Mojo: A Dynamic Optimization System

Wen-Ke Chen* Sorin Lerner* Ronnie Chaiken David M. Gillies
Microsoft Research
One Microsoft Way, Redmond WA, 98052
chwk@cs.arizona.edu, lerns@cs.washington.edu,
{rchaiken, dgillies}@microsoft.com

Abstract

Dynamic optimization systems have the flexibility to adapt program execution to changing scenarios and differing hardware configurations. Previous work has shown that this flexibility can be exploited for sizeable performance improvement. Yet, the work to date has been chiefly targeted towards running the SPEC benchmarks on scientific workstations. We contend that this technology is also important to the desktop computing environment where running large, complex commercial software applications is commonplace.

In this paper, we describe work that has been accomplished over the past several months at Microsoft Research to design and develop a dynamic software optimization system called Mojo. In particular, we present implementation details for the x86 architecture -- Mojo's initial target. Additionally, we present our experience in supporting exception handling and multi-threaded applications on Windows 2000 along with preliminary performance measurements.

I. Introduction

Due to the limitations of static analysis in predicting program's varying run-time behavior, even in the presence of profile information, static optimization still has the difficult task of mapping a static solution onto the fundamentally dynamic space of program execution. Consequently, many optimization opportunities are left unexplored in compiler generated binary code. Several approaches have been studied to exploit such opportunities. These approaches usually defer some optimization decisions until relevant information about the program's run-time behavior becomes available, for example, at run-time.

Although modern microprocessors have been using hardware to do simple run-time optimizations for a long time, run-time software optimization systems have just begun emerging. Research on run-time software optimization has however been targeted mainly for single-threaded applications on RISC machines running variants of UNIX, and has not been aggressively explored for the most widely adopted chip in desktop computing: the CISC-style x86. This may be partially due to the complexity of the x86 Instruction Set Architecture (ISA), which complicates the analysis and optimization of binary code at run time.

Also, many popular desktop applications use multiple threads of execution and the impact of threads on run-time software optimization systems has not been extensively investigated. Run-time optimization systems typically employ software caches to buffer optimized traces. Since threads may differ drastically in their behavior, determining what paths to share among threads and how to manage them becomes challenging.

Furthermore, exceptions have been gradually accepted as a structured way to separate unusual and infrequent code paths from the regular ones. Windows 2000 even exposes an API for Structured Exception Handling (SEH) that is supported directly by

* Wen-Ke Chen is a PhD student in Computer Science at the University of Arizona; Sorin Lerner is a PhD student in Computer Science at the University of Washington. This work was done while they were summer interns at Microsoft Research, 2000.

the operating system and is independent of language-level exceptions. For example, C++ exceptions on Windows 2000 are typically implemented as a layer on top of SEH. The irregular control flow caused by exceptions complicates the design of dynamic optimization frameworks. Indeed, regaining control of the application after an exception may require deep understanding of system internals

We feel that efficiently supporting exception handling and multi-threaded applications is the key to wide deployment of run-time software optimization systems. In this paper, we describe work that has been accomplished over the past several months at Microsoft Research to design and develop a dynamic software optimization system called Mojo. In particular, we present implementation details for the x86 architecture -- Mojo's initial target. Additionally, we present our experience in supporting exception handling and multi-threaded applications on Windows 2000 along with preliminary performance measurements.

The paper is divided into seven Sections. Section II describes previous work on run-time optimization systems. In Section III, the implementation details of Mojo are discussed, including details of exception handling and threads in the Windows 2000 environment. Section IV shows the preliminary performance of applications running under Mojo's control. Results are shown for several programs including SPEC benchmarks and the large Windows applications Word and Visual FoxPro. In Section V, we describe the future direction of this work and Section VI summarizes the paper.

