The Performance Effects of Virtual-Machine Instruction Pointer Updates

M. Anton Ertl
TU Wien, Austria

Bernd Paysan
net2o, Munich, Germany

Abstract

How much performance do VM instruction-pointer (IP) updates cost and how much benefit do we get from optimizing them away? Two decades ago it had little effect on the hardware of the day, but on recent hardware the dependence chain of IP updates can become the critical path on processors with out-of-order execution. In particular, this happens if the VM instructions are light-weight and the application programs are loop-dominated. The present work presents several ways of reducing or eliminating the dependence chains from IP updates, either by breaking the dependence chains with the *l*oop optimization or by reducing the number of IP updates (the *c* and *ci* optimizations) or their latency (the *b* optimization). Some benchmarks see speedups from these optimizations by factors > 2 on most recent cores, while other benchmarks and older cores see more modest results, often in the speedup ranges 1.1–1.3.

2012 ACM Subject Classification Software and its engineering → Virtual machines; Computer systems organization → Superscalar architectures; Software and its engineering → Interpreters

Keywords and phrases virtual machine, interpreter, out-of-order execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.14

Supplementary Material *Collection (Source Code, Binaries, Data)*: `https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz` [10]
Software (Source Code): `https://git.savannah.gnu.org/cgit/gforth.git` [9]
archived at `swh:1:dir:61eb3b71325060fe2e01f5e819eb0bec959e5bf0`

1 Introduction

Interpreters are a popular approach for implementing programming languages. Their benefits are simplicity of implementation, portability, and fast edit-run cycles. While they cannot compete in execution performance with JIT compilers or ahead-of-time compilers, a fast interpreter is not that far away: e.g., with the IP update optimizations of the present work, Gforth has similar performance to the SwiftForth JIT compiler and to gcc -O0 (see Section 6).

This paper uses Gforth as an example high-performance interpreter. Gforth implements a virtual machine (VM) and uses several previously published techniques for achieving high performance (see Section 2), most notably dynamic superinstructions (aka selective inlining) with replication and stack caching.

At the start of this work, every VM instruction in Gforth performed a VM instruction-pointer (IP) update [3]. It turns out that these IP updates (both the increments for ordinary instructions and the loads for taken branches) form a critical dependence path that limits the execution performance of many programs on modern processors.

We introduce a collection of optimizations for reducing these dependences: The loop optimization (*l*) breaks dependency chains in *l*oops (Section 4.1). Optimization *c* *c*ombines the IP updates of VM instructions that do not need an up-to-date IP (Section 4.2); the immediate optimization (*i*) avoids the need for an up-to-date IP for VM instructions with *i*mmediate operands (Section 4.3); The branch optimization (*b*) optimizes VM *b*ranches by replacing loads with (lower-latency) adds (Section 4.4).

Section 2 explains the interpreter performance techniques necessary to understand the present work. Section 3 explains how data dependences influence the performance of modern processors. Section 4 describes the optimizations and shows an example of their application; the novel *l*oop (Section 4.1), *i*mmediate (Section 4.3), and *b*ranch (Section 4.4) optimizations are among the main contributions of this work. Section 5 describes the measurement setup. The other main contribution of this work is in the empirical evaluation of the optimizations (Section 6). Finally, we discuss the applicability to other languages (Section 7), how to get the source code (Section 8) and related work (Section 9).

1.1 Why Gforth? Is this paper relevant for other languages?

You may wonder why we use Gforth and whether our results are relevant for other languages and their VMs.

We chose Gforth in the present work because it already implemented a number of techniques for increasing performance, in particular dynamic superinstructions and stack caching. As a result, Gforth's VM executes so few real-machine instructions per VM instruction that the dependences formed from IP updates become a bottleneck on certain programs.

We think that our IP update optimizations are also applicable to other VMs, but it depends on the VM, its implementation, and the characteristics of programs that are run on it how big the benefits will be. For a longer discussion, see Section 7.

2 Interpreter performance techniques

This section provides an overview of the performance techniques as far as necessary for understanding the IP update optimization, with literature references.

2.1 Virtual machines

Most interpreted programming language implementations compile the source code with a simple compiler into an intermediate code that represents the source program as a sequence of instructions of a virtual machine (VM) that is designed as both an easy target for the compiler and for easy (and ideally efficient) interpreted implementation of this code. Some well-known virtual machines, such as the Java Virtual Machine [15] and WebAssembly [12] also serve as program interchange formats, but in the present paper we focus on the role of virtual machines for execution in fast interpreters.

For our running example, we use Gforth's VM. Gforth is an implementation of the programming language Forth, a low-level (address arithmetic etc.) stack-based programming language.

Our running example is the inner loop of the siev benchmark:

```
do
  0 i c!
  dup +loop
```

We look only at the body of the loop, i.e., without the `do`. In Gforth's VM, the body looks as follows:

```
loophead: lit
          0
          i
          c!
          dup
          (+loop)
          loophead
```

Each line occupies one machine word, and *slanted blue* lines are immediate operands of the preceding VM instruction.

An interpreter for VM code keeps a pointer to the current VM instruction (the IP) around and uses it for finding immediate operands of the VM instruction and for finding the next VM instruction. In case of a VM-level direct branch instruction like `(+loop)`, the immediate operand is the branch target and if the branch is taken, the IP is set to the value of the immediate operand.

That's all you need to understand the optimization in the paper in the abstract, but to round out the picture, the rest of this section describes what these VM instructions do.

This Forth code corresponds to the following C code:

```
do {
  *p = 0;
  p += prime;
} while (p<pend)
```

Gforth's VM is stack-based and is relatively close to the Forth source code, with the following exceptions: Gforth compiles the number `0` to the VM instruction `lit` with the immediate operand *0*, and it compiles `+loop` to the VM instruction `(+loop)` with an immediate operand: the address of the VM instruction that `(+loop)` jumps to unless it exits the loop.

`lit` pushes its immediate operand on the data stack (or stack, for short). `i` pushes the current counter of the `do...+loop` counted loop on the stack (in this loop the counter contains the address corresponding to `p` in the C fragment). `c!` (pronounced "c-store") stores the second item on the stack to the byte pointed to by the address on the top-of-stack (TOS), popping both stack items. `dup` pushes another copy of the current top-of-stack value on the stack; this value corresponds to `prime` in the C program.

`(+loop)` pops the top-of-stack and adds it to the loop counter and checks for loop termination.[1] If another iteration is merited, `(+loop)` performs a VM-level jump to *`loophead`*.

2.2 Switch dispatch

A common way to implement an interpreter in C is to use a big switch statement along the lines of:

[1] As you will see in the assembly code later, this check is more complex than one would expect from the C code. The reason is that `+loop` is specified to support circular arithmetic and both positive and negative increments, which complicates the termination condition. For details see `https://forth-standard.org/standard/core/PlusLOOP`.

```
for (;;) {
  switch (*ip) {
    case dup: dsp[0] = tos; dsp--; ip++; break;
    case lit: dsp[0] = tos; dsp--; tos = ip[1]; ip+=2; break;
    ....
  }
}
```

In this example the data stack is represented by having the top-of-stack in a local variable `tos`, and the remainder of the data stack is in memory, and the local variable `dsp` points to where the top-of-stack would reside if it were in memory. IP is also kept in a local variable `ip`. We will use the same names for registers in assembly code shown below.

This scheme has a relatively high overhead of getting from one VM instruction implementation to the next. For `lit` with switch dispatch `gcc -O2` produces the following code for RISC-V (the destination register (if any) is leftmost):

```
.L2:                        #switch code
        ld      a4,0(ip)    # a4=*ip
        slli    a5,a4,2     # a5=a4*4  #for indexing
        add     a5,a6,a5    # a5=a6+a5 #table start in a6
        bgtu    a4,a7,.L17  # if a4>a7 goto default #bounds check
        lw      a2,0(a5)    # a2 = *a5 #load from table
        jr      a2          # indirect branch to a2

.L6:                        #lit code
        sd      tos,0(dsp)  # dsp[0] = tos
        ld      tos,8(ip)   # tos = ip[1]
        addi    dsp,dsp,-8  # dsp--
        addi    ip,ip,16    # ip += 2
        j       .L2         # back to switch code
```

Figure 1 shows the data structures involved in switch dispatch. The VM instructions are represented as integers that are used as indexes into the switch table. We use 8-byte VM-code slots for the code above.

The *payload* consists of only 3 RISC-V instructions in this case, whereas the dispatch overhead is 8 instructions.

Gforth has never implemented switch dispatch, and instead went directly for threaded code.

