

DLLs, Dynamic Optimizations, and Performance: How It All Fits Together

Francesco Spadini Greg Muthler

ECE 411 Final Report
Professor Hwu

April 22, 2002

1 Introduction

In the early days of programming, software was small and highly tuned to the task at hand. As computers became more powerful and commoditized, the programs they ran grew in complexity and size. With this code explosion, software engineering and code maintenance practices took on a prominent role. One of the coding techniques created to counter-balance the growing complexity of programs was dynamic linking.

Dynamic linking and loading are common in modern code. Libraries are often written in this way to promote code reuse and to simplify maintenance [11]. This is particularly prevalent in Microsoft applications, where much of the code is divided among distinct dynamically linked library (DLL) files. In addition, languages such as Java dynamically link all inter-class method calls [10] in order to maximize flexibility and minimize network traffic.

While DLLs are extremely helpful in product development and deployment, they can be detrimental to performance. Since a DLL is not seen at compile time, it presents a barrier to optimization for traditional compilers. For this reason, code which straddles this barrier tends to be less efficient as traditional compiler optimizations cannot be performed. The overhead of the calling procedures remains formidable and the subtle effects on hardware components are, for the most part, unknown. Despite these obstacles and the growing reliance on DLLs, processors must maintain performance.

2 General Solutions

There are various methods of dealing with performance degradation across DLL boundaries. These methods can be either static or dynamic, and either software or hardware based.

There is at least one static method for dealing with DLLs. By using profile-guided optimizations on the DLL itself, performance may be improved. In the general case, the target user of a DLL is unknown. However, one may speculate that the uses of the DLL will be similar in most cases. By pre-optimizing the DLL, some amount of performance enhancement may be obtained. Obviously, the pre-optimizations may be useless and perhaps even detrimental if the use of the DLL is changed [2].

Dynamic methods are more promising and can yield much better performance than static methods. By their nature, dynamic systems take advantage of run-time behavior to improve performance.

While a static compiler is not capable of optimizing routines crossing DLL boundaries, it can introduce additional code which can dynamically profile and reoptimize the binary as it executes. This allows the program to adapt to changes and new software that it encounters. However, reoptimizing the binary takes processing power away from the main thread of execution. Also, such special optimization directives may have to be hand placed by the user, which requires additional programming effort and may not be able to take advantage of all the opportunities possible. One such reoptimizing compiler is DyC [6]. Other similar methods that profile and reoptimize code, but do not require user annotation are Dynamo [1] and DAISY [4].

Due to the fact that the Java specification requires dynamically linking all inter-class method calls, there have been numerous techniques proposed to allow optimizations to take place, despite this linking barrier. One such method is to use just-in-time (JIT) compilers which perform all compilation at run-time. The presence of this additional information allows them to perform barrier-crossing optimizations like virtual method call resolution and method inlining [8]. These issues have been addressed by the JUDO [3] compiler.

Some have sought to attack the issues at the hardware level. The uncertainty of dynamic method call targets can be addressed with additional hardware, as done in [14]. Additionally, some have taken advantage of the low-level abstraction of hardware, which is not hindered by the abstract notion of the dynamic method call. The hardware is thus capable of performing dynamic optimization on the retired instruction stream. Two examples of this method are the rePLay Framework [5] and ROAR [12].

3 Elements of Dynamic Optimization

When evaluating any dynamic method of optimization, there are several issues that must be addressed. Here we will further define five of those issues.

3.1 Precise Exceptions and Interrupts

Both exceptions and interrupts require the application to be in a consistent architectural state so that the operating system can be invoked. A dynamic optimization system is said to maintain precise exceptions and interrupts as long as an exception or interrupt can be taken cleanly at any point in the program. In order to handle this, the system may have to revert to a previously consistent state.

3.2 Deeper analysis

In some fashion, a dynamic system must gather information about the program at run-time. By accruing some amount of run-time information, dynamic optimizations can be performed. It is important to note the method by which a dynamic system gathers this information and performs a deeper analysis than that available statically. A deeper analysis can be transparent to the main program, or it can compete for resources with the program. If competing for resources, the gain from any optimizations must outway the loss of performance due to the competition.