II. Previous Work

Opportunities for machine-specific and customized optimization often remain even in the best compiler-generated binary code [12]. To overcome the limitations of static analysis and separate compilation, the traditional compile-execute model has been extended to allow analysis and optimization to occur not only at compile time but also at link time [4, 5], and run time [1, 2]. Approaches that postpone compilation and/or code generation until run time have also been explored [6, 7, 8, 9]. Among these techniques, run-time translation/optimization distinguishes itself by its transparency, versatility and performance-delivery potential.

For example, IBM's DAISY [10] and Digital's FX!32 [11] demonstrate that run-time translation can solve compatibility problems in migrating existing code to microprocessors implementing a different ISA. In addition, Transmeta's Crusoe architecture [2] shows the value run-time technology brings to hardware design for power management and compatibility.

The work closest to ours is the Dynamo system [1], which has pioneered many aspects of software-only run-time optimization for binary code. Dynamo's approach is to optimize dynamically identified *hot traces* of code while interpreting less frequently executed portions of a program. However, their work has mostly been targeted at PA-RISC code, not CISC-style architectures. Interpretation is a natural approach for RISC-style ISA's with many registers and a large system-state. However, for a CISC-style ISA like x86 with few registers and a small system-state, there is flexibility to avoid software interpretation altogether by direct use of the underlying hardware. Mojo's implementation for x86 makes use of the underlying hardware in this way.

In addition, [1] does not address the issues of supporting exception handling and multi-threaded applications. Since Mojo supports all these features, a wider range of software can benefit from dynamic optimization.

III. Design and Implementation of Mojo

Goals

In designing Mojo, we had the following goals in mind:

- Mojo should support large, multi-threaded desktop applications which make use of exception handling;
- Mojo should require no OS support, and should control program execution in a way that allows optimizations across shared library boundaries;
- The performance of Mojo should rival the performance of native code optimized with profile feedback;
- The architecture should be flexible enough so that it can easily incorporate information provided by a static compiler.

Overview

Figure 1 shows the structure of Mojo. We will first give a brief overview of all the components and how they interact. Detailed explanations are provided in later sections.

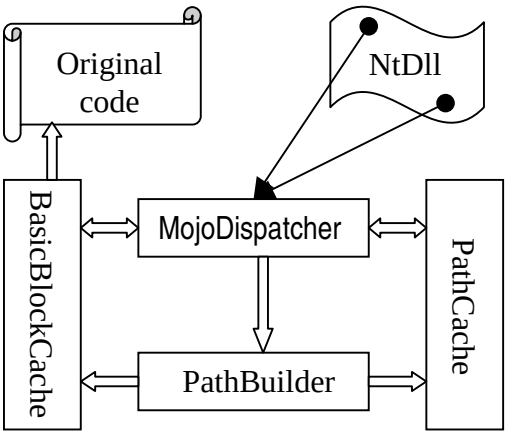

Figure 1: Mojo Structure

Mojo only executes fragments of code buffered in the Path Cache or the Basic Block Cache. These fragments, indexed by original program address, are copies of the original code with control transfer instructions modified so that control is returned to the Dispatcher after the code fragment has run. In normal execution mode, the Dispatcher is given an address in the original program so that it can start or resume Mojo execution at that address. To do this, the Dispatcher first consults the Path Cache to see whether there exists a path with the given instruction pointer as its entry point. If such a path exists, the code is directly executed from the Path Cache. Otherwise, the Dispatcher consults the Basic Block Cache to determine if there exists a *basic block* corresponding to the given instruction pointer. If such a block exists, it is executed directly from the Basic Block Cache. Otherwise the Basic Block Cache identifies and brings in the right basic block before executing it.

Each time control returns to the Dispatcher, the Path Builder is invoked to determine if some of the basic blocks are hot enough to warrant entry into the longer-persisting Path Cache. If some basic blocks are hot enough, Mojo switches to path-building mode, where a path consisting of multiple basic blocks is created. In path-building mode, each executed basic block is passed to the Path Builder so that it can be added to the growing path. Once a path-termination condition is met, the newly created path is optimized and placed in the Path Cache for later execution. If the Dispatcher finds that too much time is spent building hot paths, it will bail-out by transferring control back to the original code. When this happens, the application runs natively without Mojo intervention for the remainder of the execution.

