Seminar aus Programmiersprachen

185.307

Context Threading

Florian Fest
ff@generationfun.at
937/0125496

Based on:
Context Threading:A flexible and efficient dispatch technique for virtual machine interpreters
Marc Berndl, Benjamin Vitae, Mathew Zaleski and Angela Demke Brown

Wien, WS 2006/07

1 Abstract

This paper examines the article "Context Threading" by Berndl, Vitae, Zaleski and Brown published in 2005. [1] It will present a short description of their published work and will discuss it in context of other research done in this field. In the remainder of this paper Berndl, Vitae, Zaleski and Brown will be referred as the authors. Efficient interpreters are mostly implemented by means of Direct Threading. They use indirect branches to dispatch opcodes. This leads to a large number of indirect branches (according to Ertl [5] up to 13% of over all executed instructions in interpreters). These branches are further more not biased like branches in normal cpu workload thus causing a large number of miss predictions on modern pipelined processors. The authors introduce an implementation technique for interpreters similar to subroutine treading [2] [3] they call context threading. Context Treading reduces the number of miss predictions compared to direct threading by providing biased branches. Native subroutine calls are used for dispatching thus allowing an efficient return address prediction. Some other supplementing optimizations are introduced to handle branches not caused by the dispatch. The performance improvement is evaluated by benchmarks and can be compared to that of selective inlining.

2 Introduction

Several advantages make interpretation attractive for the implementation of programming language systems. Interpreters are easy to implement, compact and portable compared to native code compilers. They also allow the use of interactive debugging concepts. The major down side of interpretation comes with its poor performance behavior. Important systems, such as the virtual machines provided by SUN and IBM, operate in mixed mode in order to use the advantages of both techniques, interpretation and just in time compilation to native code. So interpreter performance is relevant for the overall behavior of such systems.
Direct threading has been known as a very efficient layout for interpreters but recently Ertl and Gregg [5] showed that it's performance is limited in practice on modern architectures.
Todays microprocessors use a heavily pipelined architecture to achieve their performance goals. Branch predictors are used to determine the branch target in order to keep the pipelines full. Miss predictions, so called pipeline or branch hazards, require flushing of the pipelines thus leading to a serious performance losses. Branch predictors exploit the correlation between program counter (`PC`) and branch target. With direct threaded interpreters, branches are correlated to the virtual program counter(`vPC`) rather than to the hardware program counter. This correlation is not exposed to the hardware thus leading to a high number of miss predictions. Context threading is a technique for interpreters which reduces the number of branch hazards by making the correlation between `vPC` and target visible to hardware. This is achieved by aligning virtual and real machine useing native `call` and `return` instructions.

3 The Context Problem

When a virtual program is executed, the interpreter dispatches its virtual instruction in sequence. The vPC indicates the current instruction in a sequence of virtual instructions. Virtual instructions consist of an opcode and optional arguments. Their precise layout depends on the implementation of the dispatch. Switch based interpreters use a for loop to fetch the opcode from vPC and than execute a switch over it. The individual opcodes are implemented in case blocks. The use of first class labels as provided in gcc allow a more efficient implementation in form.
With direct threading opcodes represent the address of a label with the actual implementation. The virtual program is a list of jump-target addresses and dispatch is done by an indirect jump. Each opcode implementation increments vPC and performs a dispatch to the next after its own work is finished. The advantage over the switch based type is that only one jump per opcode has to be executed and the range checks compilers usually create when translating switch statements can be left out.

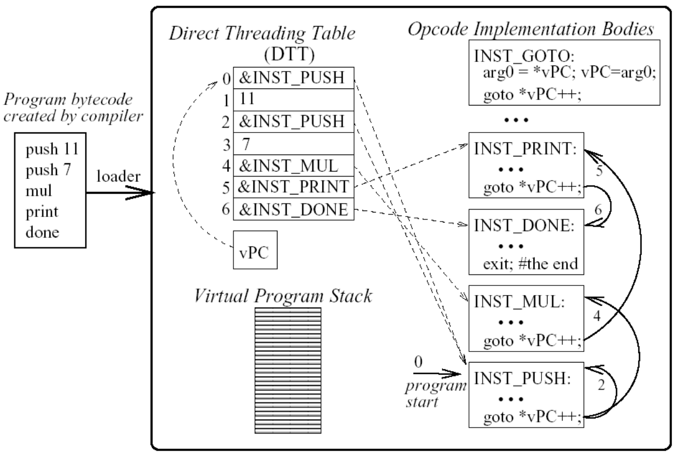

Figure 1: Direct Threaded Interpreter [1]

Figure 1 shows an direct thread interpreter executing a small program. The virtual program is translated to a list of jump-addresses and parameters. This structure will be called the Direct Threading Table (DTT). The jump at the end of INST_PUSH leads back to the start of INST_PUSH when it is executed the first time. The second INST_PUSH performs a jump to INST_MULL. The observed behavior is typical for direct threading. The target of a jump is not correlated to its location, resp. the PC. Although there is a correlation between vPC and the target, the branch predictor is not able to exploit it, because it is effectively hidden.
switch based interpreters perform even worse concerning branch hazards because all their dispatch operation take place in a single point, the indirect jump generated from the switch statement.