3.3 Memory Consistency models

Memory consistency models define a set of rules concerning the ordering of loads and stores. While programmers tend to think in terms of sequential consistency, in which all loads and stores execute in order, this model places heavy restrictions for optimizers wanting to perform aggressive code reordering. Weaker memory models such as release consistency allow very aggressive reordering, at the cost increased programming difficulty. In order for a dynamic optimization system to be memory consistency model compliant, it cannot violate any of the ordering rules.

3.4 Verification

One serious problem with doing dynamic optimizations is verifying that the optimization does not break the code. There are certain optimizations which are considered unsafe, in that they can cause incorrect execution. If a dynamic system, or a static system for that matter, utilizes unsafe optimizations, it is important to either have an architectural recovery mechanism, or allow the user to specify whether to run the unsafe optimizations.

3.5 Self-Modifying Code (SMC) and Self-Referential Code

Self-modifying code (SMC) affects the performance of almost all systems. While it is a novel software trick to re-use code and perhaps improve performance, SMC is a challenge to support. Further, any system performing dynamic optimizations that also supports SMC must find some way to undo optimizations so that the original code can be modified. This is not an easy task, which is why very few dynamic optimization systems allow SMC.

Self-referential code reads the bytes of the instruction stream for use as data. A common application of self-referential code is a program that takes its own checksum. For a dynamic optimization system to support self-referential code, it must be able to produce the same result as the unoptimized version every time a self-referencing function is invoked.

4 Software

Programs have been shown to exhibit predictable run-time behavior that is not visible with static compile-time analysis. Many compilers make optimizations based on profile information in order to reap benefit from information that is hard to accurately deduce statically, such as likely branch directions. Recently though, a number of techniques have arisen that even allow software to dynamically adapt to changing conditions through run-time optimization. This also allows them to optimize across traditional boundaries, such as dynamically linked library calls.

These techniques can be grouped into two major categories. The first is user driven dynamic optimization. DyC [6], which uses code annotations of near-constant variables to define regions for dynamic opti-

mization, is an example of this category. The second category is transparent optimization. Dynamo [1] and DAISY [4] both perform transparent code transformations at run-time, without any compiler information.

The following sections describe these three systems, DyC, Dynamo, and DAISY in further detail.

4.1 DyC

DyC is an aggressive dynamic optimization system that employs a static compiler to generate specialized optimizations. It can perform polyvariant specialization and division, in addition to a series of traditional compiler optimizations including loop unrolling. It uses programmer annotations of variables that have few values at run-time, or are never changed once set, to drive a run-time dynamic optimizer. Such variables are referred to as static variables. This allows for information about near-constant input values to be passed across dynamically linked library calls [7].

4.1.1 Overview

The DyC compiler first determines the regions of code that could be further optimized with the knowledge of the values of static variables. Much of the analysis for each region can be done statically, so as to reduce the amount of work that needs to be done at run-time. This static analysis includes the identification of candidate branches and switch statements that can be replaced with one of the possible paths at run-time, loops that can be fully unrolled at run-time, and operations that can be reduced in strength depending on the value of the run-time constant.

This information is subsequently fed to a dynamic compiler generator. The generator is responsible for taking the static analysis results and constructing a specialized optimization routine (or generating extension) to create an optimized version of the region dynamically, given the initial value of its static variables. Given the amount of analysis that can be performed statically, the generating extension does not need to use an intermediate representation of the region.

DyC also inserts code in the program binary to manage a software cache of dynamically optimized code. By doing this, the cost of the dynamic optimizations can be amortized over the number of times the region is executed. Each time a dynamic region is entered at run-time, the code cache is consulted. If that region has already been optimized for that set of input values, the cached code is executed, and the only run-time

cost is the lookup itself. If, on the other hand, the region has not been executed previously with the present input value, then the customized optimizations for that region are run, and the resulting code is placed in the cache for future use.