In addition to getting control from the Path Cache or the Basic Block Cache, the Dispatcher can also get control from the dynamic link library NtDll. This occurs when the OS transfers control to the application asynchronously, for example in the case of exceptions. The details of how NtDll interacts with the Dispatcher are given in the section on exception handling.

Gaining Control

Although our goal is to let Mojo control the execution of only those portions of the original program that can benefit from run-time optimization, in our current implementation, Mojo operates by *running* applications entirely under its control. Mojo can gain initial control of an application in two ways. First, for binary images, Mojo

relies on the Vulcan toolkit [3] for x86. Vulcan provides an API that allows a wide-range of binary-level tools to be rapidly written. We have written a tool called *MojoInit* that modifies an executable image to overwrite the normal application entry point with a call to `MojoStart` in Mojo.dll. After the call to `MojoStart`, the program will run entirely under Mojo's control beginning at the original entry point. Mojo also exposes an API to the programmer that includes the routines `MojoStart`, `MojoStop` and `MojoContinue`. With this API, Mojo execution can be programmatically started, stopped and continued. In this paper, all programs were Mojo-enabled by use of the Vulcan-based tool *MojoInit*.

MojoDispatcher

MojoDispatcher is the main controller of Mojo. It monitors both Mojo's and the original program's activities and decides what action to take during execution. Its major task is to maintain Mojo-specific state and orchestrate the interaction among Mojo components and the original program. Since control is switched between the original program and Mojo through the Dispatcher, it is natural to let the Dispatcher be responsible for saving and restoring the execution context. The context includes integer registers, the flag register, the instruction pointer, and the program stack. Mojo uses its own small, statically allocated stack in order to leave the original program's execution context unchanged. The x86 floating-point registers need not be saved because Mojo's code does not use floating-point instructions.

The Basic Block Cache

Mojo needs to simulate the effects of running basic blocks that are not part of the Path Cache, in other words basic blocks that are not currently hot. There are two ways to achieve this: direct execution, where the underlying hardware is used to run the instructions, or interpretation, where the instructions are emulated. On RISC architectures, the interpreter approach is more efficient because it relieves the dispatcher from saving and restoring registers at each basic block boundary, which is an expensive operation on architectures with many registers. However, for a CISC-style architecture such as the x86, with many instructions and few registers, a direct execution approach seems better suited. Because there are few registers, saving and restoring the register state at each basic block does not incur such a high cost. In addition, the difficult task of designing an interpreter for the whole x86 architecture is avoided.

Our implementation, which uses the direct execution approach, disassembles one basic block at a time using a lightweight disassembler whose only task is to find and decode control transfer instructions such as branches, loops, calls and returns. The basic blocks recognized in this manner are dynamic, and do not correspond to the basic blocks identified by a traditional compiler. They do however have the same property of having one entry at the beginning and one control-transfer instruction at the end. Once a basic block is recognized, it is copied into a buffer, with its control transfer instruction modified so that it passes control to the Dispatcher with the address of the next instruction to execute. In order to amortize the overhead of doing this operation (disassembly and control transfer instruction modification), recently executed basic blocks are cached in the Basic Block Cache.

The Basic Block Cache also keeps additional information about each basic block beside its code, for example what type of control transfer instruction ends the block. This information is used by the Path Builder to select and build hot paths.

The Path Builder

The Path Builder is responsible for selecting, building and optimizing hot paths. Each time control returns to the Dispatcher, the Path Builder collects statistics needed to identify and construct hot paths.

Path Entry Selection

It is well known that a program's execution time is dominated by some repetitively executed sequences of code. Mojo uses a speculative approach similar to the one in Dynamo [1] for identifying the beginnings of such sequences (which we call *hot paths*): If control returns to the Dispatcher from the Basic Block Cache through a backward branch or from the Path Cache, the Path Builder increments a counter associated with the target basic block; If the count exceeds some threshold, Mojo starts to build a new path at the target basic block, augmenting the path with the basic blocks that are executed following the entry block.