2.3 Threaded code

Threaded code [1] reduces the dispatch overhead by representing each VM instruction directly as the address of the machine code that implements it. This means that each instruction occupies one machine word (8 bytes on a 64-bit machine) and immediate operands are usually represented by one or more machine words. This concept results in the following code for `lit`:

```
sd      tos,0(dsp) # dsp[0] = tos
ld      tos,0(ip)  # tos = ip[0]
addi    dsp,dsp,-8 # dsp--
addi    ip,ip,16   # ip += 2
ld      a4,-8(ip)  # a4 = ip[-1] #address of next VM inst
jr      a4         # jump to next VM inst
```

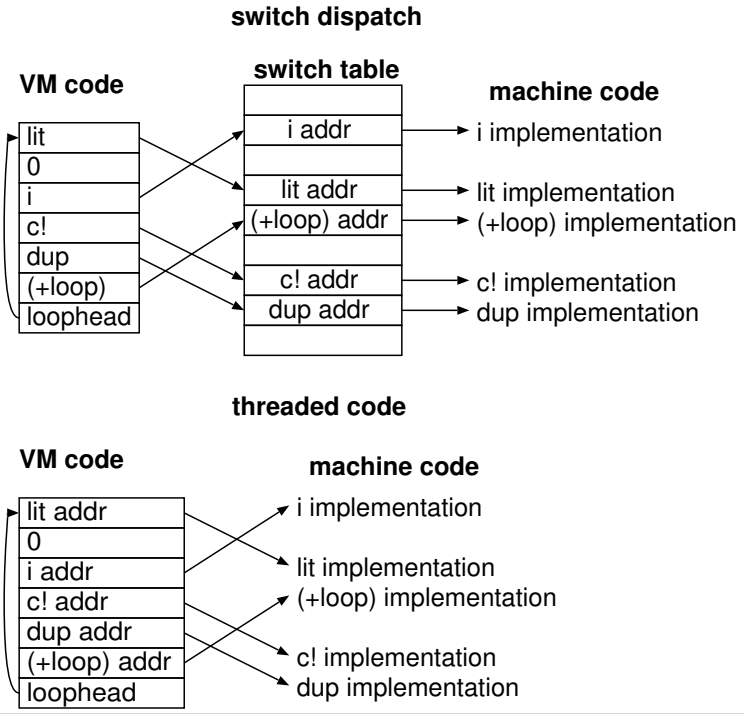

Figure 1 Switch dispatch vs. threaded code.

The dispatch code is inlined here and consists of 3 RISC-V instructions.

Figure 1 shows how the two schemes get from the VM code to the corresponding machine code. In both schemes `ip` points to the VM code, and immediate operands are accessed through `ip`. VM control flow is performed by setting `ip` to something other than the next VM instruction.

One practical consideration is how to implement threaded code in an architecture independent language. Fortunately it is possible by using the labels-as-value extension of GNU C, which has been implemented by at least gcc, clang, tcc, and icc.

2.4 Selective inlining and dynamic superinstructions

One can eliminate more of the dispatch: While generating VM code, copy (real-)machine code snippets from the interpreter to a separate memory area, thus concatenating these snippets (Fig. 2); the threaded-code addresses then point to this newly generated real-machine code rather than the originals as in normal threaded code.

This technique has first been outlined as *`memcpy` method* by Rossi and Sivalingham [20], and later explored in depth as *selective inlining* by Piumarta and Riccardi [17]. Ertl and Gregg combined it with replication [4] for better branch prediction. They call the result of the concatenation *dynamic superinstructions*, because, like static superinstructions [18, 8, 2] they combine a sequence of n VM instructions with n dispatches into something with only one dispatch.

In our example (Fig. 2), each VM instruction except the last one (`(+loop)`, which is a VM-level branch) just continues with the next one, so the machine code of the next one can be concatenated to the machine code of the dynamic superinstruction. The `(+loop)` may

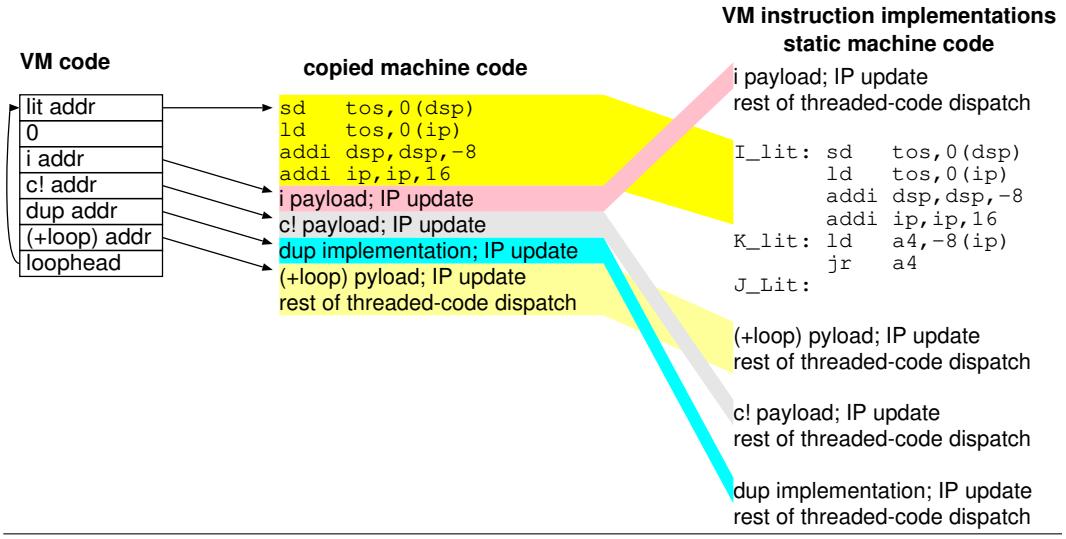

Figure 2 Concatenating machine-code snippets to further reduce the dispatch overhead.

set `ip` to something other than the next instruction, so for `(+loop)` and other control-flow VM instructions the whole code including the rest of a threaded-code dispatch is appended in order for the control-flow change to take effect at run-time; such an instruction therefore ends a dynamic superinstruction.

Gforth copies the machine code snippets at the same time as when it generates VM code for newly compiled source code. Gforth does not save the machine code in its images, so its image loader copies the machine code snippets for the VM code it loads.

This technique has provided a big performance boost to Gforth across many different CPUs, typically by a factor of 2 over threaded code (see Fig. 10). One may balk at the prospect of directly manipulating machine code, but the advantage of starting with an interpreter is that Gforth can always fall back to threaded code if conditions seem adverse (and this normally works automatically).

One may wonder if the result is not already a JIT compiler, and in certain respects it is. But for the language implementor it is an extension of a threaded-code interpreter: Each implementation of a VM instruction just gets labels before and after the "rest of threaded-code dispatch" part, and when a VM instruction is generated, it also copies the memory containing the machine code for the VM instruction (using the labels to know the boundaries), and lets the threaded-code word point to the copy (instead of the original). The only amount of machine-specific code are a few lines to synchronize the I-cache to the D-cache, and GNU C provides `__builtin___clear_cache` for that purpose. And when the conditions for dynamic code generation are not met, the system just falls back to plain threaded code, overall or on a per-VM-instruction basis (e.g., for code that contains a relative reference to an address outside the code snippet at hand). By contrast, a typical JIT compiler needs much more machine-specific work.

2.5 Multi-representation stack caching

The Gforth baseline also uses an optimization called multi-representation stack caching. This optimization reduces only the machine instructions in the payload, so you only need to read this section if you want to understand the payload of our running example, too.

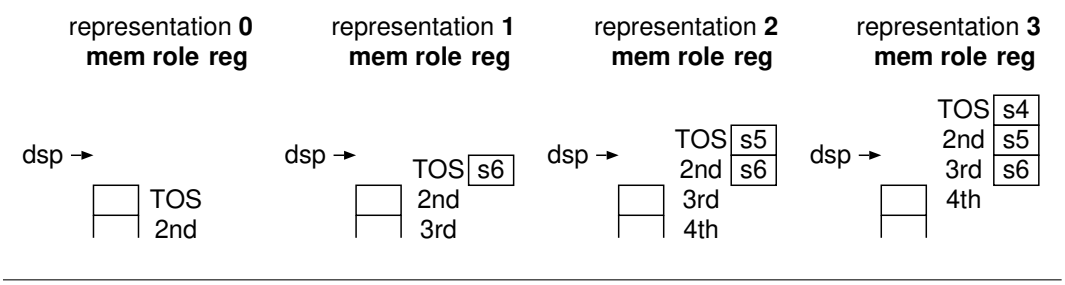

Figure 3 Four data-stack representations used by Gforth on RISC-V.

Figure 3 shows different representations of the data stack. Representation 0 keeps 0 stack items in registers, i.e., all stack items in memory. A representation with all stack items in memory is often seen in the literature (usually with the stack pointer pointing to the top-of-stack, but that is just a difference in the offsets used for the memory accesses).

The examples shown earlier use representation 1, and this is also used by Gforth when it falls back to threaded code. The advantage of this representation can be seen for `dup` which does one load and two stores with representation 0, but just one store with representation 1.

By switching between representations Gforth further reduces the stack handling effort. E.g., our running example starts in representation 1 (Gforth always uses this at the start of a basic block) with the VM instruction `lit`. By choosing the `lit` implementation that ends in representation 2 (i.e., `lit` $1 \rightarrow 2$), the payload of `lit` in this case is reduced to

```
ld  s5,8(ip)
```

The old top-of-stack stays in s6 (and becomes the 2nd stack element), and the new top-of-stack is pushed by setting s5 (the new top-of-stack) to the immediate operand.

This eliminates a memory access to the data stack as well as an update of `dsp`. If you take a closer look at Fig. 4, you do not find any memory access to the data stack nor any data-stack pointer update, so in this case data-stack caching works perfectly.

Ertl and Gregg [5, 7] discuss multi-representation stack caching in more detail.

3 Understanding performance

This section describes how program characteristics influence the performance on processors with out-of-order (OoO) execution, and in particular, it discusses the role of instruction pointer updates in interpreters with dynamic superinstructions. OoO processors have dominated general-purpose computers in this century, and are now advancing towards smaller systems. E.g., the Raspberry Pi switched to OoO cores with the Raspberry Pi 4 in 2019 and the Compute Module 4 in 2020.

3.1 ... on modern CPUs ...

Starting from an empty pipeline, the front end of an OoO processor fetches and decodes instructions as directed by the branch predictor, possibly running far ahead of execution. An instruction is then executed as soon as all its inputs are available and an appropriate functional unit is available.

