Static Basic Block Versioning

Olivier Melançon
Université de Montréal, Canada

Marc Feeley
Université de Montréal, Canada

Manuel Serrano
Inria/UCA, Inria Sophia Méditerranée, Sophia Antipolis, France

Abstract

Basic Block Versioning (BBV) is a compilation technique for optimizing program execution. It consists in duplicating and specializing basic blocks of code according to the execution contexts of the blocks, up to a version limit. BBV has been used in Just-In-Time (JIT) compilers for reducing the dynamic type checks of dynamic languages. Our work revisits the BBV technique to adapt it to Ahead-of-Time (AOT) compilation. This Static BBV (SBBV) raises new challenges, most importantly *how to ensure the convergence of the algorithm* when the specializations of the basic blocks are not based on profiled variable values and *how to select the good specialization contexts*. SBBV opens new opportunities for more precise optimizations as the compiler can explore multiple versions and only keep those within the version limit that yield better generated code.

In this paper, we present the main SBBV algorithm and its use to optimize the dynamic type checks, array bound checks, and mixed-type arithmetic operators often found in dynamic languages. We have implemented SBBV in two AOT compilers for the Scheme programming language that we have used to evaluate the technique's effectiveness. On a suite of benchmarks, we have observed that even with a low limit of 2 versions, SBBV greatly reduces the number of dynamic type tests (by 54% and 62% on average) and accelerates the execution time (by about 10% on average). Previous work has needed a higher version limit to achieve a similar level of optimization. We also observe a small impact on compilation time and code size (a decrease in some cases).

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases Compiler, Ahead-of-Time Compilation, Optimization, Dynamic Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.28

1 Introduction

Optimizing compilers perform various analyses to discover properties of the program that are preconditions for performing optimizations. In a Just-In-Time (JIT) compiler, the cost of these analyses and optimizations is a critical issue as the time they take becomes part of the program's execution time. The use of expensive analyses and optimizations incur a long *warm-up* where the first part of a program's execution is sluggish and the program may even terminate before it has reached an optimization steady state.

Basic Block Versioning (BBV) is an optimization approach that strikes a balance between the optimization cost and the speed of the generated code to achieve a fast warm-up time and reasonably good execution speed. BBV has been used in JIT compilers for dynamically typed programming languages; in research compilers for JavaScript [5, 6] and Scheme [27, 28], and it is now used successfully in production in the official Ruby implementation [7, 8].

BBV uses the program's Control Flow Graph (CFG) created by the compiler as a template for creating a specialized CFG. For this, BBV traverses the CFG starting at its entry point while keeping track of the *context* that contains program properties of relevance

for specialization, such as the type of the values contained in the live variables. Each basic block has a set of contexts. The information contained in this set is *conservative*: for each possible program state when that basic block is reached during a program execution, there must be at least one context consistent with that state. Due to its conservative nature, it is allowed to have unreachable contexts in the set. In principle, each basic block of the original CFG could be specialized to all contexts in its context set, including unreachable contexts. The specialized versions of a basic block may contain optimizations that are valid in the corresponding context, such as the elimination of type checks when the type of a value has been determined at an earlier point in the execution.

An important concern is that multiple specialized copies of each basic block may be created, leading to a larger amount of code (*bloat*) and a longer compile time, load time, and execution time (due to the reduced performance of the instruction cache, among other reasons). In theory the bloat can be exponential in the size of the program.

Previous works use the same approach to this issue: a cap is placed on the number of versions for each basic block (*i.e.*, the number of versions is no greater than N, typically a small number like 5 or 10). In a JIT compiler the versions of a basic block are generated as the program's execution advances and reaches a basic block with a new context. This variant of BBV is called *lazy* BBV. When a new context is encountered and this would cause the version limit to be reached, a version specialized to that context must not be created because it would prevent another specialization if one was needed later in the execution. Instead, a *fully generic* version that covers all possible contexts is created as the last version and is used whenever a new version would be needed. Lazy BBV is relatively simple to implement but it has some important limitations:

- **Lazy BBV is a greedy algorithm.** The versions that are generated before the generic version, which are the first ones encountered at execution time, may not be the versions that are part of hot code. For example, if some function is used both during the initialization phase and in the main part of the program, then the specializations will be focused on what happens in the initialization phase. This function may be hot code when called from the main part of the program in a new context and, because the version limit is reached, it will be using the generic version (likely the slowest of them all).
- **High specialization is hard to achieve reliably.** The precision of the versioning context, *i.e.*, the number and information content of the program properties it tracks, has a direct impact on how quickly the slow generic version is used. For example, a precise versioning context that tracks not only the type but the range of values of an integer loop iteration variable starting at 1 and incremented at each iteration, will be able to create specialized versions of the first few iterations of the loop body (one version for each specific value of the iteration variable below N). This will not work well for loops that have a large number of iterations, because the iterations N and above will be handled by a slow generic version of the loop body. On the other hand, a context of this precision will work very well for programs where the loops have fewer than N iterations because BBV will completely unroll these loops. The BBV implementer will have to choose a moderate precision of the versioning context to avoid using the generic version too quickly, and consequently this will miss optimizations in some cases, such as total loop unrolling.
- **It requires JIT compilation.** The nature of a JIT compiler makes it easy to ensure that versions for unreachable contexts are never created. Unfortunately, this entails a warm-up time at execution, and in some use cases JIT compilation is not an option. In [5], an *eager* variant of BBV suitable for an Ahead-of-Time (AOT) compiler was described and compared to lazy BBV. That implementation of eager BBV yielded comparatively

poor speed and bloat because specialization is not guided by the actual need of a program execution and parts of the CFG that are explored are not typically executed, such as error cases and out-of-line handlers. Consequently, the specialized versions created before the limit is reached are more likely to be irrelevant at improving execution speed.

In this paper we describe a new design for a BBV algorithm suitable for an AOT compiler that mitigates these limitations. Our algorithm also traverses the CFG to determine which contexts reach each basic block. The first main difference with previous work is the handling of the version limit. When a new context is encountered and this would cause the version limit to be exceeded, the algorithm heuristically chooses a pair of contexts reached for that basic block and replaces them by a *merged* context that is more conservative than the contexts in the pair (in other words, a more general context). The algorithm continues traversing the CFG until a fixed-point is reached, *i.e.*, no new versions need to be created. The use of context merging allows contexts to be very precise at first, and it is the algorithm that reduces the precision as needed to keep the number of versions within the limit. The second main difference with previous work is the refinement of the notion of types to integer intervals to allow the BBV optimization to remove integer arithmetic overflow checks and array indexing bound checks.

In the next section, we present our algorithm and discuss its termination. In Section 3 we extend the algorithm with more precise contexts. The implementation in two mature compilers is explained and evaluated in Section 4. Related work is given in Section 5.