Mojo uses separate thresholds for back edge targets (32) and path exit targets (10). Because paths are built by following the current execution flow, it is possible, although rare, that when the Path Builder starts collecting basic blocks for a path, it collects a cold path, rather than a hot one. For example, only the first block in the path might be hot, whereas all other blocks never get executed again after the path is collected. In this case, the exit of the first basic block would soon become hot, and a new path would be built starting at this exit. The new path would be linked with the exit of the first path, and this would fix the original erroneous path selection. By having a reduced threshold for path exits, we recover more quickly from such poorly selected paths.

Path Construction and Optimization

As described above, paths are constructed speculatively by incorporating one basic block at a time along the execution flow, until some path-terminating condition is satisfied. The blocks in the path are laid out contiguously regardless of the layout in the original program. In doing so, conditional branches may be reversed, unconditional jumps dropped, and call/return sequences inlined.

The constructed paths are essentially superblocks (with a single entry and multiple exits), which are amenable to further optimization [13]. Our initial implementation focused primarily on code layout to improve instruction cache locality and branch prediction. We do, however, perform some branch-related optimizations, including unrolling loops and recursive calls as part of the path construction.

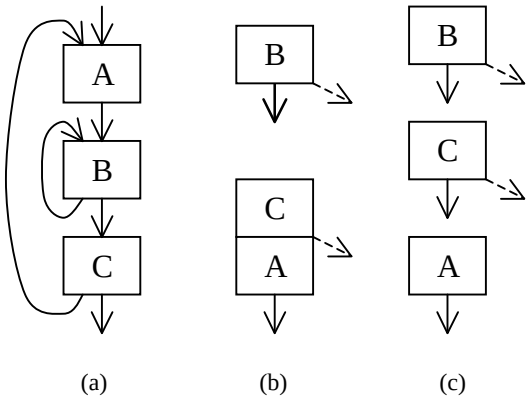

Figure 2: (a) original nested loops; (b) ideal paths built for the loops; (c) paths built with each path terminated at a back edge. In (b) and (c), solid arrows stand for fall-thru exits; dashed arrows for side exits.

Path Termination

Path termination greatly impacts both the quality and quantity of hot paths, which in turn affects the overall performance of Mojo. Terminating paths too early implies fewer optimization opportunities within each path and results in a large number of paths, which adds to the cache management overhead. On the other hand, if a path is terminated too late, there may be too many duplicates of some common blocks, which again leads to inefficient use of the Path Cache. For example, in the nested loop of Figure 2(a), it would be ideal to build two hot paths for the loops, one containing the basic block B, and the other containing the sequence (CA), as shown in Figure 2(b). If hot paths are always terminated at back edges, as proposed in Dynamo, three paths will be built: (B), (C) and (A), as in Figure 2(c). This example shows how terminating

the path that starts at C too early, as in 2(c), can miss out on creating the longer and better path (CA).

The Path Cache and Path Linking

The Path Cache maintains hot paths built by the Path Builder along with relevant path information. To reduce the overhead involved in switching execution context back and forth, paths in the Path Cache are linked together whenever possible. This includes linking the exits of newly formed paths to entries of already existing paths and linking the exits of existing paths to the entry of the newly built one. While the former can be easily accomplished at the time a new path is added into the Path Cache, the latter requires some extra programming labor and run-time cost, since all the exits of all the paths in the cache must be processed. However, a technique we call *deferred linking* can make the task as easy as the former at almost no extra labor and cost: such exits are only linked one by one when an exit is actually followed for the first time in later execution. Linking the exits of a new path to existing paths could also be deferred for the same reason, although we don't do this currently. However, the linking of the fall-through exit should not be deferred because the exit is very likely to be followed by construction. Deferred linking has several benefits: (1) Mojo uses run-time information to decide which exits should be linked so that unnecessary path linking is avoided. (2) No additional data structure needs to be maintained to speed the linking of existing paths to the newly built path. This simplification also makes it easy to unlink paths, which is inevitable in cache management; (3) Our simple and unified linking mechanism makes it efficient and easy to implement.