4.1.2 Advantages and Disadvantages

DyC has a number of advantages over other dynamic optimization systems. It is one of the only systems in which the user can direct the optimizations that are done dynamically. This allows a skilled programmer to finely tune the optimizations to the configuration that yields optimal performance. In addition, the ability to declare run-time constants actually permits the user to instruct the compiler that it only has to optimize for one possible value. This may not have been possible to discern from static analysis, and the compiler would have otherwise had to be conservative.

There are a number of downsides to this approach to dynamic optimization though. The first is that DyC requires a significant effort on the part of both the compiler writer and the programmer. Static variables may be hard for even the programmer to discern in large projects. Also, it has proven very hard in practice to stage compiler optimizations. Automatic staged optimization generation is an area of current research [13].

Second, the use of DyC in a software project makes code reuse much more difficult. By reusing a module in a different way, annotated variables may cease to have very consistent values. This could lead to considerable performance degradation due to invoking the optimizer for uncommon input values. Worse yet, it is possible to get incorrect results if the run-time constant optimization is employed. If a module with run-time constants is reused in a way such that the annotated variable is modified in the course of the program, the optimized code will not see the new value. This will result in incorrect execution unless the programmers take on the burden of updating the annotations whenever a new path into the region is added.

4.1.3 Correctness Issues

DyC requires no special support for interrupts and exceptions. Even if an interrupt or exception occurs inside an optimized region, it is still a consistent and recoverable state.

DyC's specialization can actually produce incorrect results for self-referential code. If a function checksums a region of code that is dynamically optimized, the checksum may be different from the original code.

In fact, depending on the values of the inputs to the region, the checksum may have a different value for different invocations of the same region.

Also, DyC has no apparent support for self-modifying code. A store to code that has already been dynamically optimized and placed in the code cache will not have any effect. Lastly, whether DyC violates memory consistency or not depends on the aggressiveness of the dynamic optimizations. As with any optimizing compiler, it can be written to follow the rules enforced by the memory consistency model.

4.2 Dynamo

Dynamo takes a different approach to dynamic optimization than DyC. Rather than relying on extensive programmer and compiler support to identify static variables, Dynamo runs transparently without any high-level language knowledge [1].

Acting as a middleware layer between the target application and the underlying hardware, it can perform profiling and dynamic optimization on the target. Dynamo is capable of performing safe compiler optimizations, including redundant branch elimination, and potentially unsafe optimizations including redundant load and store removal, and loop transformations. Given these are applied at run-time, after dynamic library references have already been resolved, Dynamo is capable of optimizing across dynamically linked library calls without special support.

4.2.1 Overview

Dynamo begins by loading itself into the address space of the target application, and interpreting the target's instructions. While doing this, it dynamically locates hot paths in the target code, which the authors call traces. A trace is then turned into a fragment by modifying the branch directions so that the fall-through path always remains in the trace, and the taken path always exits the trace. A linker stub is then added to each exit path, so that branching out of the trace will return control to Dynamo.

The fragment is then optimized for the newly constructed not-taken path using a run-time optimizer. This code can be fast and simple, given that fragments are guaranteed to have only a single entrance. The optimized fragment is then placed into a fragment cache for future use.

As Dynamo interprets the target application, it looks up the present PC in the fragment cache. If there

is a hit, it jumps to the optimized fragment, which runs directly on the native hardware. When the program exits the fragment, it automatically relinquishes control to Dynamo due to the linker stubs that were inserted during fragment construction.

4.2.2 Advantages and Disadvantages

The main advantage of Dynamo is that its run-time infrastructure allows the optimizer to take advantage of boundaries to static optimization. Since fragments can span dynamically linked library calls, Dynamo can remove inefficiencies that are not visible to the static compiler. Another advantage is that Dynamo is completely user, compiler, and operating system transparent.

The disadvantages of Dynamo are threefold. First, in order to achieve maximum benefit, aggressive, and potentially unsafe optimizations must be turned on. Second, when either too many fragments are created, or certain unsupported system calls are invoked, Dynamo must relinquish all control to the original binary. Once this has happened, Dynamo cannot force the binary to execute code from the fragment cache. Lastly, Dynamo is not capable of optimizing across kernel call boundaries.