The authors called this effect the context problem, because the missing context of the indirect branch instruction makes prediction almost impossible. The next section will describe their method of providing more context to hardware predictors and thus enabling better prediction.

4 The Concept of Context Threading

The disadvantages of direct treading have already been discussed. Although it has the major benefit to offer a good cache behavior.[9] One way of providing context to branches and even eliminate some of those which are caused by the dispatch, would be inlining of opcode bodies. This would also allow optimizations across the boundaries of individual opcodes.
Another option would be the use of super instructions, where common sequences of opcodes are combined to a single super instruction. Determining this sequences can be done statically [6] or dynamically [4]. Normally the number of super instructions is limited by the opcode encoding (often 1 byte).
The techniques described above have a common disadvantage, they achieve less branch hazards but to do so they require more space in the instruction cache. This might lead to cache misses thus resulting in an increase in runtime. Context Threading achieves better branch prediction behavior with only a light increase of code size.
The following sections describe the handling of the three sources of branches in an interpreter and their treatment in context threading.

4.1 Opcode Dispatch

Since every opcode is dispatched at the beginning of its processing this is the main source for branches. As already discussed, with direct threaded interpreters this is done by an indirect branch thus leading to many miss predictions. In context treading replaces the branch with a native subroutine call. Native subroutines should not be confused with function calls in higher programming languages which normally include saving and restoring registers. Almost all modern microprocessors provide an efficient prediction mechanisms for the return address of native subroutine calls in form of a return address stack. The native `call` instruction pushes the return address, usually the next instruction, onto the return instruction stack and performs the control transfer. The native `return` performs a control transfer back to the last address on the return address stack.
An opcode is dispatched by `call` to the native subroutine implementing its body. Each body ends with a `return`. Opcodes manipulating the virtual control flow are handled in a different way. Their processing will be discussed in section 4.2 and 4.3.

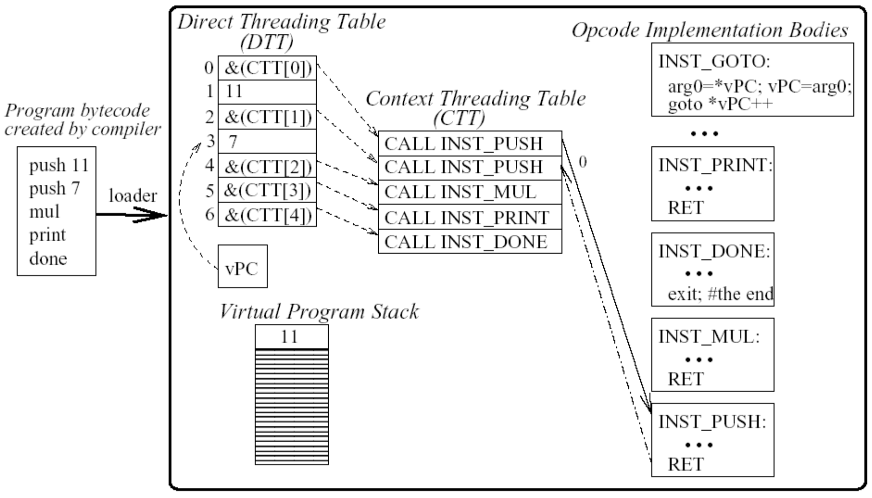

Figure 2: Context Threading Interpreter [1]

The actual implementation shall be described with respect to Figure 2. A new structure the Context Treading Table (CTT) is introduced. The CTT represents a program as a sequence of call instructions, thus allowing easy prediction of the return addresses. The authors use the term context treading because the because the hardware address of each call instruction in the CTT provides context to hardware so that an efficient prediction is possible. The direct treading table is still necessary for storing of operants and resolving the control flow of the virtual program. The entries in the DTT refer to the CTT rather than pointing directly to opcode implementations.
Compared to direct treading context threading replaces on control transfer with two transfers but due to the much better predictability of these two transfers an over all performance gain is achieved.

4.2 Branching Opcodes

The simplest way to handle branching opcodes would be branching to an opcode body which manipulates the `vPC` and then performs an appropriate branch. When following this approach all branches share a single location, and no context to the branch is provided as it can be seen at the INST_GOTO statement in Figure 2.
Additional context is provided to virtual branches by replicating them in the CTT. A virtual branch can now be processed by `call` statement and end with a `return`. The actual replication happens after the call is executed and is placed directly after the call instruction in the CTT. The `return` transfers control back to the replicated body. The authors refer to this technique as branch replication.Branch replication offers a simple way to provide context to each virtual branch, but it also has some down sides. Three control transfers are necessary to execute a single branch thus producing more overhead. Furthermore only the dispatch part is replicated and indirect branches are not replaced by direct

branches although this is possible in may cases. The authors use branch replication only to handle virtual indirect branches and exceptions.
All other virtual branches are fully inlined into the CTT (the upper part of Figure 3 shows a inlined virtual goto) Whenever possible indirect branches are replaced by direct branches. This allows more efficient prediction through conditional branch predictors instead of branch target buffers. This so called branch inlining leads increases code size but this is acceptable since most branch instructions are simple thus keeping the growth within limit.

