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 [9] 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 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 [10] 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 multiset, 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[1].

The fix proposed for this is a technique called *instruction replication*, which, as the name suggests, involves making multiple copies of a given instruction. For an instruction that occurs more than once in the working set, we can copy the instruction as many times at it occurs in the working set, and then tag each instruction separately. In doing this, each occurrence of the instruction will have a unique entry in the BTB, and thus for each occurrence of the instruction, the following instruction can be predicted separately, removing the inaccuracy.

3.3.2 Superinstructions

The technique of using *superinstructions*, which are VM instructions that represent a sequence of multiple instructions, has been well-known outside of this context, for the purposes of code size and fetch overhead reduction. The Casey paper, however, investigates superinstructions in the context of improving branch prediction accuracy. The results of their experiments suggest that use of superinstructions actually reduces mispredictions more than it reduces fetch/dispatch overhead. They note that the likely reason for this is the fact that a superinstruction standardizes a sequence of instructions that will execute deterministically each time the superinstruction is called.

4. Memory Models and Caching

Spatial locality is an important principle when dealing with caches, which have been an important optimization at the hardware level. As one might expect, virtual machines have similar memory models to their hardware counterparts, and thus there are a number of optimizations which take advtange of the principle of spatial locality, which holds even in the virtual setting. While locality applies mainly in the context of cache performance for hardware, and this is mirrored in the virtual machine setting, it can also be exploited in an interesting manner for the purposes of improving garbage collection performance. We will discuss first the application of locality to garbage collection, and then a technique for arranging VM code to provide performant instruction caching at the hardware level.

4.1 Garbage Collection

Many programming languages backed by runtimes do not allow users to manage memory manually, as they can in languages like C. Rather, memory is managed automatically and the runtime will attempt to reclaim memory that is no longer in use for when it is once again in demand. This process of automatic memory management is called garbage collection. Garbage collection is an extremely active area of research, and one of the most performant solutions to emerge so far is the so-called *compacting collector*. Compaction moves around the live objects in the heap so that the heap is less fragmented and it becomes possible to allocate larger chunks of memory. Because of the amount of copying that this technique requires, it is often responsible for long *pause times*. Pause times, which are the durations for which the program execution must pause so that the garbage collector may execute, are an important metric in measuring collector performance.

[1] This assumes the use of a branch target buffer, which for the purposes of this paper we view in a simplistic manner.

In [11] Wegiel et al. describe a method of using an operating system's virtual memory subsystem to perform this compaction operation with minimum overhead. An important observation that they exploit is the fact that the dead objects often appear in clusters in the heap. Spatial locality is most frequently associated with liveness, or "next use", but here, it plays somewhat of a dual role. Using the virtual memory mapping/unmapping API provided by the operating system, pages in which all of the objects are guaranteed to be dead is unmapped from the heap. Though this technique has limitations in that it does not 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.

4.2 Instruction caches

In [8] Lin et al. describes a methodology for arranging language runtime code in memory to enable greater instruction caching performance. In contrast to the previous application of locality, this application holds more in common with traditional applications of locality, in that it views locality as a metric for assessing next use of a memory location. Their technique relies on a mathematical model that they build to model execution flows of real applications, and exploits this information to place correlated instructions near one another, so that an instruction cache will be able to bring pages containing instructions frequently used together into memory all at once. Interestingly, they test their technology on NAND flash memory so slow that their experiment would not work if their technique were not sufficiently performant, and in so doing, demonstrate that their approach is viable.

5. Future work

In both the branch prediction and locality exploitation categories, there is interesting future work to be done. As far as branch prediction, one potential approach to tying the hardware yet more closely to the virtual machine would be to provide hardware hinting as to the likelihood of taking or not taking a given branch. The Pentium 4 had a feature that allowed this [6], but it was removed from later models, presumably because of the complexity of implementation. As far as we know, this is an open opportunity for research. On the locality side, researchers are working on improving locality for nonlinear data structures, as many such structures are common in software. Interpreters contain a myriad of data structures would be ripe for this sort of optimization. In a 2012 paper from OOPSLA [7], Youngjoon Jo and Milind Kulkarni show a scheme for improving locality for tree traversals, which they say they plan to generalize.

6. Conclusion

In conclusion, we have shown several examples of how making connections between virtual machine architectures and the architectures of the hardware machines they run on can be a valuable technique for improving the performance of language runtimes. In particular, tuning interpreters to coexist with hardware branch predictors and exploiting locality are two overarching techniques that have been successful in improving runtime performance in concrete settings.

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] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-045US. January 2013.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

A. Sample Homework Problem

Here is a working bytecode compiler and interpreter toolchain for a simple programming language, written in C++. Use this DynamoRIO tutorial to instrument the interpreter with hooks for straight-line code as described in this paper, and then compare the performance with and without the tracer enabled. What is the speedup that you observe?