If a branch is mispredicted, fetch, decode, and execution at first continue along the (mis-)predicted path, but the results are not committed. When the correct direction or branch target is determined by executing the appropriate conditional or indirect branch instruction, the front end is redirected to fetch, decode and eventually execute from the correct path.

This description indicates the ways in which program characteristics influence performance:

As long as mispredictions are rare, if there are enough independent instructions, execution will be limited by the resources, either by the program needing too many of a particular functional unit (e.g., a matrix multiply program will exercise load and store units and the FP multiply-add a lot), or by the width of the decoder and/or the retirement unit.

On the other hand, if there are lots of dependences between instructions and the processor offers enough resources, the dependences will determine the performance: an instruction that depends on another instruction i on the critical dependence path will wait in the processors's buffers until i produces a result. After prefetching for a while, all these not-yet ready instructions will fill the processor's buffers and the processor's front end has to wait until more buffers become ready by finishing an instruction on the critical path.

In the branch misprediction case, the misprediction penalty is influenced by the kind of dependences between instructions: If there is a short dependence path to the predicted branch instruction, the misprediction can be resolved early. However, if the mispredicted branch depends on an instruction in the critical dependence path, the misprediction will not be discovered until the instructions leading to the branch have been executed; only then can the correct path be fetched and decoded, so such a misprediction incurs a bigger misprediction penalty. By contrast, in case of a correct prediction, the long latency until the prediction is confirmed does not hurt, except for occupying some buffers for longer.

3.2 ... in fast interpreters

In an interpreter, there is the resource consumption and dependences inherent in the interpreted program (i.e., also present if the program is compiled to real-machine code), but there is also the overhead of the interpreter:

In particular, every VM interpreter updates the VM instruction pointer (IP), in order to access immediate VM data through it, and to access the next VM instruction (or next dynamic superinstruction). In straight-line code, this results in one addition per executed VM instruction, with a latency of one cycle on most processors. For a VM-level absolute branch (as used in Gforth), the new VM instruction pointer has to be loaded, with a latency of 3–5 cycles on recent OoO processors; if the VM-level branch is relative, the loaded value has to be added to the instruction pointer, costing an additional cycle.

A VM-level return instruction breaks the IP dependence chain of the callee, because it loads the new VM instruction pointer from the saved return address. This continues the dependence chain of the caller, but the callee's chain of IP updates ends with the return.

VM interpreters also have other overheads: A stack-based VM like that of Gforth has dependence chains through stack-pointer updates, and these dependence chains are not broken by returns. Moreover, they keep most stack items in memory with the resulting store-to-load latency: 4-7 cycles in many processors, but 0 cycles in several recent processors like Zen 3 and Tiger Lake.[2] However, stack caching (see Section 2.5) reduces these overheads substantially.

[2] `https://www.complang.tuwien.ac.at/anton/memdep/`

For a register-based VM, the VM register accesses are usually implemented through real-machine memory accesses, which increases the resource consumption substantially. On older processors there is also the latency cost of store-to-load forwarding, but the significance of this cost depends on the dependence patterns of the interpreted program.

Previous work did not consider VM instruction-pointer updates to have much effect. Ertl and Gregg [4] wrote:

> One thing that we have not implemented is eliminating the increments of the VM instruction pointers along with the rest of the instruction dispatch in dynamic superinstructions. However, by using static superinstructions in addition dynamic superinstructions and replication we also reduce these increments (in addition to other optimizations); looking at the results from that, eliminating only the increments probably does not have much effect.

For a long time our thinking was that other dependencies would dominate over VM instruction-pointer updates, and that, with processors becoming wider (being able to execute more instructions per cycle), instruction-pointer updates would become even less relevant. However, for a number of benchmarks this is wrong (see Section 6).

4 Instruction-pointer update optimization

This section discusses four mostly independent optimizations. We implemented these optimizations in Gforth, and discuss them in this context, but they can also be applied to implementations of other languages.

4.1 Loops

This optimization breaks the IP dependence chains on loop-back edges. In typical VM instruction sets, the loop-back branch takes the target address as an immediate operand (e.g., in Fig. 4 the immediate operand `loophead` following the VM branch instruction `(+loop)`).

With the *l*oop optimization, the loop-back address is stored on the return stack on entering the loop, and the loop-back branch then takes its address from there (the **bold green** instruction in Fig. 4). Because it does not need to access the VM instruction pointer to do that, this breaks the dependence chain.

In Forth the return stack is a stack that contains return addresses and counted-loop parameters. In general, the loop-back address can be stored on any stack or in a VM register; the important part for the *l*oop optimization is that this address must be readable by the loop-back instruction without requiring an IP access, so one cannot use a VM register whose number is given as immediate operand.

Unfortunately, the design of Forth makes it difficult to apply this optimization to general loops, so we only apply it to counted loops in the present work.

However, if you are designing a virtual machine for a programming language, it may be worthwhile to design it in a way that makes it possible to store the loop-back address somewhere on entry to the loop, and to load it from there on the loop-back branch without accessing the IP.

The loop optimization has very little effect on the instruction count and other dependences, and can therefore be used to see the performance effect of breaking the IP dependence chains independent of, e.g., the effect of reducing the number of executed instructions. In Fig. 9 we see speedups by a factor of 2 on some benchmarks, showing that the IP-update dependence chain really is the bottleneck for these benchmarks.

VM code		*unoptimized*	*l*	*c*	*ci*	*cib*
lit	$1 \rightarrow 2$	*addi ip,ip,16*	*addi ip,ip,16*	*addi ip,ip,16*		
0		ld s5,-8(ip)	ld s5,-8(ip)	ld s5,-8(ip)	ld s5,8(ip)	ld s5,8(ip)
i	$2 \rightarrow 3$	*addi ip,ip,8*	*addi ip,ip,8*			
		ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)
c!	$3 \rightarrow 1$	*addi ip,ip,8*	*addi ip,ip,8*			
		sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)
dup	$1 \rightarrow 2$	*addi ip,ip,8*	*addi ip,ip,8*			
		mv s5,s6	mv s5,s6	mv s5,s6	mv s5,s6	mv s5,s6
(+loop)	$2 \rightarrow 1$	*addi ip,ip,16*	*addi ip,ip,8*	*addi ip,ip,40*	*addi ip,ip,56*	
loophead		ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)
		ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)
		ld a2,-8(ip)		*ld a2,-8(ip)*	*ld a2,-8(ip)*	
		add a3,s5,a5	add a3,s5,a5	add a3,s5,a5	add a3,s5,a5	add a3,s5,a5
		sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4
		add a4,s5,a5	add a4,s5,a5	add a4,s5,a5	add a4,s5,a5	add a4,s5,a5
		xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5
		xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5
		and a4,a5,a4	and a4,a5,a4	and a4,a5,a4	and a4,a5,a4	and a4,a5,a4
		sd a3,0(rp)	blt a5,zero,x	sd a3,0(rp)	sd a3,0(rp)	sd a3,0(rp)
		blt a5,zero,x	**ld ip,16(rp)**	blt a5,zero,x	blt a5,zero,x	blt a5,zero,x
		ld a5,0(a2)	sd a3,0(rp)	ld a5,0(a2)	ld a5,0(a2)	ld a5,0(ip)
		mv ip,a2	ld a5,0(ip)	*mv ip,a2*	*mv ip,a2*	
		jr a5	jr a5	jr a5	jr a5	jr a5
		x:	x:	x:	x:	x:

Figure 4 The inner loop of the benchmark `siev` in Gforth's VM code, and the corresponding RISC-V code produced by Gforth without optimization and with various IP-update optimizations: *l* optimizes loops, *c* combines IP updates, *i* optimizes immediate operands, *b* optimizes branches. $1 \rightarrow 2$ etc. indicates a stack representation change (see Section 2.5). For instructions with destination registers, the destination is leftmost. The instruction that starts a new IP dependence chain (in the loop) is **bold green**. Instructions that continue IP update dependence chains are *slanted red*. Some register names have been changed for ease of understanding: ip is the VM instruction pointer, rp is the return-stack pointer. s6, s5, s4 contain stack elements (see Fig. 3).

While it is possible to combine this optimization with the others, we think that the combination of the others is effective enough in reducing the IP dependence chain, and that adding the *l*oop optimization would not help once the other optimizations are performed. However, we consider the *l*oop optimization to be an alternative that requires less effort.

You can see the result in column *l* of Fig. 4. The decisive difference is that the *ld a2,-8(ip)* in `(+loop)` in *unoptimized, c, ci* loads the branch target from VM code using the IP, while the **ld ip,16(rp)** loads the branch target from the return stack (using rp).

We implemented a prototype of this optimization in Gforth by adding 89 lines.

4.2 Combining instruction-pointer updates

There are VM instructions where the payload of the implementation does not read the IP and therefore does not need an up-to-date IP. In our running example `i`, `c!` and `dup` do not need an up-to-date IP.

Therefore the IP update can be left away. When there is finally a reason for an up-to-date IP, all the updates can be combined into one addition of a larger constant.

Columns *unoptimized*[3] and *c* of Fig. 4 illustrate this. In *unoptimized* every VM instruction has its own IP update; in *c*, `lit` has an IP update, because it loads its immediate operand `0` in the VM code through `ip`. The next three VM instructions `i`, `c!` and `dup` don't need an up-to-date IP, so *c* eliminates their IP updates. Finally, `(+loop)` needs an up-to-date IP in order to load its immediate operand *loophead* (the loop-back address) from the VM code, so Gforth's compiler inserts an IP update by 40 covering all VM instructions `i`...`(+loop)` (inclusive), the same as the sum of the corresponding IP updates in *unoptimized*.

