Interpreter Register Autolocalisation: Improving the performance of efficient interpreters

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
guillermo.polito@univ-lille.fr

Nahuel Palumbo
Soufyane Labsari
Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
name.surname@inria.fr

Pablo Tesone
Pharo Consortium
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
pablo.tesone@inria.fr

KEYWORDS

virtual machine, interpreter, optimisation, code transformation

1 PROBLEM

Language interpreters are generally slower than (JIT) compiled implementations because they trade off simplicity for performance and portability. However, they are still an important part of modern Virtual Machines (VMs) as part of mixed-mode execution schema. The reasons behind their importance are many. On the one hand, not all code gets *hot* and deserves to be optimized by JIT compilers. Examples of cold code are tests, command-line applications, and scripts. On the other hand, compilers are more difficult to write and maintain, thus interpreters are an attractive solution because of their simplicity and portability. In the context of this paper, we will center on *bytecode interpreters*.

Interpreter performance has been a hot topic for a long time, where several solutions have been proposed with different ranges of complexity and portability. On the one hand, some work proposes to optimize language-specific features in interpreters such as type dispatches using static type predictions, quickening [3] or type specializations [18]. On the other hand, many solutions focus on improving general interpreter behavior by minimizing branch miss-predictions of interpreter dispatches and stack caching. Solutions to branch mis-predictions propose variants of *code threading* [1, 4, 6, 7, 10] and improving it further with selective inlining [14]. Some solutions aim for minimizing branch miss-predictions by modifying the intermediate code (*e.g.,* bytecode) design with super-instructions [15] and register-based instructions [9, 16]. Stack caching [5] proposes to optimize the access of operands by caching the top of the stack.

interpreter registers are also related to stack caching: interpreter variables that are critical to the efficient execution of the interpreter loop. Examples of such variables are the instruction pointer (IP), the stack pointer (SP), and the frame pointer (FP). Interpreter registers put pressure on the overall design and implementation of the interpreter:

Req1: Value access outside the interpreter loop. VM routines outside of the interpreter loop may require access to interpreter registers. For example, this is the case of garbage collectors that need to traverse the stack to find root objects, routines that unwind or reify the stack, or give access to stack values to native methods.

Req2: Efficiency. Interpreter registers are used on *each* instruction to manipulate the instruction stream and the stack. Under-efficient implementations have negative impacts on performance.

These two requirements are opposing in the sense that **Req1** demands that register values are stored on globally accessible memory, either in the heap or the data sections of the process, while **Req2** would benefit from putting those values in registers and avoid memory accesses at all.

2 INTERPRETER REGISTERS IN SLANG, THE PHARO VM GENERATOR

The Pharo Virtual Machine is an industrial-level Virtual Machine written in Pharo itself for the Pharo language [2]. The VM implements at the core of its execution engine a threaded bytecode interpreter, a linear non-optimising JIT compiler named Cogit [13] that includes polymorphic inline caches [11] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [19]. The Pharo Virtual Machine is written in a subset of Pharo that is transpilable to efficient C using Slang, a Smalltalk-to-C VM-specific transpiler [12]. Slang operates by translating a group of classes into a single C file. Methods are translated into functions, message-sends are translated as function calls. While the Pharo source program presents dynamic behavior such as polymorphism, exceptions, or runtime reflection, Slang does not allow many of those: it either forbids them at translation time or generates invalid C code.

The interpreter is written following Pharo Smalltalk coding conventions as shown in Figure 1 where each bytecode and native method (primitives) is implemented with a different method. Slang then inlines bytecode methods inside the interpreter loop to produce an efficient token threaded interpreter [4]. Primitives are translated as independent functions outside of the interpreter loop. All communication between bytecodes and primitives happens through the stack.

Interpreter registers are manually duplicated in Pharo's interpreter: they have a local and a global version. Local interpreter registers are defined as local variables in the generated interpreter loop function, and global interpreter registers are defined as global variables available to the VM outside of the interpreter loop. The insight behind this manual optimization is that compilers are capable of better optimizing reads/writes to local variables putting them into registers, while they are mostly unable to do that when declared as global variables.