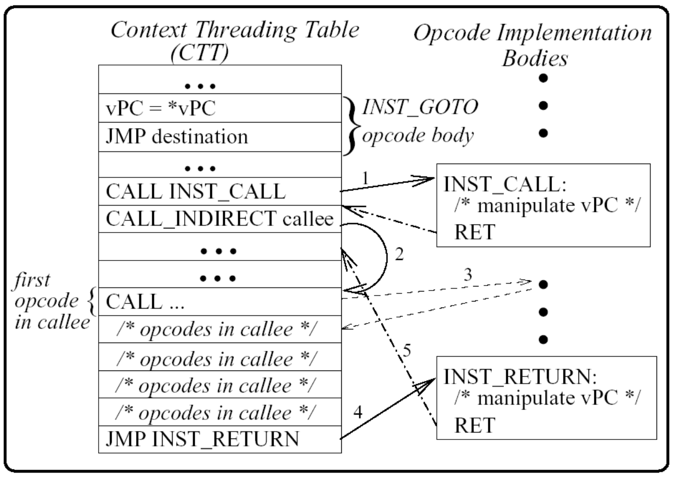

Figure 3: Inlined Branch and Virtual Call [1]

4.3 Virtual Call and Return

The last source of branches that need to be taken into consideration are the virtual call and return instructions. Especially the processing of an virtual return can be complicated due to the fact that one return may go back to multiple calls. Modern microprocessors already provide a hardware solution for this problem, the return address stack which has been already exploited for the opcode dispatch. The authors introduce a technique to use this hardware also for virtual call and returns.
Basically the call is splited into two virtual instructions both using native calls and returns. (as seen in Figure 3) The first subroutine call (1) manipulates the `vPC` so that it points to the callee while the second call (2) pushes the return address onto the stack and performs the actual control transfer. Then the body can executed in the usual way (3). The last instruction of the body is a special return instruction (4) performing the control flow transfer back (5).
An actual implementation of this method called apply/return inlining has to deal with a number of practical obstacles such as dealing with the VM-Stack or avoiding interference with other components using the return address stack.
For a detailed description the reader might refer to [1].

5 Evaluation

The authors applied context threading to Ocaml [7] an interpreter for Caml and to SableVM a Java VM provided by the Sable group [10]. They have run benchmarks of their implementations using Pentium IV and Power PC processors. On Pentium IV the use of context treading reduced the pipeline hazards by 95% and leaded on Power PC to a reduce of stall cycles (comparable to pipeline hazards) between 76% and 82%. Further more an overall performance gain compared to direct threading of about 25% for Java on both architectures, 19% for Ocaml on Pentium IV and 37% for Ocaml on Power PC has been detected.

6 Related Work

Context trading uses many concepts of subroutine threading. Subroutine threading is a rather old scheme for interpreters. Descriptions can be found at [2] [3] Once developed it has soon taken a back seat because native `call` and `return` instructions have been quit expensive on old architectures. Modern architectures however provide efficient `call` and `return` implementations thus making subroutine threading attractive when branch hazzards shall be reduced.
Another way to conquer the context problem are selective inlining as Piumarta [8] describes it or the use of dynamic super instructions introduced by Ertl [4]. These interpreter concepts eliminates branches by replicating opcode bodies. The results gained with selective inlining are comparable to those gained by context threading and even slightly exceed them in some points. Context threading uses some inlining techniques to cope with the virtual control flow.

7 Conclusion

In order to develop efficient interpreters for modern processors their pipelined architecture has to be taken into concern, otherwise performance is reduced by branch hazards. Berndl, Vitae, Zaleski and Brown present a technique which provides context to hardware predictors thus reducing miss predictions and still keeps most advantages of a simple interpreter layout, such as a small cache foot print and the simple layout. As context threading shows, on today's cpus good behavior in means of branch prediction, may even overwhelm a more efficient dispatch scheme.

References

[1] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of CGO-4*, 2005.

[2] Charles Curley. Life in the fastforth lane. *Forth Dimensions*, 1993.

[3] Charles Curley. Optimizing fastforth: Optimizing in a bsr/jsr threaded forth. *Forth Dimensions*, 1993.

[4] M. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters, 2003.

[5] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[6] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. Vmgen — a generator of efficient virtual machine interpreters. *SoftwarePractice and Experience*, 32(3):265–294, 2002.

[7] The Caml Language. Caml-homepage. http://caml.inria.fr/.

[8] Ian Piumarta and Fabio Riccardi. Optimizing direct-threaded code by selective inlining. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.

[9] Theodore H. Romer, Dennis Lee, Geffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 31, pages 150–159, New York, NY, 1996. ACM Press.

[10] Sable-Mcgill. Sable-homepage. http://www.sable.mcgill.ca/.