4.2.3 Correctness Issues

Like DyC, Dynamo has the problem that the PC of the original code is not kept intact. This means that any code that relies on specific PC values, may not output the same value each time through the code segment.

Dynamo also requires special handling for system calls and interrupts. When the target application performs a system call, it is proxied by Dynamo. This task is simplified by the fact that Dynamo is in the same address space as the target. Interrupts pose an additional problem. Since interrupt handlers are called on externally triggered events, they will not run within the context of Dynamo. This has the potential to make Dynamo's version of the target application's state inconsistent. This problem is solved by installing a special handler which first traps control to Dynamo, and then calls the original handler.

As in DyC, the safety of the Dynamo optimizations with respect to memory consistency depends on the specifics of implementation. Supporting any consistency model should be a matter of putting restrictions on optimizations that remove or reorder loads and stores.

4.3 DAISY

DAISY is a binary translation tool that guarantees architectural compatibility across instruction sets. While DAISY does not specifically perform dynamic optimization, it does have a similar set of correctness issues to other software optimizers [4].

4.3.1 Overview

DAISY makes it possible to transparently run code for a base architecture on another, migrant architecture. It is essential that the migrant architecture is a superset of the base.

DAISY is capable of performing code scheduling and optimization as well. Any speculative or hoisted operation is made to write to renamed registers that do not exist in the base architecture. When the result becomes non-speculative, it is written into an architected register.

In order to facilitate reading from base architecture physical addresses in the migrant architecture, all of the base architecture's physical memory is identity mapped in that of the migrant. This assures that self-referential code can still function, as all of the base architecture's physical memory is unchanged.

The generated code is placed in a reserved region of virtual memory. The code for each base architecture physical address is placed at the migrant architecture virtual address equal to the start of the reserved region plus the base's physical address times a scaling factor. This scaling factor represents the projected number of migrant instructions per base instruction. Thus, DAISY can easily find the correct migrant virtual address at run-time given a base physical address.

4.3.2 Correctness Issues

The fact that DAISY writes speculative operations to renamed registers allows it to correctly handle interrupts. The values in the architected registers will be the program point in the original base architecture at which to take an interrupt.

Another problem which arises is that renamed registers are not saved on a context switch or an interrupt, since the context switch code is also from the base architecture and translated through DAISY. This can be handled by regenerating valid migrant architecture code starting from the valid base architecture point at

which the interrupt was taken.

The question of when exceptions are raised on speculative loads is also of concern. If a load writes to an architected register, then it must be non-speculative. In this case, it is safe to raise the exception immediately. In the case that it writes to a renamed register, it simply sets its poison bit. When the instruction which moves that speculative result into an architected register executes, the exception is raised. Since DAISY maintains correct base architectural state at all times, the migrant architecture will have the same state as the base architecture would have had at that point in the program. This assures that all exceptions are handled in precisely the same fashion in the migrant architecture.

Memory consistency is maintained in a similar fashion. On an memory conflict misspeculation, or on a write from another processor in a multiprocessor environment, DAISY restarts code generation, starting with the offending load.

Self-modifying code is handled by adding a read-only bit to the base architecture physical memory. In order to maintain complete compatibility, this bit is not visible to the base architecture. Whenever a piece of code gets translated, its respective read-only bit is set. When a segment of memory with a set read-only bit is written, DAISY is triggered to regenerate the modified code.

5 Hardware

At the microarchitectural level there are many opportunities for dynamic optimization that are not present at the software level. One advantage of most hardware schemes is that they require no user annotation of code or software support. There is also less of an overhead involved in doing optimizations as tricky correctness issues can be often be circumvented with additional hardware. Lastly, DLLs and other boundaries can often appear transparent to the hardware.

Here we will analyze three hardware level schemes that deal with DLLs and other dynamic optimization issues. The first scheme simply attempts to maintain BTB performance. The other two schemes are more aggressive, performing dynamic optimization work at the hardware level.