In this manual approach, it is up to VM developers to copy the values from the local to the global register and vice-versa. However, the manual approach has a high cost when developing and debugging the VM. First, since bytecodes are normal Pharo methods and Slang performs aggressive inlinings inside the interpreter loop,

```
Interpreter >> pushReceiverBytecode
    self fetchNextBytecode.
    self internalPush: self receiver.
Interpreter >> pushConstantTrueBytecode
    self fetchNextBytecode.
    self internalPush: objectMemory trueObject.
```

Figure 1: Example of Bytecode Implementation in Slang

VM developers cannot easily predict when to (a) use the local or the global register in their code because (b) when to perform the copy from/to local registers. Second, many methods are duplicated because of this global/local duality: there is one version using the local register and meant to be inlined in the interpreter loop there is a second version using the global register. Finally, this hand-coded approach makes it difficult to systematically experiment and validate this optimization in modern hardware.

3 INSTRUCTION REGISTER AUTOLOCALIZATION

In this article, we propose an automatic interpreter code transformation that satisfies **Req1** and **Req2** called *autolocalization* for the Slang VM generator. Our optimization transforms Slang's intermediate representation before it's translated to C. It automatically localizes interpreter registers inside the interpreter loop function and exports their values when exiting the interpreter to *e.g.,* call expensive routines like garbage collection. This automatic transformation removes most of the manual burden from VM developers while still taking benefit from the underlying C compiler optimizations.

Our optimization works as a three-step transformation as illustrated in Algorithm 1. First, the interpreter registers are declared as local variables in the interpreter loop function, and all usages of them are replaced by their local versions (Section 3.1). Then, all calls exiting the interpreter loop are wrapped with copy statements that synchronize the local and global registers (Section 3.2). In such a way, the interpreter loop remains efficient as far as it does not call external functions (**Req2**), and external functions access interpreter register values when they require it (**Req1**). In addition, some statements containing nested expressions need to be linearized before register synchronization for the correctness of the transformation (Section 3.3)

Algorithm 1: Autolocalization algorithm overview

1: ReplaceGlobals()
2: LinearizeStatements()
3: WrapExitPoints()

3.1 Variable localisation

The first step in our transformation declares local version of the interpreter registers within the interpreter loop function and replaces all usages of global register variables with their local versions. The global definitions of the localized variables are not removed because functions outside of the interpreter loop still require access to them. Thus, special care must be taken on the function entry and exit points to synchronize the global state with the global state. Function entry should perform register value *localizations*; *i.e.,* copy the values from the global variables to the local variables. Function exit points such as return statements must be preceded with register value *globalizations*; *i.e.,* copy the values back from the local variables to the global variables.

Figure 2 illustrates this transformation with an example in pseudocode. The original interpreter code declares a global register variable called `register1` which is accessed within the interpreter loop. The first step of our transformation introduces a local variable called `local_register1` and replaces all reads and writes from `register1` to `local_register1`. Moreover, our transformation adds a globalization statement before the `return` statement at the end of the function copying the value of `local_register1` to `register1`.

3.2 Synchronization Around Exit Points

The interpreter loop may perform calls to runtime functions providing services such as garbage collection. For the purpose of this article, we call such calls *interpreter exit points*. Such runtime services may use the values of the interpreter registers to *e.g.,* push/pop values to the stack or unwind it to implement exception handling. In that case, we must synchronize the values of interpreter registers around interpreter exit points. Thus, those register values must be *globalized* before calling the exit point and *localized* upon return to the interpreter.

Figure 3 illustrates the result of this process. A local value in variable called `local_register1` is copied to a global register variable called `register1` before the `exit_point()` call. Then, this function gains accesses to the synchronized global register variable `register1` for reads and writes. Finally, the value in `register1` is copied back to `local_register1` once `exit_point()` call is finished and execution returns to the interpreter loop.