2 The Static BBV Algorithm

In this section we present the *Static* BBV (SBBV) algorithm. We will start with an overview by illustrating the algorithm's behavior using the traditional `find` function that many dynamic and functional languages provide.

2.1 SBBV by Example

The `find` function takes a predicate and a list of values and it returns the first element of the list that satisfies the predicate or false if no such element is found. In the Scheme programming language [17], which we use throughout this paper, it can be defined as:

```
1 (define (find p x)            ;; p is the predicate and x is the list to search
2    (if (pair? x)              ;; is the list non−empty?
3        (if (p (car x))        ;; call predicate on the first element
4            (car x)            ;; return it if it satisfies the predicate
5            (find p (cdr x))) ;; otherwise, continue searching the rest of the list
6        #f))                   ;; return false when no element in the list satisfies the predicate
```

The safety of this code is guaranteed by verifying the validity of the arguments of the primitive operations at run time. In this example, the primitive operations for which a type verification is needed are the `car` and `cdr` accessors (lines 3-5) and the function invocation of the predicate (line 3). In safe mode, a Scheme compiler adds the required dynamic checks to the code, making the possible points of failed safety checks explicit:

```
1 (define (find p x)
2    (if (pair? x)
3        (if ((if (procedure? p) p (fail)) (if (pair? x) (car x) (fail)))
4            (if (pair? x) (car x) (fail))
5            (find p (if (pair? x) (cdr x) (fail))))
6        #f))
```

Here we use calls to the `fail` function to indicate cases where execution cannot continue due to a failed verification. The `fail` function is special in that it never returns.

One might expect a smart compiler, such as one implementing *occurrence typing* [32], to discover that the type tests on `x` are redundant, but let us assume that no such optimization is applied. This is done for illustrative purpose and because one of our objectives is to show that SBBV subsumes other optimization techniques, such as occurrence typing.

The unoptimized CFG of `find` is displayed in Figure 1a. SBBV will produce an optimized version of that CFG with fewer dynamic checks. For that, it propagates the information about variables in order to produce *specialized* versions of the basic blocks. For instance, block `#12` (Figure 1a) checks that the end of the list is not yet reached by testing if `x` is a pair. In the positive branch, that is the path starting at block `#4`, it is known to be a pair and no further type tests are needed to ensure the correct execution until an assignment to `x` occurs (*i.e.*, the other tests that `x` is a pair are redundant).

The CFG produced by SBBV is shown in Figure 1b. We observe that SBBV has isolated the first iteration of the loop of the `find` function and the other iterations are handled by the loop formed by the subgraph { `#23`, `#25`, `#27`, `#29`, `#31`, `#32` }. In that loop the type of `p` is never tested because it has been tested in the first iteration before entering the loop handling the other iterations, and the argument `x` is only tested once per iteration, which is a necessary part of the loop termination logic. We also observe that for this simple example, SBBV has produced an optimal CFG in the sense that in a dynamic context with no global knowledge about the variable types and the data structure types, it executes the minimum number of dynamic checks required to ensure a safe execution. In Section 4.2 we evaluate the number of type tests SBBV is able to remove on more realistic programs.

In the rest of this section, we present and explain the algorithm that transforms the graph of Figure 1a into that of Figure 1b.

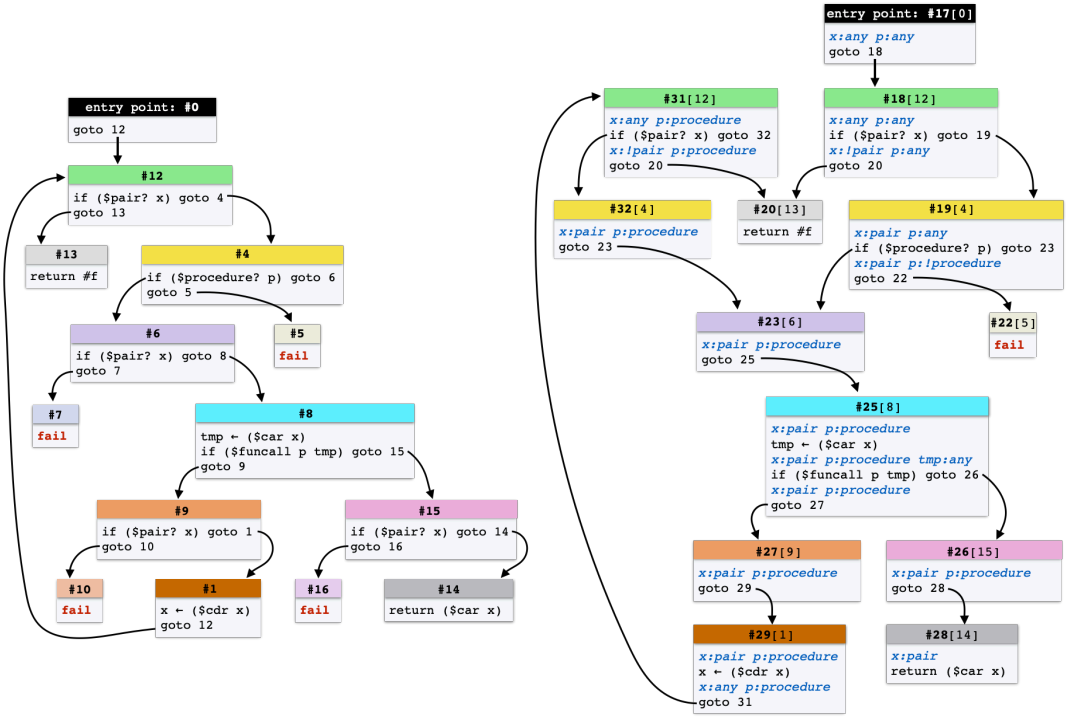
2.2 The Algorithm

SBBV specializes a CFG, which might either denote the whole program to be compiled or only a fragment of it. For instance, it can be decided to use SBBV for specializing each function in isolation or to specialize the whole program at a time. The main function of the algorithm (Algorithm 1) takes a basic block to specialize and the initial context used for that specialization as parameters. The data structures it uses are presented in Figure 2. A context is a mapping of variable names to value information. In this section these mappings associate variables to types. We will see in Section 3.2 that contexts can contain more precise type information. The algorithm specializes each instruction of the basic block and it recursively specializes the blocks that follow the basic block currently under specialization.

The algorithm performs a breadth-first traversal of the CFG. For that, it uses a work queue where it pushes the basic blocks and contexts that need further specializations. Specializing a block may cause the algorithm to specialize new blocks. For instance, when the algorithm scans the block `#12` of Figure 1a, it discovers that in the positive branch the variable `x` is known to be a pair and then, it pushes onto the work queue the demand of a new specialization of the block `#4` for the context { `x` $\mapsto$ pair }.