5.1 BTB Performance

One aspect of system hardware that has been studied in relationship to the presence of DLLs is the branch target buffer (BTB) [14]. Performance of the BTB suffers due to contention when DLLs are introduced into the instruction mix. Normally, a program is constructed so as to reduce BTB contention and provide high performance. A DLL cannot be included in the contention analysis, nor can it be constructed to eliminate contention since the code is used by many different programs and contention removal for all possible processes is infeasible.

To maintain the performance of the BTB, [14] suggest the use of a small secondary buffer called the DLL BTB, similar in structure to a victim BTB. The DLL BTB holds the BTB entries for all DLL branch targets. By restricting the DLL BTB entries to this secondary structure, the contention issue related to DLLs can be eliminated. The filtering of DLL branches can be accomplished by using an import table associated with the process. While significant improvements can be made in BTB hit rates, this design addresses only a small portion of the issues involved with DLLs.

5.2 rePLay

The rePLay framework for dynamic optimization [5] utilizes the concept of single-entry, single-exit atomic code regions called frames. Each frame is composed of a highly regular control flow path from the retired instruction stream. Branches in frames are converted into asserts, which create no control flow dependencies. Instead, the assertion checks that the predicted direction of the branch is correct.

A frame is more aptly defined as a collection of basic blocks in which all internal branches have been converted into asserts. The only branch allowed in a frame is the terminating branch at the end of the frame. Since this removes all side exits, a frame becomes an atomic entity. If one instruction executes, all instructions are guaranteed to execute. If the assert condition dictates that the program take a side exit of the frame, then the entire frame is discarded. This event is also called an assertion firing.

After a frame is constructed it is sent to a hardware dynamic optimization engine. The optimization engine performs a large set of optimizations on the frame including constant propagation, dead code removal, common sub-expression removal, and a host of other traditional compiler optimizations. An optimized

frame is subsequently stored in a frame cache. Each cycle, the processor indexes into the frame cache with the current PC and global branch history. In the event of a miss, a machine width is fetched from the traditional instruction cache.

5.2.1 Advantages and Disadvantages

RePLay has two significant advantages. First, no user, software, or compiler support is required. Since rePLay is at a lower level of abstraction, DLL boundaries are transparent, and can be included in frames.

Second, the average frame encompasses many basic blocks. This makes the optimizer more global in scope than most hardware optimization schemes. Also, the fact that there are no loops or branches of any kind in a frame make implementing optimizations in hardware much simpler.

On the other hand, rePLay has a number of disadvantages as well. The most obvious is the significant additional hardware investment required for the optimization engine. Another drawback is the limited view of the program. Static compilers have a far more global scope of the program than the 256 instructions to which rePLay is limited. Also, since rePLay functions at the hardware level, it cannot perform any optimizations requiring source-level analysis.

5.2.2 Correctness Issues

To allow for precise interrupts and exceptions the rePLay framework must assert a frame on the reception of interrupts or exceptions. This must be done because for the duration of a frame, the state of the processor does not reflect precise architectural state. This is due to the fact that the optimizer will have rescheduled and changed instructions, guaranteeing correct state only at the end of the frame.

Self-referential code can also be handled in rePLay. Since rePLay does not modify the actual instruction stream, any references to the original instructions will remain unchanged. Since the optimized routine is only visible microarchitecturally, there is no way that self-checking code could access the optimized version. Although the rePLay framework does not currently support SMC, this could be easily done in the same fashion as the P4 trace cache. The P4 uses the TLB to track which memory addresses are in the cache. If any address in the cache is written, the trace cache is flushed [9].

RePLay can be customized to whichever memory model desired. However, this decision currently rests

on the chip designers. The rePLay optimizer could also be constructed using programmable logic that could be changed later to support different memory models. Currently, the results for rePLay are gathered assuming a weak memory ordering model. All stores must take place as they may be volatile or memory-mapped, but loads may be removed by store to load forwarding.

RePLay has limited verification capabilities. The built in hardware recovery mechanism allows recovery from some potential faults including control flow asserts, register value assumptions, and address predictions. These faults are all limited to events that change the stable past behavior seen in creating the frame. Although this process may catch potential problems, it does not validate the legality of the optimizations in all cases.