3.3 Nested Statement Linearization

Another concern for correctness is nested expressions where local register variables are involved, as illustrated in the example in Figure 4. In these cases, the interpreter register variables synchronization must be performed between the local register computation and the interpreter exit point call to ensure the consistency of values. Our transformation performs a *linearization* on these statements to avoid nested calls. Statement linearization creates temporary variables for nested evaluation results and moves all subexpressions as statements of a single block.

After these transformations, it is possible to globalize and localize around the exit point calls with statement granularity. Figure 5 shows an example of the wrong and correct transformations after applying linearization and interpreter register synchronization over Figure 4.

```
var register1; // global
function interpret() {
  ...
  while(1) { switch(bytecode){
    ... register1 ... // global reads and writes
  } }
  return;
}
```

(a) Before transformation

```
function interpret() {
  // localisation: copy from global
  var local_register1 := register1;
  ...
  while(1) { switch(bytecode){
    ... local_register1 ... // local reads and writes
  } }
  // globalisation: copy back to global
  register1 := local_register1;
  return;
}
```

(b) After transformation

Figure 2: Global access to interpreter registers

```
... // inside the interpreter loop
register1 := local_register1; // globalization
exit_point();
local_register1 := register1; // localization
...

// outside the interpreter loop
function exit_point() {
  ... register1 ... // global reads and writes
}
```

Figure 3: Globalization and Localization points

```
exit_point(local_register++);
```

Figure 4: Nested statements where exit point and local register are involved

```
// Wrong transformation
register := local_register;
exit_point(local_register++); // register has wrong value
local_register := register;

// Correct transformation
t1 := local_register++;
register := local_register;
exit_point(t1);
local_register := register;
```

Figure 5: Linearized Statements

3.4 Optimizing exit points with call-graph analysis

Globalization and localization around exit points need not to synchronize all interpreter register values, but only those that are *used* by the called function. To further optimize interpreter register synchronization, we compute a recursive call-graph for each exit point call and analyze the uses of the global register variables. We then globalize and localize only used interpreter register variables around each exit point call.

Moreover, there are cases where the call-graph can not be analyzed because the call target is statically unknown. This is the case of function pointers and non-statically available functions. If one of such cases is found in an exit point, we conservatively synchronize all the interpreter registers.

4 PRELIMINARY RESULTS

We present preliminary results of our optimization by comparing the effect of localizing several interpreter registers on a set of benchmarks. The main goals of this evaluation is to see what is the effect of different localization configurations on a token threaded interpreter performance (Section 4.2), and to evaluate wether this automatic approach is worth its implementation in comparison with a manually handcrafted approach (Section 4.3).

4.1 Experimental Platform

We implement interpreter register autolocalization in Slang, Pharo's VM generator framework, to evaluate its impact on Pharo benchmarks. Before our implementation, the Pharo VM included variable localisation as a *manual* optimisation carefully handcrafted in the VM source code, we refer such configuration as *manual* in our benchmarks and we use it as baseline. We replaced the manual localization by *automatic* localization gaining the ability to control localization globally, enabling further interpreter modifications and making it easier to perform the following benchmarks in a systematic fashion.

We evaluate our approach on the suite SMark benchmark suite implemented for Pharo [17] containing

- microbenchmarks for different Pharo language aspects (array access, message sends),
- larger Pharo programs such as the bytecode compiler,
- implementation of the Computer Language Benchmarks Game [8].

All benchmarks were run on a 2015 MacBook Pro, 2,9 GHz Intel Core i5, 16 GB 1867 MHz DDR3, MacOS 10.14.6.