If the block extracted from the queue has not been merged (we explain in a moment what it means for a block to be merged), then it is specialized (line 10). This, in turn, can add new pending specializations to the queue. The algorithm proceeds until the queue is empty.

To ensure the convergence of the algorithm, it is enough to ensure that the function BlockNewVersion (Algorithm 2) pushes a new block onto the queue if and only if no such block has already been specialized for the requested context as the contexts contain a finite number of variable to value mappings. However, this convergence criteria is not enough in practice, the code size expansion of the specialization should also be controlled. This is the



(a) Original unspecialized CFG. **(b)** Specialized CFG.

Figure 1 The original and specialized CFGs of the `find` function. Basic blocks have a numeric label for easy reference. In the specialized CFG, basic blocks have the same color as the original block they were specialized from (whose label is in square brackets). For instance, block `#19` is a specialized version of block `#4` of the original CFG. In the specialized CFG, the contexts of the specializations are displayed in blue. It can be observed that a given block can be specialized multiple times. For instance, block `#12` has been specialized twice (blocks `#18` and `#31`) and block `#4` has been specialized twice (blocks `#19` and `#32`). Blocks `#7`, `#10`, and `#16` have no version in the specialized CFG because they do not have a reachable specialized context. Blocks `#9` and `#15` have been specialized to an unconditional jump because the `($pair? x)` test is known to be true.

context: ctx, ctx_0, ...
- *types:* a list of mappings of variable to type information

basic block: bb, bb_0, ...
- *instrs:* a list of instructions
- *versions:* a list of specialized basic blocks

specialized basic block: bs, bs_0, ...
- *bb*: the corresponding basic block
- *merge*: a specialized basic block into which it has been merged or false
- *ctx*: a context

Figure 2 The data structures used in the SBBV algorithm.

Algorithm 1 Main algorithm.

```
1:  function SBBV(bb_0, ctx_0)                ▷ specialize a CFG starting with the block bb_0
2:      wq ← empty_queue                      ▷ create a fresh queue used for this specialization
3:      bs_0 ← BlockNewVersion(bb_0, ctx_0, wq) ▷ push the request for specialization of bb_0
4:      while ¬wq.isempty() do
5:          bs ← wq.pop()                     ▷ fetch the first bb of the queue
6:          bb ← bs.bb                        ▷ get the original unspecialized bb
7:          if |{∀b ∈ bb.versions, ¬b.merge}| > VERSION_LIMIT(bb) then
8:              BlockMergeSome(bb, wq)        ▷ too many specializations, merge
9:          if ¬bs.merge then
10:             BlockSpecialize(bs, wq)       ▷ specialize the block only if not already merged
11:     return bs_0                           ▷ return the specialization of the initial block
```

purpose of the test at line 7 of Algorithm 1. If the number of specialized versions of a single block exceeds a threshold, which is a parameter of the algorithm, some specializations of that block must be merged (see line 8).

The ancillary function BlockNewVersion is responsible for creating new blocks to be specialized. First, it checks if the requested block already exists, in which case it returns it. Otherwise, it creates a fresh version and pushes it onto the queue (line 9). Note that at this stage the instructions of the block are not scanned nor specialized. Pushing the block onto the queue is a mere request for specialization. It might be the case that at the moment where the block will be popped from the queue that this block has been merged into a less specialized version. This happens to prevent code size explosion.

Algorithm 2 Create or fetch a specialized version.

```
1:  function BlockNewVersion(bb, ctx, wq)
2:      if ctx ∈ bb.versions then
3:          return BlockLive(bb.versions[ctx])   ▷ return the already specialized block
4:      else
5:          bs ← new basic block                 ▷ create a fresh empty basic block
6:          bb.versions[ctx] = bs                ▷ connect the new block and the parent block
7:          bs.bb = bb                           ▷ initialize the new block
8:          bs.ctx = ctx
9:          wq.push(bs)                          ▷ push it onto the queue for future specialization
10:         return bs
```

The utility function BlockLive returns the first specialized version of a block that has not been merged into a more general version.

Algorithm 3 Follow a chain of merged blocks.

```
1:  function BlockLive(bs)
2:      if bs.merge then
3:          return BlockLive(bs.merge)
4:      else
5:          return bs
```

When the number of specializations of a basic block bb exceeds VERSION_LIMIT(bb), some contexts need to be merged. Note that we express the version limit as a function of the basic block to allow the algorithm to adapt the limit to different types of basic blocks, such as

those marked by the compiler front-end as probably benefiting from more specialization. This function could simply return a constant value, as we have done in our experiments. Context merging is done by the function BlockMergeSome (Algorithm 4). It selects two versions not already merged (line 2), computes the union of the two corresponding contexts (line 3), and then replaces the merged blocks in the CFG (line 14). Merging is only triggered when a version is removed from the work queue (Algorithm 1, line 5). This allows the number of versions to temporarily exceed the version limit. This *delayed merging* increases the choices available to the selection heuristic and possibly leads to better merges.

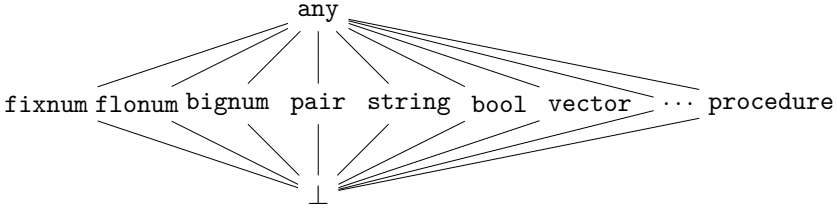
Merging a block in the CFG entails deleting all incoming edges of the merged blocks to redirect them to the block resulting from the merge. Any deletion of an edge may render some specialized block unreachable from the CFG's entry point. Similarly, an added edge can make some previously unreachable block reachable anew. Keeping track of unreachable blocks is required since those no longer have to be traversed and must not be considered when selecting versions to merge in BlockMergeSome (line 2). This can be done efficiently by maintaining an Even-Shiloach tree [14] of the CFG to help the SBBV algorithm detect whenever the reachability of a specialized block changes. Any block made unreachable by a merge is marked so that it is no longer considered in the merge selection and CFG traversal.