5.3 ROAR

The ROAR processor [12] uses a similar method to rePLay in performing dynamic optimizations. First, a hot spot detection mechanism records branch behaviors from the retired instruction stream. Once a hot spot is detected, the trace generation unit collects the relevant code and performs optimizations. The trace itself is stored in a reserved page in memory and is accessed by modifying the BTB entry for the original branch so that it now points to the trace set instead of the instruction target. This relocation of a hot spot into a contiguous region of memory gives some of the benefit of a trace cache since code that is nearby in dynamic program order will be nearby in address.

5.3.1 Advantages and Disadvantages

A major difference between a frame utilized in rePLay and a trace constructed by ROAR is the presence of side exits within the trace. While this can potentially limit some future optimizations it eliminates the need for hardware supported recovery.

A drawback of the ROAR architecture is that it requires some amount of operating system support for the code pages in virtual memory. On the other hand, it has the advantage of being the most general scheme analyzed. It works with minimal outside support, and it removes the restriction of rePLay that there must be very common control flow paths in order to construct code regions for optimization.

5.3.2 Correctness Issues

Self-referential code could potentially be a problem with ROAR since branch targets are modified. This could cause a code checksum to be different for two different invocations of a function.

Precise exceptions are not currently modeled in the ROAR simulator. However, since the trace sets do not change the original code except for branch targets, precise exceptions and interrupts should not be an issue.

In a similar vein, ROAR does not need expensive verification, since traces are constructed to be both consistent and well-formed. Without performing further optimizations on the code beyond basic trace construction, branch replication, and branch promotion, verification is not necessary. There are conceptually no differences between the trace sets and the original code.

Any memory model used by the original code will be maintained.

6 Java

The Java language imposes many unique challenges to achieving high performance. Language restrictions meant to maximize the security, robustness, and network mobility of Java code often impose barriers to static optimization that are not present in most other languages.

There are two restrictions that are of primary interest in this paper. The first is that of not allowing any static method inlining. Since every class in Java has certain security permissions, inlining statically could potentially violate run-time determined permissions. The second is the rule that all classes are to be linked and loaded on demand at run-time. This minimizes the initialization latency of Java programs whose class files may be on another node on the network.

6.1 Just-In-Time Compilers

Early versions of Java executed bytecode in an interpreter. More recently, JITs have emerged as a way to maintain portability while approaching the performance of statically compiled binaries. At run-time, JITs translate Java bytecode into native instructions [8].

With run-time information, JITs are also capable of overcoming some of the restrictions on static compilers. Once such JIT, JUDO [3], performs speculative virtual function inlining in order to expose some of the cross-class inefficiencies to its run-time optimizer. While other cross-class optimizations would be possible, JUDO must be very fast so as to avoid imposing a significant run-time overhead.

6.1.1 Overview

JUDO [3] is an aggressive JIT that performs speculative inlining, exception optimizations including lazy exceptions, and a host of traditional Java optimizations such as array bounds checking deletion, and safe casting removal. Only inlining will be discussed in depth, as it is the only optimization that specifically relates to optimizing across static boundaries at run-time.

Virtual function inlining is performed by converting the virtual function lookup code into a jump to the inlined function, followed by an appropriate number of nops to fill the difference in code size. Unfortunately, this is potentially unsafe in Java, as the class hierarchy may change dynamically. In order to ensure correctness of the resulting code, JUDO generates an inline patch, and notifies the underlying JVM that a speculative inline has been performed between the caller and callee. Once inlining has been performed, all of the Java-specific and traditional compiler optimizations can be applied to the callee.

The patch consists of a pointer to the callee, a pointer to the modified code, and a byte array of the original code. In the event that the callee code is overridden, it will notify JUDO that the caller-callee relationship is no longer accurate with an inline override. It is able to do this easily because it is notified of every inline. Upon reception of the inline override, JUDO applies the patch, thus returning the code to the safe virtual function table lookup.

6.1.2 Advantages and Disadvantages

Java execution with a JIT achieves considerably higher performance than with an interpreter. It has the additional advantage of being transparent to the user and the programmer.

JIT execution has two main disadvantages. First, most high performance JITs, such as JUDO, require specialized support from the JVM. While this is done to optimize performance, it also makes them less portable.

