. DELI is language independent and does not require a special code format and can handle legacy code.

Overview of the DELI System

The DELI layer has 3 main components

- Application Programming Interface
- Binary Level Translation
- Hardware Abstraction Module

The DELI API

This provides the basic code caching and linking service as well as the infrastructure to support dynamic code transformations to the running client applications. This interface provides functions which allow the DELI to run with or without user control, allows registering of callback functions which can be associated with an event or fragment among others.

The Binary Level Translation Layer

This contains and manages a set of code caches. Performance is delivered using linking of fragments which avoids unnecessary code cache exits and dynamic optimizations performed on the DELI Intermediate Representation (DELIR). Optimizations can be applied to the fragment when it is first emitted or when it becomes "hot".

Hardware Abstraction Module

This provides a virtualized view of the hardware to the DELI system and other higher level applications. It contains static and dynamic components. The static components include memory mappings and globally defined hooks for handling exceptions and interrupts. Dynamic components are the HAM page translation tables.

Services to the Client

DELI offers the following categories of services to the client, program manipulation, program observation and emulation. Since the DELI code is the last to touch the code before it is executed it can replace instructions (dynamic code patching) verify legality of instructions (sandboxing). Its interface allows emulation of legacy operating systems and also mixing of native and emulated code.

Case Study: Emulating the PocketPC

This used the DELIverX emulation system, the emulated processor was a Hitachi SH3 and the target processor was a 4-wide VLIW machine. Performance numbers for execution time showed that emulation with optimizations performed better than the native emulated machine and that mixing native and emulated code or completely running native code showed very good improvements in execution time.

Critique

This paper compared performance numbers of an emulated and native processor where the native processor was much more powerful in terms of speed, cache and TLB size.
Also performance was measured purely in execution time the memory footprint of the program on the cache was not discussed. The native processor was a simulated machine which was compared against a real processor.

Discussion

The advantage of user annotations

The programmers are aware of the code structure and therefore can come up with annotations for optimization quite easily. Thus there is no overhead for dynamically figuring out what to optimize. Furthermore, the warm up period can sometimes be eliminated.

The disadvantage of user annotations

Despite the fact that annotation is quite easy to do for the programmer, it is still quite tedious and less accurate than information a dynamic profiler can provide. Deli, on the other hand can figure out the "hot spots" of programs automatically, i.e., without the programmer's involvement. Furthermore, some annotations could hurt performance if a programmer creates a wrong annotation or she is not careful enough to add them in the correct place.

Code size will increase after compiling as well due to "dynamically inserted code", which might be a concern in some domains.

Yet the most important of its shortcomings is that techniques as Dyc uses have restricted scope. Firstly, it assumes that one has access to the source code so many legacy binary libraries cannot be optimized. Secondly, it is blind to the architecture details in which many other opportunities exist for optimization, such as caches.

A Happy Medium

One would imagine a scheme that combines the benefits of both approaches of Dyc and DELI – i.e., one runs on top of the other. On one hand, users give strong hints as to where to optimize. On the other hand, dynamic profiling will further optimize for run time conditions.

Opportunities for Dynamic Optimization

Memory disambiguation is one potential domain. Static compilers have limited information and therefore conservatively assume some kind of dependencies exist. As a result, they tend to under optimize for such conditions.

Resource allocation is also a good area a dynamic complier can look at. For example, if the program is running matrix operation and threads are blocking on one another's data, a dynamic complier can schedule some other threads to utilize the waiting cycles. In that sense, this dynamic complier can be part of the OS. Others include cache optimization,

considering the dynamic complier has all the information about cache access patterns, etc.

