Examining the Connection Between Hardware Architecture and Virtual Machine Design

Ethan A. Kuefner Madhukar N. Kedlaya
University of California, Santa Barbara
{eakuefner,mkedlaya}@cs.ucsb.edu

Abstract

Virtual machines for programming language interpretation have become popular despite the long-standing generality that machine code is faster than interpreted code. By viewing interpreters as virtual machines, we can exploit the similarity between virtual machines and the underlying hardware architectures on which they run to take advantage of optimizations implicit in hardware and improve the performance of virtual machines. In this paper, branch prediction and cache performance are reviewed as two critical optimizations to be exploited by VMs.

1. Introduction

Interpreters have played a central role in the study of programming languages for as long as programming languages have been studied. Explicit in McCarthy's definition of LISP [8] is the notion of an evaluator, a program designed to run (evaluate) other programs. For many years, interpreters have been used to study the design and semantics of programming languages. Until recently, however, languages like FORTRAN, COBOL, and later C remained popular for programming, because of how fast programs compiled to machine code from these languages were.

Java, which appeared in 1995, was one of the first interpreted languages to use a so-called *virtual machine*, and has proved popular. Java programs are compiled down to Java bytecode, which is machine code for the Java Virtual Machine. Since Java, bytecode interpreters have become commonplace, and indeed, even in the original academic circles where interpreters were conceived and made popular, researchers have connected the old notions of interpretation as evaluation with the modern idea of interpreter as virtual machine [1]. With the advent of the virtual machine language runtime has come an opportunity to rectify the main flaw of interpretation alluded to above: the speed gap between interpretation of high-level code and execution of machine code. One of the major ways that this has been done is through exploitation of properties of the underlying hardware architectures on which these interpreters run.

We divide this paper into three main parts. In the first part, we discuss three papers related to improving hardware branch prediction accuracy for virtual machines, and in the second part, we discuss two papers related to exploiting memory models and caching. Lastly, we conclude by discussing the possibility of turning the tables and applying techniques developed in virtual machine design to the design of new hardware architectures.

2. Language Runtime Background

Designers of interpreted languages peform a delicate balancing act between the inclusion of interesting language features and the maintainence of acceptable performance. Modern production-quality interpreters are implemented in a variety of styles that span this continuum. Simplest among these design patterns is the recursive "big-step"-style evaluator, which walks the abstract syntax tree (AST) of the program in a recursive fashion and reduces AST nodes to values. The process of adding a new feature is then as straightforward as adding a handler for the new type of AST node. Because of its reliance on recursion and resulting tendency to exhibit nonlinear control flow, this style of interpreter is often accompanied by a strong performance overhead.

A now-common fix to this approach is the so-called *bytecode interpreter*, alluded to in the introduction. This style of interpreter works by performing a compilation of the program AST to an intermediate, flat bytcode language. The bytecodes are then fetched, decoded, and executed in a loop. Classically, this is done using a switch statement that will match opcodes to their corresponding handlers. Unfortunately, while this allows linearization of control flow, it introduces new overhead due to bytecode fetching and branching at the hardware level (as opposed to branching in the interpreted language).

3. Improving Branch Prediction Accuracy

Advanced threading techniques such as direct threading and indirect threading, which improve on the bytecode interpreter model by executing sequences of precompiled machine code to handle VM opcodes, manage to do away with most of the instruction fetch overhead but retain the potential for overhead from branch misprediction. This is because indirect branching is extremely frequent in interpreters, and

does not follow a pattern which can be resolved easily by today's hardware branch predictors. The branching behavior of an interpreter is somewhat unique in that it is entirely driven by the execution flow of the program being executed, so it may vary arbitrarily. The problem is exacerbated by the frequent tendency of dynamic languages to include features like dynamic typing and reflection, which are themselves frequently handled by additional conditionals.

Indeed, previous research [5] has found that 81%-98% of branches in a conventional switch-based bytecode interpreters are mispredicted, and even in more advanced interpreters with threaded code, the overhead is still on the order of 57%-63%. That interpreters undermine branch prediction in this manner is clearly a problem, and we discuss two attempts at fixing the problem from researchers.

3.1 Dynamo and DynamoRIO

Dynamo [2] is a pseudo-interpreter designed to perform hot trace specialization for native code by presenting the same interface as the processor for running native code, but transparently performing the specialization in software at runtime. This is similar to the JIT compiler model for language VMs, but operates at a much lower-level.

In [9] Sullivan et al. give an overview of DynamoRIO, a solution for reducing emulation overhead present in Dynamo through translation caching, developed jointly by MIT and HP Labs. They then observe that for interpreters, DynamoRIO's trace collection heuristics do not function correctly, and thus the impressive speedups typical of DynamoRIO are not present in this setting. This is because, as discussed, the control flow of an interpreter is unpredictable, due to the runtime dependency on the program being executed. Their solution was to extend DynamoRIO with an API to be used by language runtime developers. This API provides provides hooks to a special tracing framework to be inserted in the interpreter, which signal the trace collector to start and stop. This means that the interpreter itself can identify basic blocks in the interpreted code and correlate them with basic blocks in the underlying machine code that results, so that the tracing infrastructure has much more information about the execution than it would without the hints provided by these hooks.

