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 truth 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. Branch Prediction

Programming language design has always been a tricky area of research. More importantly the design of the runtime plays an important role in enabling interesting language features while maintaining acceptable performance. When we consider the implementation of modern language runtimes, implementing a quick interpreter has mostly been the first choice. The main reason for this is the ease of implementation and the ease of extensibility. Unfortunately, interpreters come with a large performance overhead. Classical big-step style interpreters which traverse through the AST of a program suffer from this. An improvement over this design are the bytecode based interpreters where the bytecodes are fetched, decoded and then executed in a loop. The classical way of impementing this is using switch statement matching the opcode with various possible cases and executing the corresponding opcode handler.

The major drawbacks of this approach are the bytecode fetching/dispatch overhead and branch misprediction. It is well documented in previous research[5] that switch based dispatchs are mispredicted 81% - 98% of the time. Other alternatives to the switch-case approach like direct threading, indirect threading also suffer from the similar branch misprediction overhead (57% - 63%) [5]. 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 misprediction 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 mis-predicted during execution.

2.1 Dynamo and DynamoRIO

Dynamo[2] 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.

In [8] 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.

2.2 Context Threading

Context threading takes a different approach on solving the problem. In [3] Berndl et al rephrase the above problem as a "Context problem". Their solution improves upon the direct-threaded interpreter architecture by using subroutine threading.

In direct-threaded interpreters the bytecodes are transformed into an array of labels called Direct Threaded Table (DTT) indexed by a virtual PC (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. indirect branch. 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 still suffers a lot due to branch misprediction and pipeline flushes.

The solution to this problem is a variation of 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 direct threaded code (using indirect calls). The main idea is to provide different contexts for different instructions that are executed so that the branch predictors can make better decisions. Though at first glance executing a series of function calls seem to be slower compared to a series of jumps, the results suggest otherwise. Context threading reduced the mean branch mispredictions by 95

2.3 Instruction replication and Superinstructions

In [4] Casey et al describe various techniques of improving branch prediction. **[Need to expand this section]**

3. 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.]

3.1 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 [9] 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]**

3.2 Instruction caches

In [6] 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]**

4. 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] 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`.