Algorithm 4 Merge blocks.

```
1:  function BlockMergeSome(bb, wq)
2:      bs1, bs2 ← Θ²{ ∀b ∈ bb.versions, ¬b.merge }      ▷ select two versions to merge
3:      ctx ← bs1.ctx ⋓ bs2.ctx                          ▷ merge the two corresponding contexts
4:      bs ← BlockNewVersion(bb, ctx, wq)                ▷ create a new block for the merge
5:      if bs1 ≡ bs then                                 ▷ replace the merged blocks
6:          BlockMergeAndReplace(bs2, bs)
7:      else if bs2 ≡ bs then
8:          BlockMergeAndReplace(bs1, bs)
9:      else
10:         BlockMergeAndReplace(bs1, bs)
11:         BlockMergeAndReplace(bs2, bs)
12: function BlockMergeAndReplace(obs, mbs)
13:     obs.merge ← mbs                                  ▷ mark that obs is merged into mbs
14:     replace obs with mbs in the CFG                  ▷ patch the CFG
```

The operator Θ^2 selects two unmerged specializations for the block *bb*. For the sake of the correctness of the algorithm, this operator might select any two versions. For instance, it could select two random versions. Of course, a better operator would positively impact the result of the compilation. In Section 4.3 we present the operator we have used so far.

The behavior of the merge operator $\curlyvee$ is independent of the SBBV algorithm but it must produce a new context that at least encompasses the contexts of the merged blocks. The natural solution is to use a lattice for organizing the information associated with variables and to go up some level for each merge. We will use the following lattice:



Note that a number can be a `flonum` (floating point numbers), a `fixnum` (small integers that fit in a machine word), or a `bignum` (integers that don't fit in a machine word). While this would be a good match for JavaScript numerical types, for a full Scheme, or Python, implementation there would also be a representation for rational and complex numbers, but we will ignore them to simplify the discussion.

Consider the merge of two contexts mapping a variable v respectively to the types `fixnum` and `flonum`. The merge operation should produce a context mapping v to `any`. In Section 3.2 we show the use of a more precise lattice for representing properties.

The last part of the SBBV algorithm is in charge of specializing blocks. It merely creates a new block where the instructions have been specialized one by one.

Algorithm 5 Specialize a block and its instructions.

```
1: procedure BlockSpecialize(bs, wq)
2:     bb ← bs.bb
3:     ctx ← bs.ctx
4:     for all i ∈ bb.instrs do
5:         (ni, nctx) ← InsSpecialize(i, ctx, wq)
6:         bs.instrs ← bs.instrs + ni
7:         ctx ← nctx
```

Specializing an instruction that implements a type test produces a new context. For instance, when specializing the block `#12` of Figure 1a, in the positive branch, the argument `x` is known to be a pair. The block `#4` is then specialized with a context that reflects that information, which is then propagated to following blocks. Conversely, on the negative branch, the argument `x` is known not to be a pair. This information too, is propagated to following blocks. To handle these evolutions of the contexts, the procedure BlockSpecialize updates the context it uses for specializing the instructions after each iteration (Algorithm 5, line 7).

The function InsSpecialize selects a specializer appropriate for the instruction.

Algorithm 6 Specialization of an instruction.

```
1: function InsSpecialize(i, ctx, wq)
2:     if i.kind is "goto" then return InsSpecializeGoto(i, ctx, wq)
3:     else if i.kind is "if" then return InsSpecializeIf(i, ctx, wq)
4:     else if i.kind is ... then ...
5:     else return (i, ctx)
```

The specialization of a `goto` instruction mostly consists in forwarding the specialization context ctx to the target of the instruction. For instance, when specializing the `goto` in block `#1` of Figure 1a with the context $\{\texttt{x} \mapsto \text{pair}, \texttt{p} \mapsto \text{procedure}\}$ the `goto` instruction triggers the specialization of the block `#12` with the same context. The instruction brings no new knowledge about the variables types so it returns an unmodified context.

Algorithm 7 Specialization of `goto` instructions.

```
1: function InsSpecializeGoto(i, ctx, wq)
2:     ni ← i.dup()
3:     ni.target ← BlockNewVersion(i.target, ctx, wq)
4:     return (ni, ctx)
```

The specialization of an `if` is more involved because this is where new knowledge is acquired and where requests for new specializations are emitted. If the test of the expression is not a type test, the specialization behaves as the specialization of a `goto` instruction (Algorithm 8, line 2). If the instruction implements a type check and if the context is such that the test always succeeds, then the instruction is replaced with a `goto` instruction which directly branches to the positive block (line 6). This is illustrated by the specialization of the block `#6` of Figure 1a into the block `#23` of Figure 1b. Conversely, if the test is known to evaluate to false, the instruction is replaced with a `nop` instruction.

Algorithm 8 Specialization of `if` instructions.

```
1:  function InsSpecializeIf(i, ctx, wq)
2:      if ¬ i.test is a typecheck then
3:          ni ← i.dup()
4:          ni.target ← BlockNewVersion(i.target, ctx, wq)
5:          return (ni, ctx)
6:      else if ctx.types.isTrue(i.test) then
7:          ni ← new insGoto(i.target)
8:          return InsSpecialize(ni, ctx, wq)
9:      else if ctx.types.isFalse(i.test) then
10:         ni ← new insNop()
11:         return (ni, ctx)
12:     else
13:         ni ← i.dup()
14:         ctx+ ← ctx ∪ { i.test.var ↦ i.test.type }
15:         ctx− ← ctx ∪ { i.test.var ↦ ¬ i.test.type }
16:         ni.target ← BlockNewVersion(i.target, ctx+, wq)
17:         return (ni, ctx−)
```

The most interesting situation is when the result of the type test cannot be inferred from the current context (line 12). In that case, two new contexts are created, in accordance to the narrowing rules for that test. The positive branch of the test will be specialized with a context reflecting the success of the type test and the context reflecting a negative result is returned to the procedure BlockSpecialize (for instance, see the block `#12` of Figure 1a that creates the context of the block `#19` of Figure 1b).

3 Improved Specializations

In Section 2 we have presented the general SBBV algorithm. We have exposed its principles that we have illustrated with a simple type analysis that maps variable usages to types. In this section, we show how the algorithm can be extended to specialize the blocks according to more fine-grained information.

3.1 Variable Aliasing

The contexts used in the example in Figure 2 enables SBBV to specialize blocks according to the type of the variables. However, it does not keep track of variable aliases, which jeopardizes the benefit of the optimization. Compilers tend to introduce many temporaries for evaluating expressions and if these aliases are not handled efficiently by the SBBV specialization, what is learned about a variable's value will not be propagated to the other

context: ctx, ctx_0, ...
- types: a list of mappings of variable to type information
- equiv: **a list of equivalence classes**

Figure 3 Extended specialization contexts with variable class equivalence.

variables containing the same value. Thankfully, handling aliases merely requires extending the definition of the contexts and to handle the specialization of the `mov` instruction, which assigns a value to a variable. The new definition of the contexts is extended into that of Figure 3. The specialization of the `mov` instruction, that copies a variable into another and that is represented by the $\leftarrow$ operator in the CFGs, is given in Algorithm 9. For the sake of simplicity, this extension keeps track of aliasing of read-only variables only. Assigned variables are never treated as aliases of other variables. Also, not presented here, the specialization of the `if` instruction (Algorithm 8) is modified so that it also propagates the gathered type information to the variable's aliases.