The optimization itself is trivial: The code generator just keeps track of where IP actually points to, and when an up-to-date IP is needed, it inserts the appropriate update.

One not quite trivial part, however, is: When is an up-to-date IP needed?

Superblock end: The next VM instruction is the target of a VM jump. Because the IP may be used afterwards, we have to synchronize the IPs coming from different paths at this point, and we do it by letting it point to the first VM instruction in the new superblock.

Calls: VM instructions like `call` and `execute` (an indirect call) also require an up-to-date IP: calls save the IP (which points to the next instruction at that point) as return address, and after returning execution continues at that address. The routine invoked by `execute` finishes with a threaded-code dispatch, which needs an up-to-date IP.

Non-relocatable VM instruction: When the machine code for a VM instruction is not relocatable (typically because there is a call to a C function in the machine code), this code cannot be used in a dynamic superinstruction. Instead, this code is called through a threaded-code dispatch (which uses IP), and this code then updates the IP and makes another threaded-code dispatch for continuing execution after this VM instruction.

Immediate operands: The IP is used when accessing immediate operands of VM instructions.

One particular case of this optimization is VM instructions like Gforth's `;s` which returns from a definition. It does not need an up-to-date IP beforehand, and it branches elsewhere (returning to the caller at the VM level), so there is no need to update the IP afterwards, and we suppress such an update.

The other not quite trivial part is how to generate the machine code in the dynamic superinstruction framework for which the actual machine code that is copied around is just an opaque code snippet.

The first question is how to separate the IP update that is part of every VM instruction implementation (as part of the threaded-code dispatch) so that we can copy the machine code without the IP update.

Because the IP needs to be up-to-date in front of some VM instructions, we put the IP update at the start of each VM-instruction implementation, resulting in the following template:

```
I_inst:
  update ip
L_inst:
  non-dispatch code
K_inst:
  rest of dispatch
J_inst:
```

[3] This column shows the code for a Gforth version that includes ip-update optimizations but has them disabled; this means that it uses the same register allocation and instruction schedule as the various optimized variants, which makes it easy to compare with the other columns.

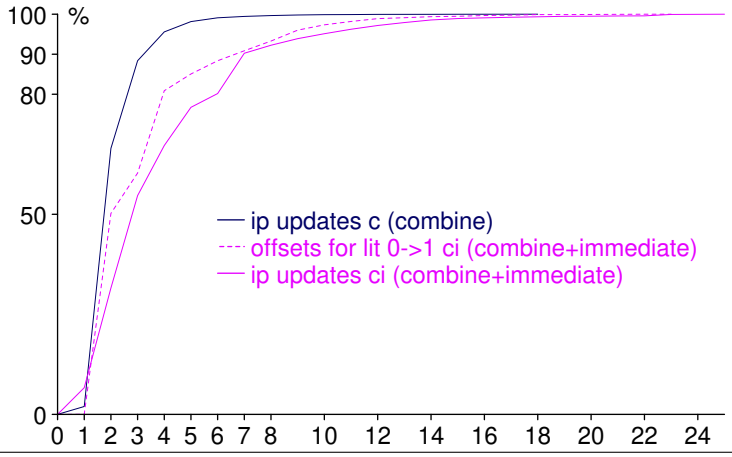

Figure 5 Proportion of IP updates or `lit 0->1` offsets with distances less than a given number of machine words in Gforth's image (static counts).

If the resulting code is relocatable and code for the VM instruction *inst* should be generated, the code generator first generates an appropriate IP update if necessary (but normally does not use the update between `I_inst` and `L_inst` for that). Then it copies the code between `L_inst` and `K_inst`.

The code generator needs code snippets for different amounts of IP updating, because it cannot just patch a constant into a template for IP-updating (the code generator does not know anything about the internal structure of the machine code). Instead, we added code snippets for IP updates for a range of values (by 1–23 machine words) to the C source code of Gforth, and the code generator selects the right one, or (for IP updates > 23 words) generates a sequence of IP updates.

Figure 5 shows that if we have IP update code snippets for updates by 1–6 machine words, 99% of the cases statically occuring in the Gforth image can be performed with one instruction, so for Gforth limiting the IP upates to this range would be good enough as long as VM instructions with immediate operands (Section 4.3) are not optimized as well.

These data are somewhat specific to the Gforth VM, so if you want to minimize the number of IP update code snippets, you should do your own measurements. As Fig. 5 shows, a major reason for IP updates is VM instructions with immediate operands. VMs that use local-variable accesses more than Gforth and specify the local with an immediate operand will have shorter sequences between IP updates, which makes *c* alone less beneficial, but means that even fewer IP update code snippets cover nearly all occuring distances with one IP update.

4.3 Immediate operands

VM instructions with immediate operands are relatively frequent. We can eliminate this reason for requiring an up-to-date IP in most cases: We introduce additional variants of the most frequent VM instructions with immediate operands.[4] These additional variants access their immediate operand at an offset (1–23 machine words in our experimental implementation) from where their base variant accesses the immediate operand, thus allowing the actual IP to point 1–23 machine words in front of to up-to-date IP.

[4] `lit`, `call`, `?branch`, `lit@`, `branch`, `(loop)`, `lit-perform`, `lit+`, `does-xt`

When the code generator has to compile such a VM instruction, if the difference between the actual and the up-to-date IP is within the offset range of the variants, the code generator copies the code of the appropriate variant, and no IP update needs to be generated.

In Fig. 4, column *ci* shows how combining IP updates is enhanced by this immediate-operand optimization: The first VM instruction is `lit`, and in *c* it needs an IP update; in *ci*, a variant of `lit` that accesses its immediate operand at `ip+8` instead of `ip-8` (an offset of 2 machine words) is used, so there is no need to update IP.

However, `(+loop)` is a VM instruction that does not have such variants, so the code generator updates the IP at the start of `(+loop)`.

While it is not obvious from this example, this extension contributes a lot to the effectiveness of combining IP updates: In the Gforth image, the number of static IP updates is reduced by a factor of 5; the dynamic reduction in our benchmarks usually a factor < 2 compared to the reduction from *c* alone (see Fig. 8).

Figure 5 shows that IP updates by 1–16 machine words are sufficient for performing (without resorting to sequences of adds) 99% of the remaining IP updates statically occuring in the Gforth image. It also shows that for the most frequent VM instruction with a literal, `lit` in its stack caching variant $0 \rightarrow 1$, 99% of the IP offsets are in the range 2–13 (machine words).

The relevance of these numbers is as follows: The compilation time of the VM implementation increases with the number of VM instruction implementation variants, so we only want to add additional variants when a benefit is expected. This is particularly relevant for instructions with immediate operands, because there are a number of them, and stack caching multiplies the numbers.

E.g., we selected only 9 VM instructions with immediate operands; stack caching increases this to 15 variants, and having 24 subvariants with different offsets for each variant results in a total of 360 implementations of these 9 VM instructions. We did not use additional variants for other VM instructions with immediate operands (e.g., `(+loop)`) to avoid increasing the compilation time of the interpreter too much. For the same budget of 360 implementations, it might have been a little better to use a smaller offset range and to have offset-variants of more VM instructions.

Another way to deal with this problem is to eliminate immediate operands by introducing versions of VM instructions for specific immediate operands. E.g., Gforth has a general VM instruction `@local#` with an immediate operand *n* for pushing the value of local variable *n* onto the stack, but it also has `@local0`, which fetches the local variable 0 without needing an immediate operand. To increase the benefits from IP update optimization, we added more such variants to Gforth.[5]

These optimizations also shift the balance in VM design towards splitting one VM instruction into several, especially if it means that an unoptimized VM instruction with a literal operand can be replaced with an optimized one. E.g., we have replaced the general case of `@local#` *n* (i.e., cases not covered by specialized variants like `@local0`) in Gforth by the sequence `lit` *n*`; @localn` where `@localn` takes *n* from a register representing the top of the data stack (pushed there by `lit`). The resulting code is often better than for `@local#`:

[5] In the mainline, not in the variants used for the empirical results of the present work. The benchmarks used for the present work don't use local variables much, so we don't expect that this would make a significant difference.

unsplit			split		
@local#	0 → 1	*addi ip,ip,88*	lit	0 → 1	
64		ld a5,-8(ip)	*64*		ld s6,80(ip)
		add a5,a5,lp	@localn	1 → 1	add a5,s6,lp
		ld s6,0(a5)			ld s6,0(a5)

In this code lp is a register containing Gforth's locals pointer.

4.4 Branches

When executing VM instructions, every taken VM branch that loads the target address from the VM code (such as `(+loop)` in Fig. 4) performs an IP-dependent load, and thus extends the IP dependence chain with the load latency (3–5 cycles on modern processors). Even with the *ci* optimizations, these loads can mean that IP updates are still the critical dependence path in branch-heavy code like the `siev` benchmark.

However, branches are often to nearby targets, which inspires the following idea: If the target is nearby, set the IP to the target, and then execute a branch-to-IP variant of the branch; i.e., if the branch is taken, it just needs to perform a threaded-code dispatch to branch to where the IP currently points to. If the branch is not taken, execution just continues after the branch, taking the changed IP into account.

To implement this, we have extended the code snippets for updating the IP to increment the IP by -24–23 machine words. Only one branch-to-IP variant is needed for each branch, so we implemented this additional variant for all branches where the ordinary variant just takes the target address as immediate operand; there are branches in Gforth with an additional immediate operand, and we cannot apply this optimization to those branches; fortunately, they are rarely used.