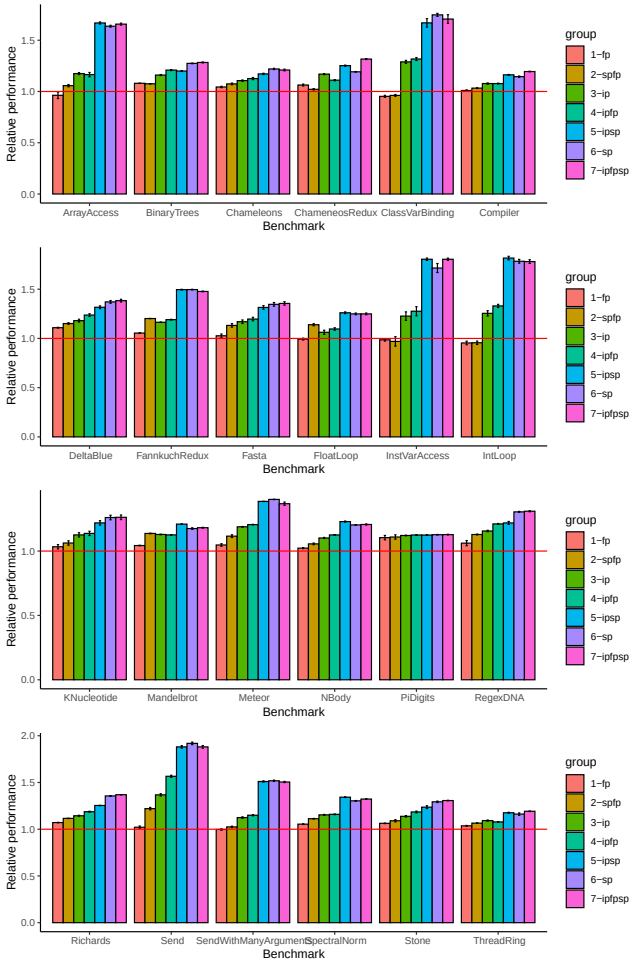

Figure 6: Average performance on different configurations. Error bars show standard deviation. Results are relative to no localization (1x, no bar). Higher is better.

4.2 Localization Improvements

We evaluate the performance of a large set of benchmarks on different interpreter configurations, localizing different combinations of the main interpreter registers: IP, SP, FP. Our benchmark suite is implemented in the SMark benchmark suite and the benchmark game suite for Pharo. We run each benchmark 100 times given that the interpreter does not need a warmup. We report averages and standard deviation. We use as comparison baseline (1x) the interpreter built without localization *i.e.,* using always global interpreter register variables.

Figure 6 presents the results of each configuration relative to our baseline on the Intel x86-64 architecture. Except the first two configurations in the graph: FP and SP+FP localization, all others autolocalized configurations are faster than non-localized version. We observe that localizing all IP, FP and SP, has the better improvement average of 1.39x. The best performance improvement is 1.92x when only localizing SP in the message send benchmark. In the worst case, we observe a short performance loss only when localizing FP and accessing class variables with a relative performance of 0.95x. One interesting result we observe that localizing only the SP is in average better than localizing the IP and SP by 0.1, and that localizing IP and SP is in average better than localizing all three register variables by 0.2.

4.3 Manual vs Automatic Localisation

We evaluate the performance of the manual localizing against our automatic approach on the same set of benchmarks, both localizing the three main interpreter registers: IP, SP, FP. We report averages of 100 iterations and their standard deviation. We use as comparison baseline (1x) the interpreter built manual localization. Figure 7 shows our results. The average difference between both configurations is 0.014x.