Algorithm 9 Specialization of `mov` instructions.

```
1: function InsSpecializeMov(i, ctx, wq)
2:     nctx ← ctx
3:     nctx.equiv[i.target] ← ∅
4:     if i.source is a read-only variable then
5:         nctx.equiv[i.target] ← {i.source}
6:     return (i, nctx)
```

3.2 Specialization of Arithmetic Operations

The SBBV algorithm is general-purpose and it can be applied to other properties of the variables and values to go beyond type check removal. In this section we show how to leverage this flexibility to also specialize the basic blocks according to fixnum integer intervals LO..HI, where LO and HI are values in the fixnum range. This will allow the compiler to generate code using fixnum arithmetic operators that are fast (because they directly map to machine instructions) and removing overflow checks and bound checks.

The benefits of this extension can be illustrated on the expression `(+ x 1)`, which adds 1 to `x`, a very common operation in most programs. The specific operation executed depends on the type of `x`. The result could be a fixnum, a flonum, a bignum, or the operation could raise an exception if `x` is not a numerical type. Moreover, the result of `(+ x 1)` will be a bignum if `x` is maxfix, the largest fixnum value. A similar dispatch is part of the semantics of most arithmetic operators (`-`, `*`, ...) and comparison operators (`=`, `<`, ...), and in the general case, such as `(+ x y)`, the dispatch is on the combination of types of x and y.

Optimizing compilers usually inline the handling of the most common cases, such as all operands being fixnums, and all operands being flonums, and defer the handling of the remaining cases to an out-of-line function. For example, the expression `(+ x 1)` could be expanded by the compiler to this code:

```
(if ($fixnum? x)
    (or ($fx+? x 1)    ;; fixnum add 1 with overflow check (#f returned on overflow)
        ($+ x 1))      ;; call $+ function to handle bignum result case
    (if ($flonum? x)
        ($fl+ x 1.0)   ;; flonum add 1
        ($+ x 1)))     ;; call $+ function to handle other cases including errors
```

Here we use names prefixed with '`$`' to indicate internal operations of the system:

- `($fixnum? x)` and `($flonum? x)` test to see if `x` is a fixnum, or a flonum respectively.
- `($+ x y)` is an addition function in the runtime system that handles all possible type combinations for `x` and `y`, including those that raise an exception.
- `($fl+ x y)` adds two flonums to give a flonum result.
- `($fx+? x y)` adds two fixnums to give a fixnum result or false in the case of an overflow. This operation includes an overflow check that makes it somewhat more expensive than `($fx+ x y)` that does not check for overflow (note the absence of the trailing '`?`').

When the CFG of the above expansion of `(+ x 1)` is processed by the SBBV algorithm various optimizations can happen. If the context indicates that `x` is a fixnum then the `($fixnum? x)` test in the specialized basic block is an unconditional jump to the CFG of `(or ($fx+? x 1) ($+ x 1))`, effectively removing the code that handles flonums, bignums, and other types. Moreover, if it is known that `x` is in the interval LO..HI where HI $<$ maxfix, then the result of `($fx+? x 1)` is in the fixnum interval LO$+1$..HI$+1$, so an overflow is impossible, *i.e.*, `($fx+? x 1)` is necessarily a fixnum. Consequently the CFG can be specialized to `($fx+ x 1)`, which is a machine integer addition with no overflow check.

Other languages, such as Ruby, support similar generic arithmetic, but more importantly, languages such as JavaScript [16] and Python that do not expose small integers require that the compiler be able to detect when operations can be implemented as fixnum operations. Hence, these fast implementations must be able to detect when an operation overflows and, exactly as Scheme does, promote the number in such a case (JavaScript promotes them to IEEE floating point numbers, Python to bignums).

To extend the analysis, we refine the definition of specialization contexts (Figure 4) and we refine the lattice of values (Figure 5). Fixnum values are now represented with intervals.

Handling numerical values requires us to modify the specialization of the `if` and the *fixnum arithmetic with overflow* instructions, such as `$fx+?`. The first one must compute new interval approximations to be propagated in the positive and negative branches by using interval narrowing techniques [11]. The second one must implement arithmetic operations over intervals [23], such as $x_{\text{LO}}..x_{\text{HI}} + y_{\text{LO}}..y_{\text{HI}} = (x_{\text{LO}} + y_{\text{LO}})..(x_{\text{HI}} + y_{\text{HI}})$.

Algorithm 10 Specialization of `if` instructions with numerical values.

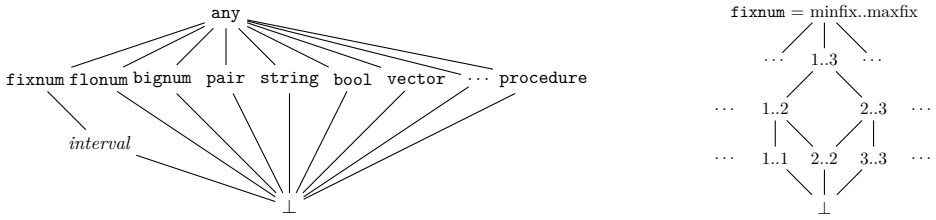
```
1: function InsSpecializeIf(i, ctx, wq)
2:     if i is an integer comparison then
3:         (ctx+, ctx−) ← intervalNarrowing(i, ctx)
4:         ni ← i.dup()
5:         ni.target ← BlockNewVersion(i.target, ctx+, wq)
6:         return (ni, ctx−)
7:     else
8:         as in algorithm 8
```

For instance, let us assume the specialization of the instruction "`if ($fx> i 3)`" in a context $\{\texttt{i} \mapsto \text{minfix..10}\}$. The new InsSpecializeIf will generate the two new contexts $\{\texttt{i} \mapsto 4..10\}$ and $\{\texttt{i} \mapsto \text{minfix..3}\}$ for the positive and negative outcomes respectively.

The numerical operation specialization replaces a numerical operator, such as `$fx+?`, whose result is known not to overflow with a faster operator that does not check for overflow, such as `$fx+`. It must also implement some widening operation [10] in order to hasten the convergence of the algorithm. Without widening the algorithm would require a number of iterations proportional to the size of the interval representing numbers, which would be prohibitive. The widening is handled by the $\triangledown$ operator of the Algorithm 4 (see Section 2.2).

context: ctx, ctx_0, ...
- types: a list of mappings of variable to type information
- equiv: a list of equivalence classes
- range: **a list of mappings of variable to integer intervals**

Figure 4 Specialization contexts extended with integer intervals.