Memory and Cache Management

During initialization Mojo allocates a fixed-size region of memory for the Dispatcher (saved execution contexts and Mojo stack), for the Basic Block Cache, for the Path Builder, and for the Path Cache. Since both the Dispatcher's and the Path Builder's usage of memory does not change dynamically, memory management is reduced to management of the two caches. Furthermore, the two caches are used in similar ways, so the same algorithm can be applied to both of them. For this reason, the following description of Path Cache management applies equally well to the Basic Block Cache.

The Path Cache is divided into several sections (currently two for simplicity), organized and used as a circular buffer. Paths are not allowed to straddle sections. If the *current section* cannot store a newly created path in its entirety, the Path Cache will try the *next section* (the Path Builder is informed not to build paths longer than a section can hold). If the *next section* is not empty, all links between its paths and paths in other sections are broken and all its paths are discarded before any new path is added. The presence of multiple threads complicates cache flushing since another thread could be executing in the flushed buffer. The thread support section discusses how we address this issue.

Since cache management must be simple and cheap, flushing is a re-initialization of a pointer to the start of a buffer. This fast flushing policy can have a big impact on the number of paths that must be regenerated since some of the flushed paths may still be hot. For this reason alone we chose to split the Path Cache into two equal sized buffers so that a flush affects the oldest paths. This leads to the cache's natural and smooth adaptation to program's changing set of hot paths caused by phased behavior. However, in the event flushing rarely occurs, there is a chance that Mojo will not adapt well to program phase changes. Dynamo has shown that pre-emptive cache flushing addresses this issue and as part of our future work we intend to investigate this in more detail.

Exception Handling Support

Program points often get invoked in ways that are not always straightforward. The most often cited example of this is setjmp/longjmp where setjmp copies the

register state into a buffer and longjmp uses this information to transfer control back to the setjmp statement. This kind of control transfer must be intercepted otherwise the application might begin executing natively beyond Mojo's control. Most Windows applications use Structured Exceptions, C++ Exceptions, which are built on top of SEH, and OS callbacks, all of which are examples of control transfers that Mojo must manage.

In the Windows environment, each time control is returned to the application asynchronously, execution re-enters user mode through a dynamically linked library known as ntdll.dll (NtDll). After the machine state is correctly setup, NtDll transfers control to the application.

During program initialization Mojo patches various entry points in NtDll with calls to MojoContinue. This mechanism guarantees that the patched entries are executed before any code in the original program when the OS transfers control asynchronously to user mode.

When Windows dispatches an exception it passes the 'catch' address and other parameters, on the stack, to NtDll. As MojoContinue executes it first saves some state for the dispatcher and then invokes the dispatcher with the continuation address. As the dispatcher executes, it is aware of the exception state and prohibits path generation from polluting the PathCache.

For the particular case of exception handling, in addition to calling Mojo-Continue, the patched entry point in NtDll also maps the exception address back to an address in the original program. The application is tricked into thinking that the exception occurred at a point in the original program, thus making it easier to track down access violation errors. Currently the mapping is easy to do because Mojo only performs block reordering but further optimizations will make the address translations more difficult.

Thread Support

Our approach for handling multi-threaded applications in Mojo is to allocate a private Basic Block Cache per thread and a single shared Path Cache. Each thread has its own Basic Block cache and because the Basic Block cache is small (64K) we felt the cost of sharing outweighed the additional memory overhead. The path cache, on the other hand, is proportional to the size of executable code and is used more often, and thus the I-Cache performance improvements from sharing outweigh the synchronization costs.

There are three Path Cache operations that need to be synchronized in Mojo: (1) path linking (2) adding a path to a non-full cache (3) adding a path to a full cache. In all three cases, a global lock on the PathCache is acquired before the operation starts, and released after the operation ends. This lock prevents simultaneous path operations. However the common actions of entering the path cache, or running code in the path cache do not require any locks on the cache.