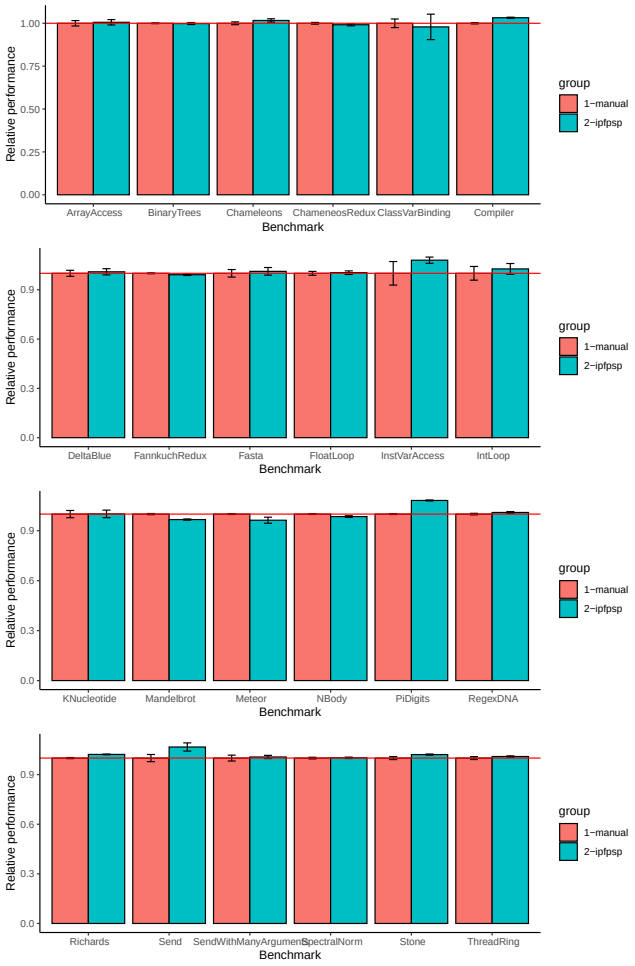

Figure 7: Manual vs Automatic Localization of IP, FP and SP. Average of 100 iterations, error bars show standard deviation. Results are relative to no localization. Higher is better.

5 CONCLUSION AND FUTURE WORK

In this paper we revisit *interpreter registers i.e.,* interpreter variables that are critical to the efficient execution of the interpreter loop. We study the impact of caching different interpreter registers configurations and present an automatic interpreter transformation that aims to store interpreter registers in machine registers.

We built our optimization in Slang, Pharo's VM generator framework, and show preliminary results showing that this transformation improves the performance of several benchmarks up to 1.92x on x86-64. We observed that localizing only the stack pointer (not the stack-top as proposed by Ertl [5]) shows similar improvements than localizing all of instruction, stack and frame pointers altogether, and that all our benchmarks improve in average 1.39x in comparison with the absence of this optimization. Moreover, we observed that our automatic approach has similar performance than the previous hand-tuned approach while keeping the code base simpler to modify and debug.

In the future we plan to study what are the architectural conditions that make these optimizations worth the effort and the behavior of our outliers. We plan to perform similar analyses on different architectures such as less powerful devices such as the aarch64 Raspberry pi and register starved architectures such as x86. On the design front, we plan to extend this work to automatically detect an optimal list of localization candidates for an architecture, and evaluate further optimizations on the synchronization on exit points to avoid redundant reads/writes. Finally, this automatic transformation opens the door to re-explore techniques such as top-of-stack caching without major VM and interpreter rewrites.

REFERENCES

[1] M. Berndl, B. Vitale, M. Zaleski, and A. Brown. Context threading: a flexible and efficient dispatch technique for virtual machine interpreters. In *International Symposium on Code Generation and Optimization*, pages 15–26, 2005.

[2] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[3] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 429–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *J. Instruction-Level Parallelism*, 5, Nov. 2003.

[5] M. A. Ertl. Stack caching for interpreters. *SIGPLAN Not.*, 30(6):315–327, jun 1995.

[6] M. A. Ertl and D. Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 278–288, 2003.

[7] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen-a generator of efficient virtual machine interpreters. *Software: Practice and Experience*, 32(3):265–294, 2002.

[8] I. Gouy and F. Brent. The Computer Language Benchmarks Game, 2004. http://benchmarksgame.alioth.debian.org/.

[9] D. Gregg, A. Beatty, K. Casey, B. Davis, and A. Nisbet. The case for virtual register machines. *Science of Computer Programming*, 57(3):319–338, 2005.

[10] D. Gregg and M. Ertl. A language and tool for generating efficient virtual machine interpreters. In *Domain-Specific Program Generation '03*, pages 196–215, jan 2003.

[11] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming (ECOOP'91)*, 1991.

[12] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications conference (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[13] E. Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

[14] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. Association for Computing Machinery.

[15] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 322–332, New York, NY, USA, 1995. Association for Computing Machinery.

[16] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4), jan 2008.

[17] Smark, benchmarking suite for smalltalk – consulted on 26 february 2022. https://github.com/smarr/SMark.

[18] T. Ugawa, H. Iwasaki, and T. Kataoka. eJSTK: Building javascript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages*, 51:261–279, 2019.

[19] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.