(a) extended lattice. **(b)** *interval* lattice.

Figure 5 Extending the lattice used for variable values to handle integer intervals.

3.3 Vector Support

The previous section showed how to extend SBBV to approximate integer values as intervals described with two integer bounds. While this enables the compiler to remove overflow checks, it is insufficient to remove bound checks of vector accesses as the length of a vector is generally unknown statically and cannot be represented with an exact integer value.

To handle vectors we introduce a refined representation of intervals. With this extension interval bounds can be represented with integers as before, but also with the symbolic value $[\![v]\!]-i$ that is equal to the length of vector v minus the integer offset $i \geqslant 0$. Here we assume that the length of a vector is always a nonnegative fixnum. For instance, the interval $0..[\![v]\!]-1$ denotes all the nonnegative fixnums that are lower than the length of the vector `v`, in other words, the valid indexes of vector `v`. The upper bound of that interval, *i.e.*, $[\![v]\!]-1$, itself denotes a fixnum in the interval $-1..\text{maxfix}-1$.

We illustrate the benefit of this extension to SBBV with a variant of the `find` function presented in Section 2 that operates on vectors and lists:

```
1  (define (findv p x)
2    (if (vector? x)
3        (let ((len (vector-length x)))
4          (let loop ((i 0))
5            (if (< i len)
6                (let ((e (vector-ref x i)))
7                  (if (p e)
8                      e
9                      (loop (+ i 1))))
10               #f)))
11       (find p x)))
```

```
(define (findv p x)
  (if ($vector? x)
      (let ((len (if ($vector? x) ($vector-length x) ($fail))))
        (let loop ((i 0))
          (if (if (and ($fixnum? i) ($fixnum? len))
                   ($fx< i len)
                   (if (and ($flonum? i) ($flonum? len))
                       ($fl< i len)
                       ($< i len)))
              (let ((e (if (and ($vector? x) ($fixnum? i)
                                ($fx>= i 0) ($fx< i ($vector-length x)))
                           ($vector-ref x i)
                           ($fail))))
                (if ((if ($procedure? p) p ($fail)) e)
                    e
                    (loop (if (and ($fixnum? i) ($fixnum? 1))
                              (or ($fx+? i 1) ($+ i 1))
                              (if (and ($flonum? i) ($flonum? 1))
                                  ($fl+ i 1)
                                  ($+ i 1))))))
              #f)))
      (find p x)))
```

Figure 6 The code of the `findv` function where all dynamic checks are explicit.

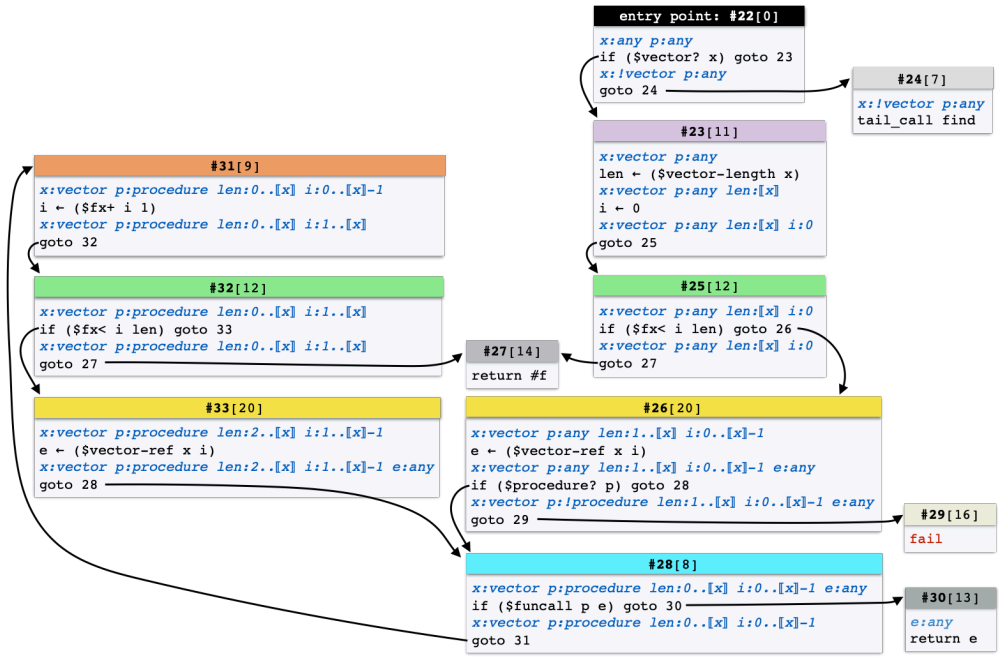


Figure 7 The specialized CFG of the `findv` function showing that the index calculations are done entirely with fixnums with no overflow checks and no vector bound checks. For brevity, we write ⟦x⟧ instead of ⟦x⟧−0, and the interval ⟦x⟧..⟦x⟧ is abbreviated to ⟦x⟧.

Figure 6 shows the code after the compiler has blindly expanded each operation to include all required dynamic checks. The refinements of SBBV presented in this section enables the compiler to create the specialized CFG shown in Figure 7, which is optimal (all bound checks and overflow checks have been removed). There is only a procedure check for parameter `p` in the first iteration of the loop, and only if parameter `x` is a non-empty vector.

This refinement does not impact the SBBV algorithm nor does it demand to change the specialization of the arithmetic instructions but it requires us to extend the interval operators to treat cases where at least one bound is a symbolic value. For example, in the case of the interval addition $x_{\text{LO}}..x_{\text{HI}} + y_{\text{LO}}..y_{\text{HI}} = (x_{\text{LO}} + y_{\text{LO}})..(x_{\text{HI}} + y_{\text{HI}})$, the addition of the lower bounds $(x_{\text{LO}} + y_{\text{LO}})$ is computed from the following rules, where i and j denote integer values, and v and w denote vector identifiers:

$$\begin{array}{rcl} i +_{\text{LO}} j & \mapsto & i + j \\ (\llbracket v \rrbracket - i) +_{\text{LO}} j & \mapsto & j - i \\ j +_{\text{LO}} (\llbracket v \rrbracket - i) & \mapsto & (\llbracket v \rrbracket - i) +_{\text{LO}} j \\ (\llbracket v \rrbracket - i) +_{\text{LO}} (\llbracket w \rrbracket - j) & \mapsto & -i - j \end{array}$$

and the addition of the upper bounds $(x_{\text{HI}} + y_{\text{HI}})$ is computed from the following rules, where *overflow* denotes an upper bound that is not a fixnum:

$$\begin{array}{rcll} i +_{\text{HI}} j & \mapsto & i + j & \\ (\llbracket v \rrbracket - i) +_{\text{HI}} j & \mapsto & \llbracket v \rrbracket - (i - j) & \text{if } i \geqslant j \\ (\llbracket v \rrbracket - i) +_{\text{HI}} j & \mapsto & \textit{overflow} & \text{if } i < j \\ j +_{\text{HI}} (\llbracket v \rrbracket - i) & \mapsto & (\llbracket v \rrbracket - i) +_{\text{HI}} j & \\ (\llbracket v \rrbracket - i) +_{\text{HI}} (\llbracket w \rrbracket - j) & \mapsto & \textit{overflow} & \end{array}$$