For the case of path linking, the operation consists of rewriting the offset of a given jump instruction in the code of a path. On the x86 platform this can be done with an atomic write operation while all other threads are running thereby eliminating the need for further synchronization. In the case of adding a path to a non-full cache, the operation consists of appending the new path code to the cache. The critical section for this operation is small since it consists of a simple pointer increment in the Path Cache memory allocator. This operation can also be done while all other threads are running, since no paths are being displaced. For the third case, where a path is added to an already full cache, some cache management needs to be done in order to make space. This might cause some of the paths to be deleted making it impossible to do when the threads are running. Since Mojo does not keep any path state on a thread while it is executing in the Path Cache it is necessary to suspend and move the threads on a per-

need basis. Although this is an expensive operation, it only occurs when the cache is full, or when Mojo decides to do cache management. These events occur rarely, and thus the cost will be amortized over the run-time of the program.

IV. Performance

In this section, we present measurements for three types of programs: (1) small synthetic kernels (2) medium size, single threaded SPEC benchmarks (3) large multi-threaded desktop applications that make use of exception handling. All of these programs were compiled with -O2 optimizations using the Microsoft Visual C++ compiler without profile feedback. The comparisons were made by running the programs natively and then running them under Mojo. Although Mojo does have a bailout mechanism similar to Dynamo, the measurements in this section were done with the bail-out mechanism turned off.

For these experiments the PathCache size was 512KB, except for Word, where it was 5 MB. The measurements were made on a single processor 800Mhz Pentium III with 256KB L2 cache and 512MB of RAM.

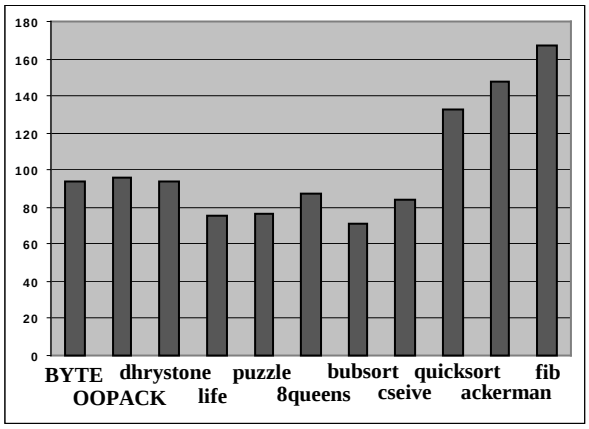

Figure 3: Normalized execution time of small kernels under Mojo where 100 indicates native speed

Figure 3 shows measurements for several small kernels. The numbers shown are execution times in normalized units, with 100 being the run-time of the optimized native binary. Some of these small kernels show performance improvements as large as 30% over the optimized native code. However, three applications, all of which are recursive, show large degradations. This shows that the path selection criteria do not work properly in the face of recursion.

Figure 4 shows measurements of several programs from the SPEC 2000 and SPEC95 benchmarks. The numbers shown are execution times in seconds for the reference input set, as reported by the SPEC testing harness. Two programs, MCF and compress, exhibit a gain when run under Mojo control. The rest of the tests show varying amounts of degradation. Further analysis must be performed in order to gain a better understanding of the performance degradations. In all cases where improvement was observed, the improvements were due to superior basic block layout that resulted in better I-cache locality and more predictable branches.