You can see an example in column *cib* in Fig. 4. Thanks to the `ci` part of the optimization, there is no IP update for the `lit`, so when Gforth's code generator reaches the `(+loop)`, the actual IP is still at the start of the loop. The code generator determines that the target is nearby, and proceeds to insert an IP update for setting IP to the branch target. Because the actual IP already points to the target location, the IP update would be by 0 bytes, and no code is generated for that, an ideal outcome; in the general case you would see one or more IP update instructions at this point. Next, the code generator appends the code of the `(+loop)` variant for the *b*ranch optimization; note that this code does not contain the instructions *ld a2,-8(ip)* and *mv ip,a2* for modifying the IP; it expects that the IP already contains the right value for taking the branch.

In our experiments, we considered the target to be nearby, if it can be reached with one IP update for conditional forward branches, or if it can be reached with three IP updates for unconditional branches and conditional backwards branches. This assumes that backwards branches are usually taken, and also takes into consideration that on the fall-through path IP update for the branch might require a followup correction.

We did this for the following reasons: For unconditional branches, three IP updates have a smaller or the same latency as a load. In case of conditional branches, a backwards branch is a loop branch and therefore probably taken.

For the conditional forward branch, a classical rule-of-thumb says that not-taken is more likely. If we use the original branch instruction instead of the branch-to-IP variant, the not-taken path may work without IP update; with the branch-to-IP variant, we incur the IP update for setting the target in either case, and in the not-taken case we may need another IP update because of an instruction with an immediate operand in the code before the branch target. One could reduce the latter cost by introducing variants of instructions with immediate operands with negative offsets, but that also has its costs.

Another idea that we have not implemented (yet) is to have IP update variants with larger granularity. E.g. have IP update variants for −16, −15, −14, ..., 14, 15 machine words and then −272, ..., −80, −48, 47, 79, 111, ..., 271 machine words. This would allow to compose IP updates by −288...286 machine words by concatenating two code snippets (typically with one instruction each) using only 47 IP-update variants (the same number currently used in Gforth).

We do not present empirical data for branch distances, because they depend strongly on the programming language usage (large or small routines), the VM design (e.g, already the splitting of VM instructions discussed in Section 4.3 changes the distances), and on compiler features such as tail call optimization, inlining or jump-to-jump optimization. So you will have to do your own measurements to see the distribution of distances for your VM.

For Gforth, Fig. 9 shows speedups from *cb* over *c* or from *cib* over *ci* on most benchmarks (exceptions: brainless, cd16sim, sha512), so even the −24...23 machine-word range of IP-update variants provides some benefit for this VM.

We implemented *cib* in Gforth by inserting 864 lines and deleting 316 lines.

5 Evaluation setup

5.1 Systems

We present measurements for the versions described in Section 4. As *baseline* we use a Gforth version without any IP-update optimization work. We branched a variant from that that contains only the loop optimization, and a variant that contains all new optimizations developed in the present work, selectable individually (however, the loop optimization does not work with *cib* at the moment). The Gforth variants we measured are:

baseline The Gforth version we started from. This is the numerator in the factors shown in the speedup and instruction factor graphs. The variants/system for the specific bar is the denominator.

unoptimized The version that contains all optimizations developed in the present work, but with the optimizations turned off. While in the baseline the IP update of a VM instruction is anywhere in its code, the IP update is at the start in *unoptimized* (so the IP-update optimizations can eliminate it or replace it). We show this variant in some figures to see whether the code changes had some additional effect (and to isolate this effect, if any).

baseline+loop opt This variant adds VM instruction variants for the loop optimization (see Section 4.1) and uses these for counted loops instead of the variant that loads the branch target from the VM code.

unopt+loop opt This uses the same executable as the *unoptimized* variant, but for counted loops it uses the VM instructions that perform the loop optimization.

c: combine IP updates This uses the same executable as *unoptimized*, but enables combining IP updates (Section 4.2).

ci: c+immediate opt Like *c*, but also enables the optimization of VM instructions with immediate operands (Section 4.3).

cb: c+branch opt Like *c*, but also enables the optimization of short VM branches (Section 4.2).

cib: ci+branch opt Like *ci*, but also enables the optimization of short VM branches.

Program	Author	Description	Lines	Characteristics
bench-gc	Anton Ertl	Garbage Collector	1155	calls
brainless	David Kuehling	Chess	3648	calls, app
cd16sim	Brad Eckert	CPU emulator	937	calls, app
fcp	Ian Osgood	Chess	2046	calls, app
lexex	Gerry Jackson	Scanner Generator	3655	calls, app
siev	Gilbreath/Paysan	Count primes	25	counted loops
bubble	Hennessy/Fraeman	Sort	74	counted loops, cond. br.
matrix	Hennessy/Fraeman	Integer matrix multiply	57	counted loops
fib	Anton Ertl	Recursion	14	calls, cond. branch
fft-bench	Bernd Paysan	Fast Fourier transform	106	calls in counted loop
pentomino	Bruce Hoyt	Puzzle	516	conditional branches
sha512	Marcel Hendrix	Cryptography	538	counted loops, huge body

Figure 6 Benchmark programs used.

Evaluating *b* alone would also have been interesting, but we left it away for time and space reasons. However, you can see the effect of *b* by comparing the results of *c* with *cb* and of *ci* with *cib*.

In addition, for Fig. 10 we compare with the following systems/compilers.

PFE is an interpreted Forth system written in C that uses one C function per VM instruction implementation. PFE is designed to rely on explicit register allocation (a GCC extension) for performance, but unfortunately, for AMD64 no explicit register definitions have been added yet. We use PFE-0.33.71.

Gforth threaded code only This is the baseline Gforth with the option `--no-dynamic`, which means that it falls back to using plain threaded code (Section 2.3); this option also disables stack caching.

SwiftForth, VFX Forth Two commercial Forth systems with JIT compilers. We measured SwiftForth x64-Linux 4.0.0-RC87 and VFX Forth 64 5.43.

gcc-12 Various optimization options for GCC 12.2. Manually written C code for four of the benchmarks is available and was used for generating these results. The C programs were linked statically so that the binaries could also run on machines with older glibc implementations. For gcc the results do not include the compile time (unlike for the Forth systems).

We compiled the three Gforth branches with gcc-12.2 on Debian 12 for AMD64 and with gcc-10.2 on Debian 11 on ARM A64 and statically linked them so they would run on the other platforms we used. All variants use stack caching with 0–3 registers.

5.2 Benchmarks

We use the benchmarks shown in Fig. 6. The first five are from the appbench suite of Forth benchmarks[6]; they are substantial programs and therefore are probably more representative of significant Forth applications and idiomatic Forth usage than the other benchmarks.

The next five are small benchmarks that come with Gforth: **siev** is based on the Byte Sieve by Gilbreath, but we use Bernd Paysan's Forth version and Al Aburto's C version.

[6] `https://www.complang.tuwien.ac.at/forth/appbench-1.3.zip`

μArchitecture	Architecture	Family	CPU	year
K8	AMD64	AMD P	Athlon X2 4600+	2005
Zen3	AMD64	AMD P	Ryzen 7 5800X	2021
Penryn	AMD64	Intel P	Xeon E5460	2007
Nehalem	AMD64	Intel P	Xeon X3460	2009
Sandy Bridge	AMD64	Intel P	Xeon E3-1220	2011
Haswell	AMD64	Intel P	Core i7-4790K	2014
Skylake	AMD64	Intel P	Core i5-6600K	2015
Rocket Lake	AMD64	Intel P	Xeon W-1370P	2021
Tiger Lake	AMD64	Intel P	Core i5-1135G7	2021
Golden Cove	AMD64	Intel P	Core i3-1315U	2023
Silvermont	AMD64	Intel E	Celeron J1900	2013
Goldmont	AMD64	Intel E	Celeron J3455	2016
Goldmont+	AMD64	Intel E	Celeron J4105	2017
Tremont	AMD64	Intel E	Celeron N4500	2021
Gracemont	AMD64	Intel E	Core i3-1315U	2023
Firestorm	ARM A64	Apple P	M1	2020

Figure 7 Microarchitectures measured and shown in Section 6. The year shows when the CPU we measured was released. Some of the microarchitectures were released earlier in different CPUs. "P" stands for performance core, "E" for (power or die area) efficiency core.

bubble and **matrix** are based on Hennessy's Stanford integer benchmarks (in C), and have been translated to Forth by Marty Fraeman. Four of these benchmarks are available in Forth and C in `http://www.complang.tuwien.ac.at/forth/bench.zip`.

Pentomino and sha512 were included because they exhibit unusual performance characteristics (for Forth programs): They both spend much of their time in long definitions, with many branches for pentomino, and straight-line code for sha512.

Idiomatic Forth code calls many short routines, as exhibited in the appbench programs and in fft-bench. So the results for these programs may also be representative for other programming languages where call-heavy programs are idiomatic and implementations that neither inline nor tail-call-optimize. On the other hand, the results for the programs dominated by counted loops may be more representative for programs in Algol-family languages and for systems that tail-call optimize or inline.

5.3 Hardware

We have measured a variety of different microarchitectures and show results for them. Figure 7 gives information about what the code names we use for the microarchitectures mean.

5.4 Measurements

Each benchmark was run on each system and each microarchitecture 30 times, and measured with `perf stat`, measuring the events instructions:u, cycles:u, branch-misses:u, L1-dcache-load-misses:u, and L1-icache-load-misses:u, where available (but we only show results based on cycles and instructions here). The median of these runs is shown.