The narrowing operations for comparisons are similar to that of regular intervals, considering that vector lengths are themselves modelled as intervals from 0..maxfix. Below are the rules for the narrowing of $x < y$ from which the rules for the other comparisons can be derived:

Narrowing rule for: $x < y$ with $\{x \mapsto x_{\text{LO}}..x_{\text{HI}}, y \mapsto y_{\text{LO}}..y_{\text{HI}}\}$

Positive outcome: $\{x \mapsto x_{\text{LO}}..\min_{\text{HI}}(x_{\text{HI}}, y_{\text{HI}} - 1), y \mapsto \max_{\text{LO}}(x_{\text{LO}} + 1, y_{\text{LO}})..y_{\text{HI}}\}$
Negative outcome: $\{x \mapsto \max_{\text{LO}}(x_{\text{LO}}, y_{\text{LO}})..x_{\text{HI}}, y \mapsto y_{\text{LO}}..\min_{\text{HI}}(x_{\text{HI}}, y_{\text{HI}})\}$

$$\begin{array}{rcl} \max_{\text{LO}}(x, y) & \mapsto & x \textbf{ if } \text{val}_{\text{LO}}(x) > \text{val}_{\text{LO}}(y) \textbf{ else } y \\ \min_{\text{HI}}(x, y) & \mapsto & x \textbf{ if } \text{val}_{\text{HI}}(x) < \text{val}_{\text{HI}}(y) \textbf{ else } y \end{array}$$

$$\begin{array}{rcl} \text{val}_{\text{LO}}(i) & \mapsto & i \\ \text{val}_{\text{LO}}(\llbracket v \rrbracket - i) & \mapsto & i \end{array}$$

$$\begin{array}{rcl} \text{val}_{\text{HI}}(i) & \mapsto & i \\ \text{val}_{\text{HI}}(\llbracket v \rrbracket - i) & \mapsto & \text{maxfix - } i \end{array}$$

It is noteworthy that, as a result of the interval widening, some of the inferred intervals are somewhat conservative (for example at block `#28` the interval for `len` could have been $1..\llbracket v \rrbracket$). The widening loses some information but makes computing a fix-point faster.

4 Experiments

In this section we demonstrate the practicality of SBBV through experiments. To ensure that our results are not overly system specific, we have integrated an SBBV pass in the compilation pipeline of two existing Scheme compilers, Bigloo [19] and Gambit [20]. Both

of these are independently developed mature optimizing AOT Scheme to C compilers that use a CFG representation of the compiled program. Moreover these compilers are used as back-ends of optimizing compilers for JavaScript [31, 30] and Python [22]. Bigloo and Gambit implement a slew of features and classical optimizations such as constant-folding, function inlining, flat closures, and lambda-lifting. Any performance improvements would constitute a notable achievement given the many years of fine-tuning that went into their development. We put this in perspective in Section 4.5.

Adding SBBV to these compilers allows them to use the type, range and value of variables to perform flow sensitive code specialization that is tailored to the program logic, with low code bloat. The experiments have been designed to demonstrate further performance improvements by SBBV than by other optimization techniques implemented by these compilers.

We evaluate the impact of SBBV by applying it to a suite of benchmarks. Each benchmark is compiled with and without SBBV to measure its impact on the number of dynamic checks, program size, execution time, and compilation time. Section 4.1 provides a brief description of the benchmark suite. Benchmarks were executed on a machine with an Intel Core i7-7700K, 48 GB of RAM, and under Debian 10.13 with kernel version SMP Debian 4.19.269-1.

In order to measure SBBV's impact on the number of dynamic checks, both compilers have been instrumented to count the number of dynamic checks during a program execution. Executions for measuring time and dynamic checks are done separately to ensure that counting checks does not affect the measured execution time. Execution time is measured by profiling each executable with "`perf stat`" to measure its execution real-time. Each benchmark is parameterized such that its execution lasts at least five seconds on our machine, and is repeated 50 times, removing the top and bottom 5 outliers. The parameters of each benchmark are provided as command-line arguments to ensure that the compiler does not optimize for specific values or types. All timing results in this section are the average execution time of each benchmark. The relative standard deviation of the execution time never exceeds 0.24% on macrobenchmarks, and 2.20% on microbenchmarks; consequently we omit standard deviations in figures to improve readability.

4.1 Benchmark Programs

Our benchmark suite combines programs from two sources: the R7RS benchmark suite [1] that is commonly used for evaluating the performance of Scheme systems, and benchmark programs used in [30] that have Scheme and JavaScript versions.

We use both macrobenchmarks and microbenchmarks, which we classify according to their size (fewer than 150 lines of code is a microbenchmark). These two classes are distinguished because microbenchmarks stress a narrow set of features and consequently are poor predictors of the overall performance of a system. We only use microbenchmarks as instruments for shedding light on specific behaviors of the SBBV algorithm.

Here is a brief description of the benchmark programs:

Macrobenchmarks:

- `almabench` (430 LOC): Compute the celestial coordinates of the sun at noon. Uses floating point numbers, vectors, and assignments.
- `boyer` (610 LOC): Prolog-like rule-directed rewriting engine. Uses pairs and symbols.
- `compiler` (11,740 LOC): Old version of the Gambit Scheme compiler generating M68000 code. Uses pairs, symbols, vectors, and strings.
- `conform` (490 LOC): Graph type checker using equivalence classes. Uses lists and strings.

- `dynamic` (2,350 LOC): Dynamic type inference for Scheme. Uses lists, symbols, and higher-order functions.
- `earley` (660 LOC): Earley parser parsing an ambiguous grammar. Uses vectors, lists, small integers and symbols.
- `leval` (560 LOC): Scheme interpreter based on closures. Uses lists, symbols, and higher-order functions.
- `maze` (740 LOC): Hexagonal grid maze generator. Uses vectors and small integers.
- `nucleic` (3,510 LOC): 3D structure determination of a nucleic acid. Uses vectors and floating point numbers.
- `peval` (630 LOC): Partial evaluator for Scheme. Uses lists, symbols, and higher-order functions.
- `scheme` (1,090 LOC): Other Scheme interpreter based on closures. Uses lists, symbols, and higher-order functions.
- `slatex` (2,470 LOC): Scheme to Latex processor. Uses characters, strings, lists, vectors, and small integers.

