Examining the Connection Between Hardware Architecture and Virtual Machine Design

Ethan A. Kuefner Madhukar 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.

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 [7] 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 non-linear 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.

Continuing here.

Other alternatives to the switch-case approach like direct threading, indirect threading also suffer from the similar branch misprediction overhead. There branches that are taken depend on the flow of execution or the stream of bytecodes of the program being executed. It does not follow any pattern which is very much the foundation of branch prediction strategies implemented in the hardware. It is important to note here that the branches that we are describing are the ones in the interpreter loop and should not be confused with the branches in the program being interpreted. The problem with branch mis-prediction is even more exagerated in the case of Dynamic Scripting Languages (DSLs) where the opcode handlers for a dynamic instruction have to first determine the types of the opcodes and them perform operations based on the observed types. This is generally implemented using a *if-else* chain or a *switch-case* construct adding to the number of branches mispredicted during execution.

3. Improving Branch Prediction Accuracy

3.1 Dynamo and DynamoRIO

Dynamo is an pseudo interpreter which interprets machine code and specializes the hot traces that are executed at runtime. It follows the same philosophy as a JIT compiler but at a much lower level. Sullivan et. al presented DynamoRIO, based on IA-32 version of Dynamo, as a solution to reduce the interpretation overhead. Using Dynamo or DynamoRIO naively as a binary optimizer for any interpreter does not yield any speedup. This is because DynamoRIO relies on its trace collection heuristic to optimize a binary and as mentioned above the trace of execution of any interpreter depends on the program being interpreted which is unpredictable at runtime. To solve this problem DynamioRIO infrastucture provides APIs to language runtime developers to instrument their interpreters with hooks to a special traceing framework. The hooks provide signals to the underlying framework when to start and stop the trace collection. The idea here is to make sure that the trace that is collected matches the program that is being interpreted rather than the interpreter that is interpreting it. This gives lot more information to the tracing infrastructure to work on and optimize.

3.2 Context Threading

3.3 Instruction replication and Superinstructions

4. Memory models and Caching

4.1 Garbage Collection

4.2 Instruction caches

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] 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`.

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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`.