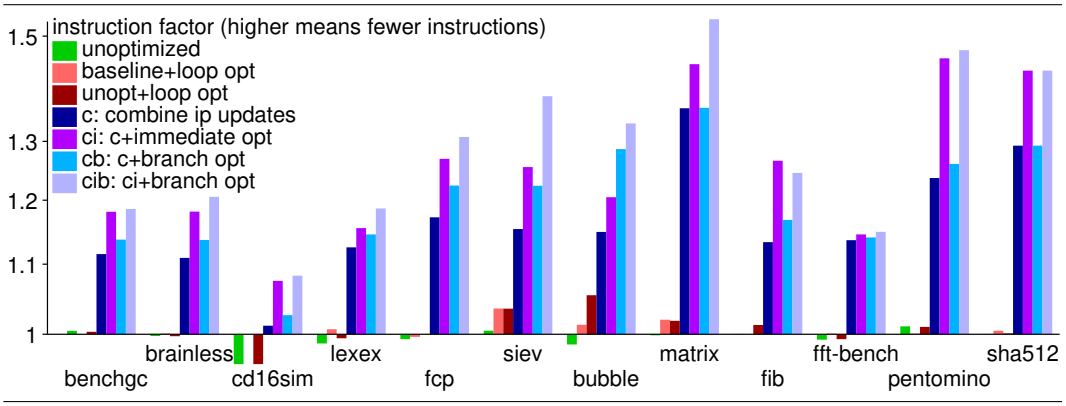

Figure 8 Reduction factor in the number of dynamically executed AMD64 instructions of various optimizations over *baseline*.

6 Results and discussion

6.1 Executed instructions

Figure 8 shows the effect of the IP update optimizations on the number of executed instructions on AMD64. For ARM A64 and RISC-V the results look similar.

For *unoptimized* and both loop optimization variants, the differences in executed instructions from the baseline are small, as expected (so small that sometimes you don't see the bar).

For most benchmarks *c* (combining IP updates) reduces the executed instructions, and *ci* (also optimize VM instructions with immediate operands) further reduces them (because more IP updates can be eliminated); adding *b* often has little effect on the number of executed instructions: in the usual case a load is replaced by an add.

On AMD64 and RISC-V where the IP updates have separate instructions, we can use the reduction in instructions to get an idea of the number of payload instructions in these benchmarks: If the unoptimized case has 1 IP update for n payload instructions, and the optimizations eliminate the proportion α of the IP updates on average, and the instruction reduction factor is f, we can compute $n = (1-(1-\alpha)f)/(f-1)$. This leaves us with the problem of knowing α. However, if we assume that $\alpha = 1$, we get an upper bound for n; e.g., for $f = 1.53$ (matrix), $n \leq 1.87$, while for $f = 1.2$ (brainless), $n \leq 5$. For matrix and siev *cib* eliminates all IP updates in the inner loop, and nearly all of the executed VM instructions of these benchmarks are in the inner loops, so α is close to 1, and n is close to 1.87 for matrix and close to 2.62 for siev.

6.2 Speedups from IP-update optimization variants

Figure 9 shows the speedups of the optimizations on Tiger Lake. As we will see, this is the microarchitecture where we typically see the best results, but Zen3 and Gracemont are not far off (Fig. 11).

On Tiger Lake, moving the IP updates to the start of each VM instruction (*unoptimized*) hurts a little on most benchmarks, but occasionally also helps.

Applying the loop optimization provides a speedup by a factor of about 2 on the three benchmarks (siev, bubble, matrix) that spend most of their time in short-to-medium length counted loops. However, for the huge loop body of sha512, the IP updates result in a

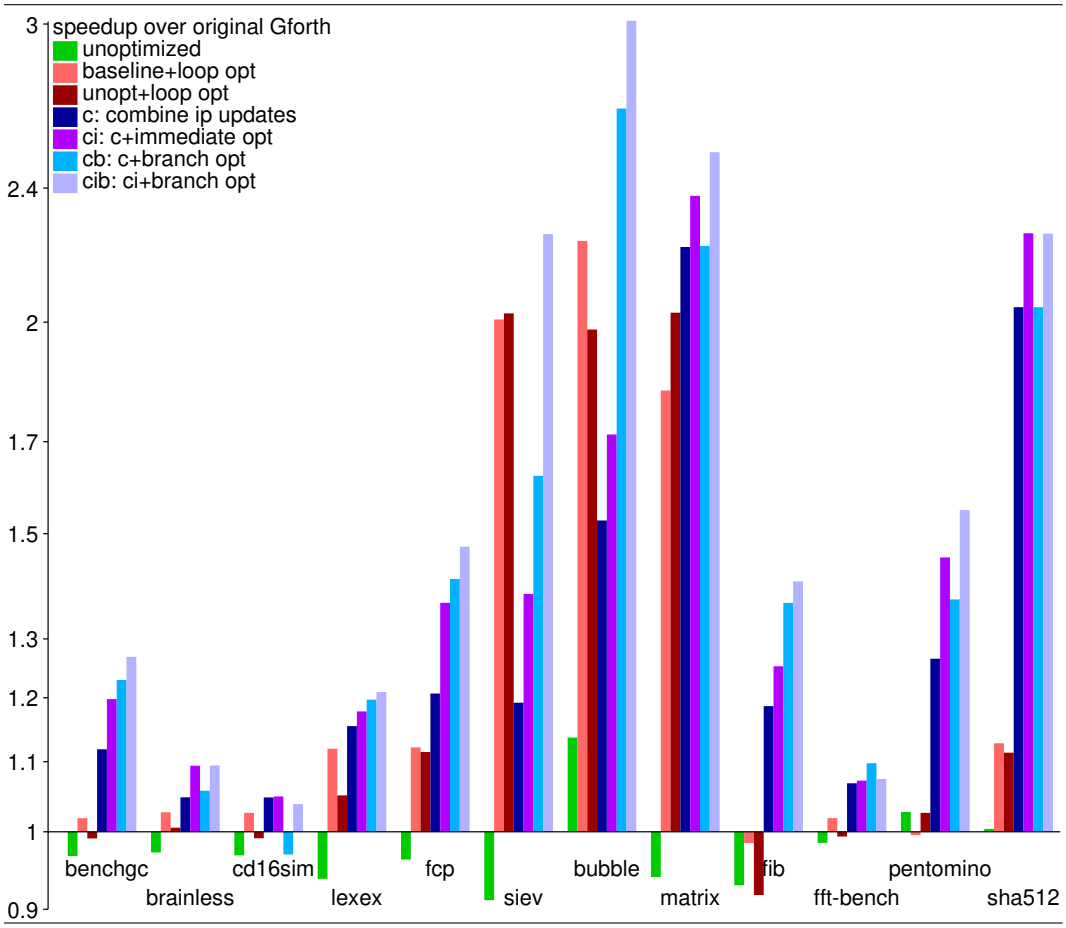

Figure 9 Speedup (reduction factor of execution cycles) on Tiger Lake of several optimizations over *baseline* (higher is better).

dependence chain that fills the processor buffers long before the loop-back branch breaks it, and the speedups of the loop optimization tend to be small. For fft-bench the inner loop is also a counted loop, but the loop body contains calls where the return breaks the IP dependence chain, so fft-bench does not benefit from the loop optimization. Pentomino hardly uses counted loops, so it cannot benefit from the loop optimization (as we implemented it). Most application benchmarks don't benefit, either.

Among the other variants, let us first look at *cib*: It dominates the loop optimization (whereas *c*, *ci* and *cb* don't, as demonstrated by siev). The speedups of *cib* depend on the benchmark, with siev, bubble, matrix, and sha512 showing a speedup of > 2 on Tiger Lake, while the speedups on fft-bench and the application benchmarks are much more modest; in code where returns break the dependence chains, the main benefit of *cib* is the reduction in the number of executed instructions.

The results of *c*, *ci*, and *cb* are helpful in understanding the *cib* results: In short loops (siev, bubble) or code with many taken branches (bubble, fib) *cb* helps more than *ci*, because the loads of the branches are a large part of the latency chain in case of *ci*. By contrast, for programs with long loop bodies like sha512, the branch optimization does not work (it