3.2 Context Threading

In [3] Berndl et al. view the branch prediction accuracy problem as one of "contexts"–the prediction of a branch to be taken by an interpreter depends on the particular context given by the program being executed. The solution that results improves specifically upon the direct-threaded interpreter architecture by using subroutine threading.

In traditional direct-threaded interpreters the bytecodes are transformed into an array of labels called a direct threaded table (DTT), which is then indexed by a virtual program counter (vPC). A direct threaded instruction is a label to a specific region of the code in the interpreter that performs the opcode handling. After each opcode is executed the next opcode is "*dispatched*" by calling `goto DTT[vPC++];` i.e. an indirect branch instruction. In case of a branch instruction the vPC is computed based on the branch taken in the program being interpreted. This negates the switch-case overhead and has good cache behavior but as discussed, will still falter due to branch misprediction and pipeline flushes.

The solution to this problem is a variation on subroutine threading called *context threading*. In context threading, the bytecode instructions of the program are transformed into an array of function call instructions and the interpreter executes each one of them serially. The control flow instructions in the program are handled the same way as in direct threaded code (using indirect calls), but because we are now using function calls instead of simple jumps, the interpreter is able to provide more context for the instructions being executed, so that the branch predictor will be able to make better decisions. Though intuition might suggest that a series of function calls would execute slower than a series of jumps, results suggest otherwise. The context threading implementation described in the paper reduced the mean branch mispredictions by 95% for standard benchmarks on the SableVM virtual machine and the Inria OCaml interpreter on the Pentium 4 microprocessor when compared to traditional direct-threaded interpreters.

3.3 Modifying Instruction Layouts

In [4] Casey et al. describe two different techniques for improving branch prediction for virtual machines at the hardware level. The techniques are instruction replication and superinstructions, and both techniques involve restructuring the instruction stream in some way.

3.3.1 Instruction Replication

We can view a program's VM instruction listing as a multi-set, that is, a table of instructions together with the number of times each instruction occurs. In the paper, this is referred to as the "working set" of the program. Casey et al. observe that if an instruction occurs only once in a program's working set, then as expected, there is no problem with branch prediction; the instruction to occur following that instruction will be the same every time. However, if an instruction occurs more than once in the working set, then the behavior of the interpreter following the handling of this instruction will no longer be deterministic.

The fix proposed for this

4. Memory models and Caching

[Give a brief introduction describing the problem here. In garbage collection section we deal with the data cache and memory. The section subsection deals with instruction caches.]

4.1 Garbage Collection

Garbage collection involves reclaiming the precious heap space available after an object is no longer needed by a managed runtime system. This has been an active research agrea and most of the effecient implementations involve use technique called compaction. Compaction moves around the live objects in the heap into a compact region so that there is less fragmentation of the heap. This is a costly operation and usually introduces noticable pause times during program execution. In [10] Wegiel et al describe a way of using virtual memory system to perform this operation with minimum overhead. The main idea here is to exploit the fact that the dead objects often appear in clusters in the heap. Using virtual memory mapping/unmapping API provided by the operating system, the virtual page which in which all the objects are guaranteed to be dead is unmapped from the heap. Though this technique has its limitations since the collection phase doesn't always free all the dead objects, it is shown to be an efficient technique in collecting the objects from tenured region of the heap in a generational garbage collector. **[Not complete]**

4.2 Instruction caches

In [7] Lin et al describes a methodology of arrangement of language runtime code in memory to enable greater performance in cache-sensitive architectures. **[expand this even more]**

5. Future directions

[Important questions to be answered - What does is mean to design instruction caches for functional programs? What does it mean to design caches for non-linear data structures like objects in dynamic languages? What are the hardware/architectural modifications required to support better branch prediction strategies for dynamic scripting language interpreters? What about specialized hardware to speedup JIT optimizations and program analysis? If most of the objects created in a dynamic scripting language are short lived, do we really need to cache each object as soon as it is created? Can the compiler/program analyzer give hints to the branch predictor for better prediction and hint memory management system not to cache certain objects?]

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *Technical Report BRICS RS-03-14*, Mar. 2003.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349303. URL `http://doi.acm.org/10.1145/349299.349303`.

[3] M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 15–26, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: 10.1109/CGO.2005.14. URL `http://dx.doi.org/10.1109/CGO.2005.14`.

[4] K. Casey, M. A. Ertl, and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007. ISSN 0164-0925. doi: 10.1145/1286821.1286828. URL `http://doi.acm.org/10.1145/1286821.1286828`.

[5] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:2003, 2003.

[6] Y. Jo and M. Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 355–374, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384643. URL `http://doi.acm.org/10.1145/2384616.2384643`.

[7] C.-C. Lin and C.-L. Chen. Transactions on high-performance embedded architectures and compilers iii. chapter Cache sensitive code arrangement for virtual machine, pages 24–42. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-19447-4. URL `http://dl.acm.org/citation.cfm?id=1980776.1980779`.

[8] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, Apr. 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL `http://doi.acm.org/10.1145/367177.367199`.

[9] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. Dynamic native optimization of interpreters. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, IVME '03, pages 50–57, New York, NY, USA, 2003. ACM. ISBN 1-58113-655-2. doi: 10.1145/858570.858576. URL `http://doi.acm.org/10.1145/858570.858576`.

[10] M. Wegiel and C. Krintz. The mapping collector: virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 91–102, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346294. URL `http://doi.acm.org/10.1145/1346281.1346294`.