Comparing DyC and DELI

1. Since DyC does staged compilation, it can look at large and complex blocks. DELI is limited in the sizes of traces that it can consider.
2. User annotations in DyC enable it to focus on the promising parts of the code. DELI has to profile the code for this purpose.
3. DyC can increase the size of the binary significantly.
4. DELI bails out if nothing optimizations are not working. This feature is useful to have and the DyC framework as in the paper does not mention how this feature can be incorporated.
5. DyC is affected by errors in user annotations.
6. As mentioned in the DyC summary, an annotation-based approach requires availability to source code which is often not an option with third-party software and legacy binaries.
7. There are many fruitful areas of dynamic optimization such as copy propagation into libraries for which the DELI approach is helpful.

It was pointed out that the DyC and DELI /Dynamo approaches are often orthogonal and complimentary in many respects and it might be worthwhile to consider building a system where they coexist, presumably DyC running on top of DELI.

Portability of DELI

Since DELI is packaged as a generic service, portability of the system is an important concern. The layered design of the system is such that the API and BLT layers are portable so that only the HAM layer would need to be rewritten while porting DELI to another architecture. The features of the HAM layer were discussed. HAM would need to abstract the details of virtual memory, IO, interrupts, exceptions, cache-coherence, etc. Clearly the HAM layer would need to be developed by someone with a good understanding of the underlying architecture. Further discussion on the design on HAM for CMPs is summarized later.

DELI for Dynamic Parallelization and Reconfiguration

The following are some useful features to provide in DELI if it is to target dynamic parallelization and reconfiguration for a CMP.

1. Binding a thread on a processor.
2. Configuring memory as FIFO, cache, etc. For example, if an application is using memory as a queue, DELI can dynamically compile load and store into push and pop.

3. Like binding threads to processors, objects can be bound to specific (non-uniform access) memories.
4. Speculation, e.g., like in Hydra.
5. Eliminating unnecessary synchronization.

DELI provides a mechanism to allow application participation while making reconfigurations for a polymorphic CMP.

Additional Topics Discussed

Desired Features in a Dynamic Compiler

For complex programs that are sufficiently long running, dynamic compilation could provide significant benefits. We want to exploit static compilation as much as possible in a system supporting dynamic compilation. For example, the static compiler can tell the dynamic compiler which areas of the code to focus on at run-time. User annotations are helpful along these lines, but using user annotations as a hint that can be ignored if necessary is probably the right approach. Using profiles collected at run-time will be useful, especially to monitor the cost/benefit ratio of dynamic compilation and bail out if costs are exceeding the benefits.

Virtualizing the Hardware

Designing a hardware abstraction layer is complicated by the tremendous amount of variation in system hardware. Also, designing the hardware abstraction layer is often complicated by the need to exploit specific hardware features for efficiency. Particularly hard for virtualizing are:

1. Interconnects
2. Memory configuration and cache coherence.

One idea that was proposed was a query interface that applications can use to figure out which features are supported by the hardware and which features are emulated, maybe even the relative costs of these operations. Such an interface will give applications the ability to make the right tradeoffs.

Dynamic Compilers and the Operating System

Several questions were raised about the interaction between dynamic compilers and the OS. Some of the features that we are thinking about in dynamic compilers such as run-time resource allocation and allocating threads to processors are traditionally under the control of the OS. Dynamic compilers like DELI can run above or below the OS. Running below the OS gives the dynamic additional flexibility but at the cost of needing to reimplement many of the services provided by the OS , e.g., virtual memory.

Conclusions

The following points summarize the discussion:

1. Dynamic compilation has the potential to improve the performance long-running programs. Dynamic compilation is an efficient vessel for binary compatibility and run-time code manipulation and observation.
2. Static compilation must be exploited as much as possible in a dynamic compiler. Combining the DyC and DELI approaches seems promising.

Open Problems

1. The most promising open problem in dynamic compilation seems to be in mechanisms to combine static and run-time information for efficient dynamic optimization. The information obtained by the static compiler by program analysis can be used to reduce the space of optimization techniques considered at run-time.
2. Problems in efficient run-time profiling and summarizing profiles in a manner useful for a dynamic compiler are important.
3. The design of the DELI API and HAM layers for a CMP need to be worked out to bring the benefits of DELI to CMPs.