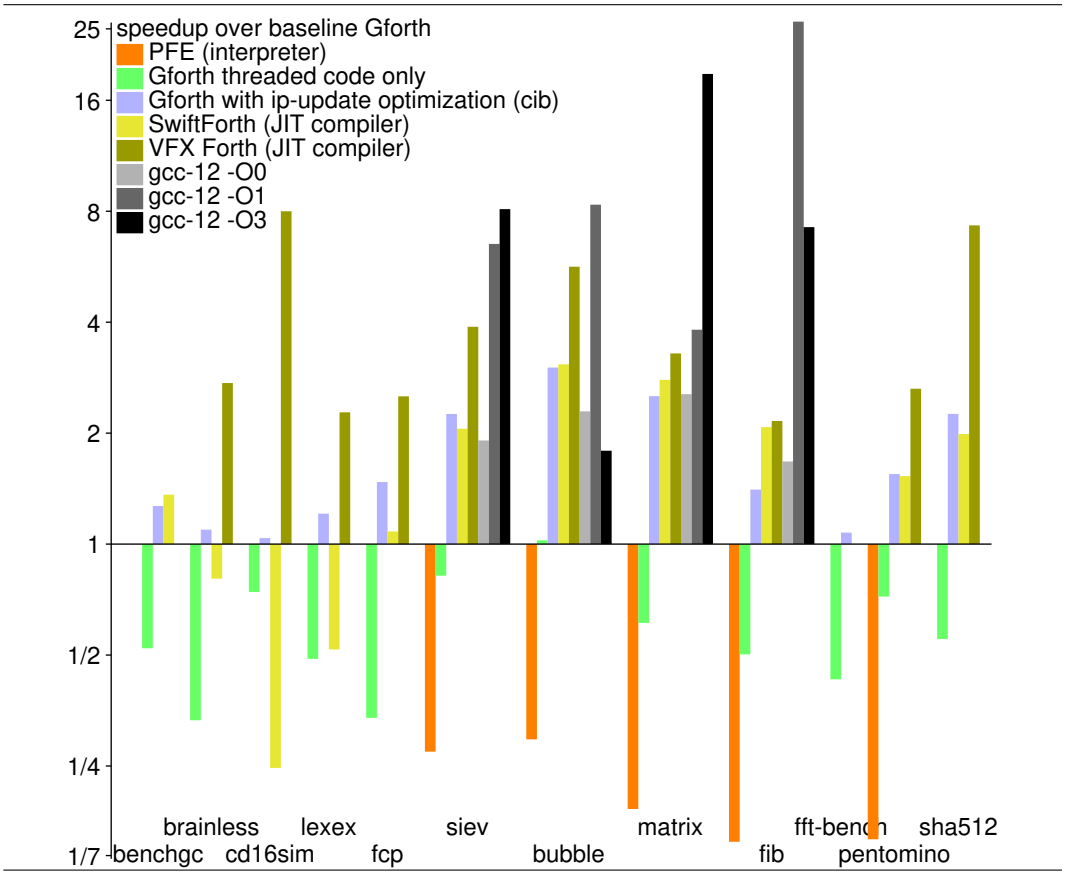

Figure 10 Speedup of several Forth systems and gcc over the Gforth baseline (higher is better), on Tiger Lake. If a benchmark does not work on a system, no bar is shown for the combination.

only covers short-distance branches) and we therefore see no difference between *ci* and *cib*. Pentomino has many branches, but many of them are long-distance branches as far as the branch optimization is concerned, so the benefit of the branch optimization is relatively small for this benchmark, and the benefit of the immediate optimization is more pronounced. Overall, for some benchmarks *ci* is better than *cb*, for others *cb* is better than *ci*; with the exception of cd16sim, both dominate *c*, and are dominated by *cib*. The difference is pretty big in some cases, so *cib* can be worth the additional implementation effort.

6.3 Comparison with other systems

Figure 10 compares a selection of the Gforth variants to several other Forth systems and to gcc. As in the other graphs, the baseline is Gforth version we started from.

Gforth with *cib* tends to be competitive with SwiftForth and with gcc -O0. SwiftForth shows some slowdowns for the application benchmarks despite executing significantly fewer instructions than Gforth; for cd16sim we identified the architectural pitfalls that it runs into[7]

[7] I-cache/D-cache ping-pong from having instructions close to data, and `ret` mispredictions from using the return address of `call` as data instead of returning to it.

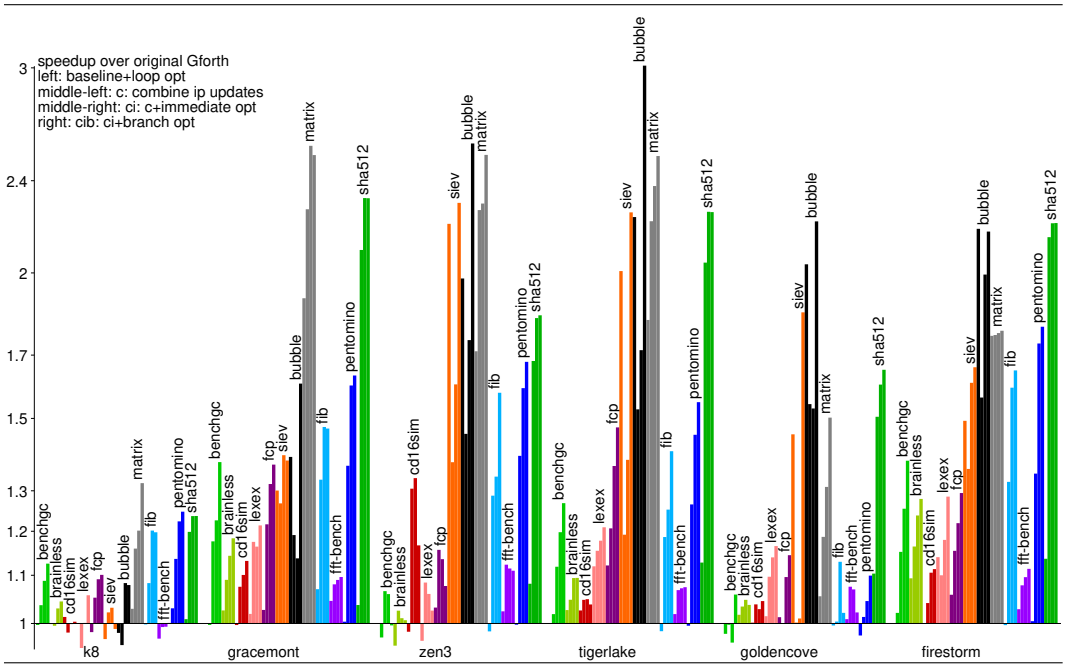

Figure 11 Speedup (reduction factor of execution cycles) of several optimizations over *baseline*; for each benchmark (colour), four bars show, from left to right: *l*, *c*, *ci*, *cib*.

and what implementation technique causes that, and reported it to the vendor. We did not investigate the SwiftForth performance on other benchmarks. The more sophisticated VFX Forth outperforms Gforth with *cib* usually by a factor of 2. Inlining of Forth definitions (performed by VFX, but not by Gforth) is particularly effective for cd16sim, leading to a speedup of VFX over *cib* by a factor of 8. Gforth with *cib* is a factor > 8 faster than PFE on the benchmarks where PFE works.

Gcc -O1 shows a factor 3–20 speedup over Gforth with *cib*, while for gcc -O3 the speedups over *cib* range from 0.6–8. For bubble the bad performance of gcc -O3 is caused by auto-vectorization, which exercises a slow hardware path for store-to-load-forwarding (due to partially overlapping accesses). We also looked at the gcc -O3 code for fib, but did not find an explanation for the slowdown compared to gcc -O1.

6.4 Speedups on different microarchitectures

Figure 11 shows a selection of the Gforth variants on several different microarchitectures. Most of them show similar speedups to Tiger Lake, which we discussed earlier.

One exception is Golden Cove (the P-Core of recent Intel CPUs); Golden Cove implements a hardware optimization that reduces the latency of adding a constant to zero cycles.[8] This hardware optimization subsumes the *c* and *i* optimizations to some extent, and consequently,

[8] `https://www.complang.tuwien.ac.at/anton/additions/`
`https://chipsandcheese.com/2021/12/21/gracemont-revenge-of-the-atom-cores/`

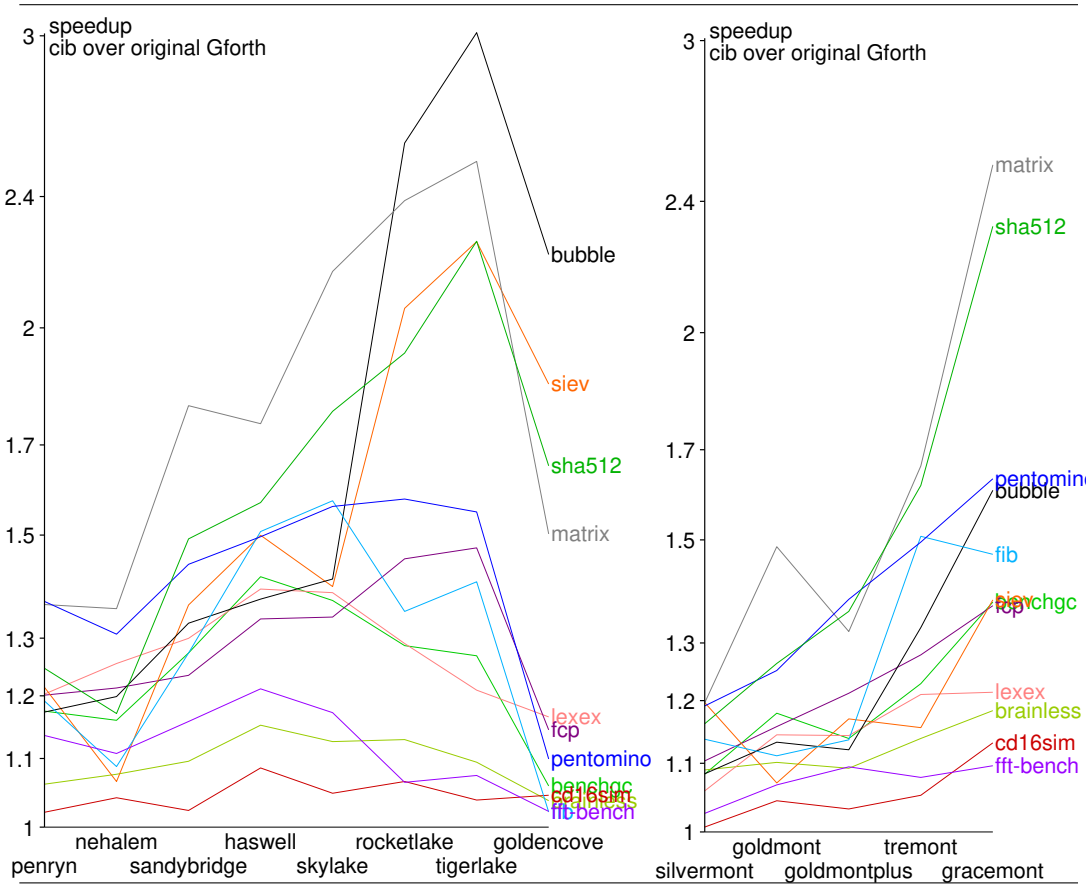

Figure 12 Speedups of *cib* over the baseline for different benchmarks on successive generations of Intel's P-cores (left) and Intel's E-cores (right).

we see lower speedups on Golden Cove than on Tiger Lake from these optimizations on a number of benchmarks. The benefit of the loop and branch optimization is still present, and shows up especially in code with short loops, such as siev and bubble, but the overall tendency is lower speedups from IP-update optimizations on Golden Cove. However, it still can pay off to apply IP-update optimizations, because CPUs with Golden Cove cores usually also have Gracemont E-cores, which benefit more from IP-update optimizations.