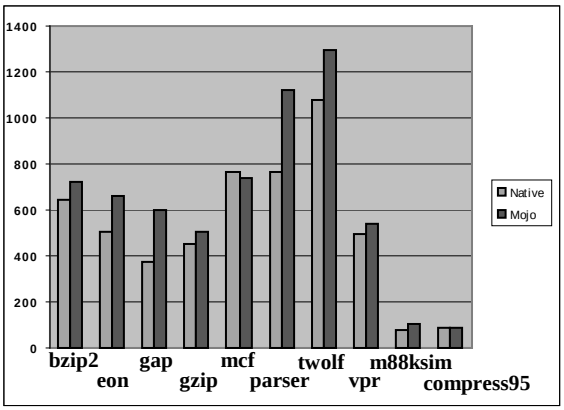

Figure 4: Execution time of SPEC benchmarks for Native vs. Mojo in seconds

Figure 5 shows measurements for two large, multi-threaded C++ applications: Microsoft Word and Visual FoxPro. The values given are execution times in seconds for the binary running either natively or under Mojo's control. The test scenario for Word consists of loading a realistic 2 MB Word file (an internal specification document), changing all the occurrences of the letter 'o' to the letter 'a', and then saving the file. The Visual FoxPro scenario consists of an

automated correctness test that queries an existing database.

In both cases, Mojo execution is slower than native execution. In the case of Word, Mojo execution is twice as slow as native, whereas for FoxPro it is about 40% slower.

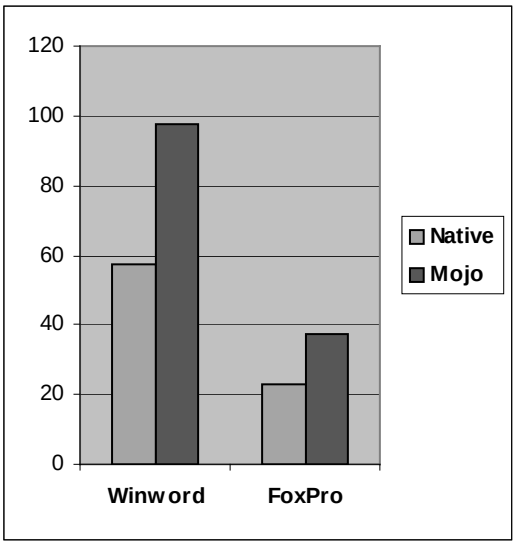

Figure 5: Execution time of Windows applications in seconds

To gain a better understanding of where the slow downs might come from, Figure 6 shows the distribution of the execution time for the last two applications. The Mojo.dll value shows how much time is spent in the dispatcher, the Path Builder, and the cache management code; the "In caches" value shows the time spent running code in either the Path Cache or the Basic Block Cache; the Ntoskrnl.exe value is the time spent running natively in kernel mode; and finally the "Other" value is the time spent running natively in user mode[†].

In both Word and FoxPro, more than 15% of the time is spent within Mojo.dll, which is a very high overhead. We believe this to be an indication that new paths are continually generated, thus requiring Mojo's frequent intervention. Indeed, we noticed that on some input sets for Word, Mojo generated more than 25 MB of paths, whereas the static size of the binary was only 5 MB.

However, the high number of paths generated is not the only source of problems. If we ignore the amount of time spent in Mojo.dll, then the amount of time spent in the caches, in the kernel and running user code natively is 75 seconds for Word, and 31 seconds for FoxPro. These values are still larger than the run-times of the native binaries, thus indicating that the path selection criteria could also be a source of slowdowns.

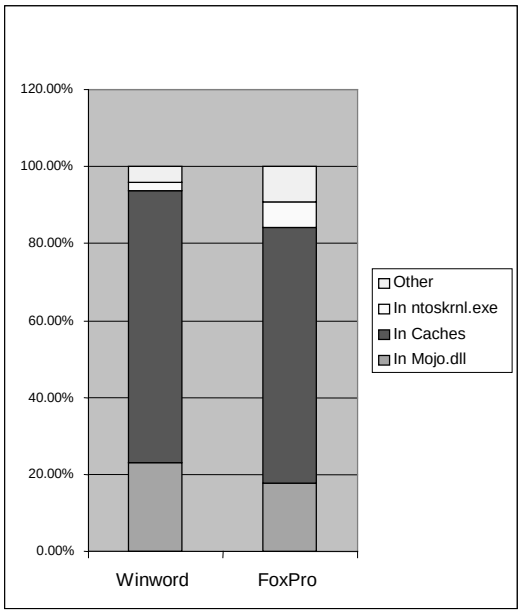

Figure 6: Time Distribution for Windows applications

V. Future Directions

Our initial primary goal was to build a run-time optimization infrastructure that supports large desktop applications with reasonable performance. Now that we have achieved this objective we are focusing our efforts on improving the performance of Mojo so that it rivals the best code optimized with profile feedback.