Second, most JIT optimizations address easy to fix Java language-specific bottlenecks such as array bounds checking and null pointer checking, rather than confronting the main barriers to static optimization. This is due to the fact that the entire program cannot be analyzed at once, and the original source is not available. These factors combine to make higher-level global or whole program analysis nearly impossible.

6.1.3 Correctness Issues

Many of the correctness issues present in other languages are not applicable to Java without the use of native methods. For example, there is no way to determine a function's PC in Java, and there is no inline assembly interface. Therefore, the self-referential code that could potentially be a problem for some of the general software optimizers cannot occur in Java. The even greater problem of self-modifying code is not permitted either, as there is no way to determine where in memory the JIT will place a given instruction, or what machine code the JIT will generate.

Java has its own memory consistency and exception model independent of the underlying hardware. Given that these regulations are often decided by the JVM and JIT itself, it may have fewer restrictions than conventional dynamic optimizers.

7 Conclusion

The code size of modern programs necessitates the use of dynamic linking and loading. While this allows for efficient code reuse and simplified maintenance, it is often hard to maintain the performance of statically linked modules. Several methods have been proposed to deal with this growing issue.

The simplest method relies on static compiler profiling to exploit the fact that DLLs are used in a similar fashion between processes. This method, however, does not adapt readily to changes in DLL usage. Also, this does not help optimization of the DLL caller.

There are several dynamic software based methods that allow code to be re-optimized based on run-time behavior. Software methods, while easy to deploy on existing systems, have to take processor time away from the target application. This problem is addressed in two ways. First, only code that is seen many times is optimized. Second, many systems keep a cache of already optimized code. This effectively amortizes the

cost of the optimizations over all of the times the code is executed.

There also exist several dynamic hardware based methods for dynamic optimization. Hardware mechanisms have the advantage of allowing the optimizations to appear transparent to the operating system and users, as well as operating without overhead as they run in parallel to the target application. Since they are built into the hardware itself, these systems can take advantage of knowledge about their specific microarchitecture. Hardware systems that perform dynamic optimization are presently purely academic as currently no hardware exists that does aggressive optimizations.

The effectiveness of all dynamic schemes rests in how efficiently they can make optimizations and then use them effectively. If a dynamic scheme takes too long to adapt, then the process behavior that it is trying to capture may no longer be pertinent. Additionally, if a method requires too much overhead, then any possible gains from the optimization will be lost.

The increasing use of DLLs will decrease the effectiveness of static compilers. This phenomenon requires some form of dynamically adaptive systems.

References

- [1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
- [2] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The predictability of branches in libraries. In *WRL Research Report 95/6*, 1995.
- [3] Michael Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation*, pages 13–26, 2000.
- [4] Kemal Ebcioglu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

- [5] Brian Fahs, Satarupa Bose, Matthew Crum, Brian Slechta, Francesco Spadini, Tony Tung, Sanjay J. Patel, and Steven S. Lumetta. Performance characterization of a hardware mechanism for dynamic optimization. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 16–27, 2001.
- [6] Brian Grant, Marcus Mock, Matthai Phillipose, Craig Chambers, and Susan J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report UW-CSE-97-03-03, University of Washington, May 1999.
- [7] Brian Grant, Matthai Phillipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, pages 293–304, 1999.
- [8] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Kamatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, 1999.
- [9] Stephan Jourdan. The p4 architecture, 2002. Talk given at the University of Illinois at Urbana-Champaign, March 7, 2002.
- [10] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Computing Surveys*, 32(3):213–240, 2000.
- [11] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Baer, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 27–38, 1998.
- [12] Matthew C. Merten, Andrew R. Trick, Ronald D. Barnes, Erik M. Nystrom, Christopher N. George, John C. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–588, 2001.

- [13] Matthai Philipose, Craig Chambers, and Susan J. Eggers. Towards automatic construction of staged compilers. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 113–125, 2002.
- [14] Stevan Vlaovic, Edward S. Davidson, and Gary S. Tyson. Improving BTB performance in the presence of DLLs. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 77–86, 2000.