The other exception is the K8 microarchitecture (first released 2003). On the K8 the loop optimization tends to provide no benefit, and the other optimizations tend to provide benefits smaller than the reduction in instructions. This indicates that on the K8 the IP updates are not the critical path in instruction execution on any of these benchmarks.

We also looked at a variety of other CPUs (Fig. 12), and the tendency is that within a family of microarchitectures (e.g., Intel's P-cores, with the exception of Golden Cove, as discussed above), the speedups from *cib* tend to be higher for more recent microarchitectures and lower for older microarchitectures. Along with the K8 results this explains why investigations on IP update optimizations have not been published earlier.

7 Applicability to other languages

In principle the IP-update optimizations can be applied to any VM implementation. In practice the benefit depends on how light-weight or heavy-weight the payload of your VM instructions is, on the characteristics of the executed programs.

Concerning program characteristics, loop-dominated programs benefit much more from the IP-update optimizations than call-dominated programs (see Section 6.2); but note that if you implement inlining or tail-call optimization, this can change call-dominated programs into loop-dominated programs.

Concerning the weight of VM instructions, IP update optimizations benefit VMs with lightweight instructions, such as Gforth, the OCaml interpreter, the JVM or WebAssembly. The lighter the payload is, the more these optimizations pay off.

By contrast, for a language implementation like Tcl with its heavy VM instructions, already dynamic superinstructions did not pay off; the speedup from the reduced dispatch overhead was small, and was compensated by increased I-cache misses [22].

Even if the VM instructions are middle-weight, we expect the benefit of the IP-update optimization to be small. E.g., if a VM instruction has an average payload of 10 instructions per VM instruction, the bottleneck will be in the payload (in the resource requirements, or in the latency), and the only benefit of the IP-update optimization will be to reduce the resource load, and that contribution will be relatively small (10%).

If you design a virtual machine that is lightweight enough that IP updates could be a bottleneck one day, it's a good idea to make it flexible enough make the loop optimization (Section 4.1) possible, which can be applied with relatively low effort.

8 Source code

The source code is in the git repository of Gforth:

```
git clone https://git.savannah.gnu.org/git/gforth.git
```

After that you can get the versions used for generating the data with:

```
cd gforth
# one of:
git checkout ecoop24-ip-updates-baseline #baseline
git checkout ecoop24-loopopt             #baseline+loop opt
git checkout ecoop24-ip-updates   #unopt, unopt+loop opt, c, ci, cb, cib
git checkout master #Gforth mainline
```

The main line of Gforth now uses *cib* by default.

You can find a package containing the checked out source code, binaries for AMD64, ARM A64, and RV64GC, benchmarks, and the resulting data on `https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz`.

9 Related work

One difference between the approaches of Piumarta and Riccardi [17] and Ertl and Gregg [4] on selective inlining/dynamic superinstructions is that Piumarta and Riccardi eliminated the VM instruction slots of the VM instruction slots that are no longer needed for threaded-code dispatch. This eliminates as many IP updates as the *c* optimization, but Ertl and Gregg

expected that this "does not have much effect". And indeed, the K8 results (similar to the hardware they used at the time) show a speedup ≤ 1.1 for *c* on most benchmarks (Section 6). However, on newer cores `c` provides speedups > 1.3 on some benchmarks, and we expect the same speedups from Piumarta and Riccardi's instruction slot elimination. In any case, neither paper gives any performance evaluation of this issue, while the present work does and also explores additional optimizations: for loops, immediates, and branches.

kForth implements counted loops in the same way as the *l* optimization.[9]

There has been a significant body of work on combining VM instructions at VM-interpreter build time into (static) superinstructions [13, 18, 16, 11, 8, 3], which reduces instruction pointer updates, among other benefits. But again, none of these works have evaluated how much of the benefit is due to reducing IP updates.

More recent work on interpreter performance includes Rohou et al.'s reevaluation of the performance impact of indirect branches in the light of improvement in hardware indirect branch predictors [19], and Titzer's work on an in-place interpreter for WebAssembly (which has been designed for translation) [21].

Larose et al. [14] argue that a sophisticated metacompiler (like RPython and Truffle) can optimize an AST interpreter written in a high-level language just as well as a VM interpreter. However, unless they completely eliminate all references to the AST or the VM code, they still have to maintain a pointer to the AST or VM code, and optimizing the IP updates is relevant.

In more ambitious earlier work [6], Ertl and Gregg eliminated the VM instruction pointer completely by eliminating all accesses to the threaded code: like the present work, it concatenates code snippets produced with gcc, but it patches constants and branch targets into the copied code snippets, making all access to the threaded code unnecessary. They found a median speedup by a factor 1.32 on a K7 (a 32-bit-only predecessor of the K8), quite an interesting contrast to the more modest speedups of the present IP-update optimization on the K8. However, this approach requires architecture-specific support for patching the constant and branch targets, whereas the present work is just as architecture-independent as dynamic superinstructions. This approach cannot fall back to threaded code, and therefore did not make it from proof-of-concept into a production feature of Gforth.

Xu and Kjolstad [24] have also used code snippets produced by a compiler and combined them, patching in constants and branch targets. The result also does not need a VM instruction pointer and its updates, and moreover, it uses the architecture's call and return instructions (instead of jumps and indirect jumps. The price paid for this, like in Ertl and Gregg's work [6], is architecture-specific code for patching the results.

The Maxine virtual machine [23] contains T1X, a compiler that uses snippets coming from a Java compiler, again without referencing the VM code, but also requiring architecture-specific code.

10 Conclusion

The IP-update optimization combination *cib* reduces the number of executed instructions by roughly a factor 1.2 on AMD64, ARM A64, and RISC-V. The effect on performance varies a lot across microarchitectures and benchmarks, between slowdowns by a factor 1.1 and speedups by a factor 3.01.

The reason for the more spectacular speedups is that, without *cib*, IP-update dependence chains become the critical path of execution on loop-dominated programs.

[9] `news:<us68iq$3jsgk$1@dont-email.me>`

Another way to address this problem is the *l*oop optimization: perform a loop-back branch to a location stored at loop entry, breaking the dependence chain. While the speedups from this optimization are not quite as spectacular as those from *cib*, and this optimization speeds up only some benchmarks, it is much easier to implement.

References

1 James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

2 Helmut Eller. Optimizing interpreters with superinstructions. Diplomarbeit, TU Wien, 2005. URL: `https://www.complang.tuwien.ac.at/Diplomarbeiten/eller05.ps.gz`.

3 M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg03euroforth.ps.gz`.

4 M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg03.ps.gz`.

5 M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg04ivme.ps.gz`.

6 M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg04pact.ps.gz`.

7 M. Anton Ertl and David Gregg. Stack caching in Forth. In *21st EuroForth Conference*, pages 6–15, 2005. URL: `https://www.complang.tuwien.ac.at/papers/ertl%26gregg05.ps.gz`.

8 M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. `vmgen` — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002. URL: `https://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz`.

9 M. Anton Ertl and Bernd Paysan. Gforth. Software, version 0.7.9_20240821., swhId: `swh:1:dir:61eb3b71325060fe2e01f5e819eb0bec959e5bf0` (visited on 2024-09-02). URL: `https://git.savannah.gnu.org/cgit/gforth.git`.

10 M. Anton Ertl and Bernd Paysan. ip-updates. Collection, version 7. (visited on 2024-09-02). URL: `https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz`.

11 David Gregg and John Waldron. Primitive sequences in general purpose Forth programs. In *18th EuroForth Conference*, pages 24–32, 2002. URL: `http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/gregg.ps.gz`.

12 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3062341.3062363`.

13 R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.

14 Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. AST vs. bytecode: Interpreters in the age of meta-compilation. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. `doi:10.1145/3622808`.

15 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, first edition, 1997.

16 Henrik Nässén, Mats Carlsson, and Konstantinos Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Principles and Practice of Declarative Programming (PPDP01)*, 2001. URL: `http://www.csd.uu.se/%7Ekostis/Papers/sicstus.ps.gz`.

17 Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998. URL: `ftp://ftp.inria.fr/INRIA/Projects/SOR/papers/1998/ODCSI_pldi98.ps.gz`.

18 Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.

19 Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015. URL: `https://hal.inria.fr/hal-01100647/document`.

20 Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996. URL: `http://www.cs.hut.fi/~cessu/papers/dispatch.ps`.

21 Ben L. Titzer. A fast in-place interpreter for WebAssembly. *Proc. ACM Program. Lang*, 6(OOPSLA2):148:1–148:27, 2022.

22 Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.

23 Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30:1–30:24, January 2013.

24 Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation. *Proc. ACM Program. Lang.*, 5(OOPSLA):136:1–136:30, October 2021. URL: `https://fredrikbk.com/publications/copy-and-patch.pdf`.