There are three classes of improvements that are currently being explored. The first class can be characterized as basic engineering, whereby the improvements come from fine-tuning the infrastructure. We expect to see

[†] Recall that some parts of NtDll are natively executed in user mode before Mojo regains control of the application.

significant performance improvements in this area because the performance bottlenecks have not been properly addressed. The second class of improvements is classic compiler optimizations on paths [13]. We expect a modest performance gain from these optimizations given the published gains from Dynamo. The third class of improvements revolves around investigating novel ways of using static information from the compiler. Though it unclear at this time how much of a performance gain this will entail we feel it is an interesting area for future research.

VI. Summary

This paper describes the Mojo run-time optimization system. Mojo is capable of handling a wide range of programs including multi-threaded applications that make use of exception handling. The paper also gives preliminary performance results for a variety of programs. Currently, Mojo can show performance improvement on a number of small applications and two SPEC benchmarks but shows degradations on several larger applications.

Acknowledgements

The authors would like to thank the performance group in the Programmer Productivity Research Center at Microsoft Research for support and encouragement of this work. We would also like to thank Jim Larus, Ben Zorn and our reviewers for their constructive comments on our work and valuable suggestions on improving this paper.

VII. References

[1] Bala, Vasanth, Evelyn Duesterwald and Sanjeev Banerjia. "Dynamo: A Transparent Run-time Optimization System", *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, Vancouver, Canada, June 2000.

[2] Klaiber, Alexander. "The Technology Behind Crusoe Processors", Available from http://www.transmeta.com, January 2000.

[3] Srivastava, Amitabh, et al. "Vulcan", Technical Report, Microsoft Research MSR-TR-99-76, September 1999.

[4] Srivastava, Amitabh and David Wall. "A Practical System for Intermodule Optimization at Link Time", *Journal of Programming Languages*, pp 1-18, March 1993.

[5] Muth, Robert, Saumya Debray, Scott Watterson, and Koen De Bosschere. "alto: A Link-Time Optimizer for the Compaq Alpha", To appear in *Software Practice and Experience.*

[6] Grant, Brian, M. Philipose, M. Mock, C. Chambers, and S. Eggers. "An Evaluation of Staged Run-Time Optimizations in DyC", *Proc. SIGPLAN'99 Conference on Programming Language Design and Implementation*, pp. 293-304, May 1999.

[7] Leone, Mark and R. Kent Dybvig. "Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization", Technical Report #490, Computer Science Department, Indiana University, Sept. 1997.

[8] Leone, Mark and Peter Lee. "Optimizing ML with Run-Time Code Generation", *Proc. SIGPLAN'96 Conference on Programming Language Design and Implementation*, pp. 137-148, May 1996.

[9] Burke, Michael, J.-D. Choi, S. Fink, et al. "The Jalapeno Dynamic Optimizing Compiler for Java", *Proc. 1999 ACM Java Grande Conference*, June 1999.

[10] Ebcioglu, Kemal and Erik Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility". *Proc. 24th Int'l Symposium on Computer Architecture*, pp. 26-37, June 1997.

[11] Hookway, Raymond and Mark Herdeg. "Digital FX!32: Combining emulation and binary translation", *Digital Technical Journal*, 9(1), August 1997.

[12] Barnes, Ronald, Ronnie Chaiken and David M. Gillies, "Feedback-Directed Data Cache Optimizations for the x86", *2nd ACM Workshop on Feedback Directed Optimization (FDO)*, November 1999.

[13] Hwu, Wen-mei et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, Kluwer Academic Publishers,Vol. 7, No. 1, pp. 229-248, Jan. 1993.