Microbenchmarks:

- `ack` (10 LOC): Ackermann function. Uses small integers and recursion.
- `bague` (110 LOC): Solver of the *baguenaudier* puzzle. Uses small integers and vectors.
- `fib` (20 LOC): Fibonacci function. Uses small integers and recursion.
- `fibfp` (20 LOC): Fibonacci function. Uses floating point numbers and recursion.
- `nqueens` (40 LOC): Solver of the N-queens puzzle. Uses lists, small integers, and recursion.
- `primes` (40 LOC): Sieve algorithm for finding primes. Uses lists and small integers.
- `tak` (20 LOC): Takeuchi function. Uses small integers and recursion.

4.2 Counting Dynamic Checks

In the Bigloo and Gambit implementations, many built-in procedures implicitly check the type of their arguments and signal an error if they are invalid, such as arithmetic on non-number types or index out of bound when indexing a vector. Polymorphic operators also use implicit dynamic type checks to dispatch computation to specialized primitives.

To measure dynamic checks, all built-in procedures used in our benchmarks are redefined with macros that use inline checks. This is semantically equivalent to operations that apply checks and dispatch to specialized primitives implicitly. For instance, the `BBVvector-ref` and `BBV+` macros implement the `vector-ref` and `+` operations respectively:

```
(define-macro (BBVvector-ref v i)
  `(let ((v ,v) (i ,i))
      (if (and ($vector? v) ($fixnum? i)
               ($fx>= i 0) ($fx< i ($vector-length v)))
          ($vector-ref v i)
          (error "vector-ref error")))))

(define-macro (BBV+ x y)
  `(let ((x ,x) (y ,y))
      (if (and ($fixnum? x) ($fixnum? y))
          (or ($fx+? x y) ($+ x y))
          (if (and ($flonum? x) ($flonum y))
              ($fl+ x y)
              ($+ x y)))))
```

The macros use specialized operators (prefixed with `$` in the example), making all checks explicit (type checks, array bound checks, and integer overflow checks) and ensuring that both compilers perform the same set of checks and in the same order (see Section 3.2 for the definitions of `$fx+?`, `$+`, and the other primitive operations).

When SBBV has determined the type of a value, the type tests that are in the expansion of these macros (`$fixnum?`, `$flonum?`, etc) are effectively removed. Similarly, SBBV's interval analysis may determine the range of possible integer values, allowing comparisons such as `($fx>= i 0)` to be removed, and calls to overflow checking operators such as `($fx+? x y)` to be replaced by the non-overflow checking `($fx+ x y)` when the result cannot overflow.

However, not all dynamic checks originate from safe operators since programmers can add type tests and bound checks as part of their program's logic. For this reason, we distinguish between checks introduced by safe operators, which we call *safety* checks in the context of this experiment, and those introduced by the programmer. The number of safety checks is computed by replacing all operators, such as `BBVvector-ref` and `BBV+`, by unsafe ones that execute no type tests, overflow checks or array bound checks. By subtracting the number of checks executed by the unsafe version of a program from the number of checks of its safe version, we obtain the number of safety checks.

4.3 Merge Selection

Applying SBBV requires a heuristic for selecting which versions of a block to merge when that block's version limit is exceeded. This corresponds to choosing a concrete implementation for the Θ^2 operator from Algorithm 4. The quality of the selection function impacts the general performance of the SBBV algorithm. The current Bigloo and Gambit implementations use a merge heuristic that is rudimentary but still sufficient to establish the benefit of the approach.

In our implementations, when the version limit of a basic block is exceeded, versions that are the most similar are merged first. The similarity of two versions is computed by counting how many variables have the same type when entering the block. While both Bigloo and Gambit implement a *selection by similarity*, the exact implementations of their selection functions differ slightly due to how they represent types internally.

Given that the versions selected depend solely on the contexts when entering a block, this is a *local* merge heuristic. It requires no usage analysis of each variable in the block and its successors. A nonlocal merge heuristic could lead to better results if it prioritizes some versions over others with the objective of maximizing the benefits for the whole program. The space of possible merge heuristics is large and we intend to explore it in future work.

4.4 Results

We first present broad results before diving into a deeper analysis in the subsequent sections.

SBBV permits a trade-off between code size and dynamic checks removal. As the version limit increases, more blocks are duplicated and more dynamic checks are removed. This comes at the cost of increased executable size and compilation time. We found a limit of 2 versions to offer a good trade-off between checks removal, size, and compilation time.

Figure 8 shows the proportion of safety checks removed by SBBV, such as type, overflow, and array bound checks. In all benchmarks, the number of checks decreases when compared to the executable without SBBV. This indicates that SBBV can remove dynamic checks that the existing optimizations of Bigloo and Gambit could not remove.

The proportion of checks removed varies between Bigloo and Gambit, despite both compilers applying the same SBBV algorithm. Bigloo removes more checks without applying SBBV, thus leaving fewer checks to be removed by SBBV. However, the absolute number of remaining checks is similar between both implementations. To a lesser extent, differences in the implementations of the heuristic for selecting versions to merge also influence the number of removed checks by each compiler.

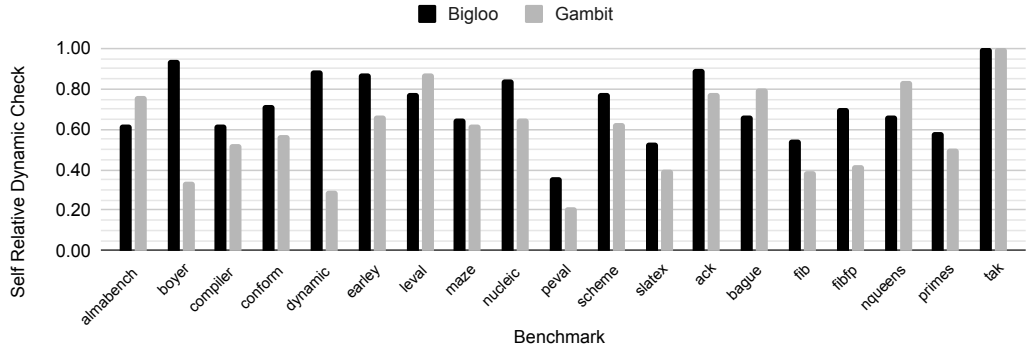


Figure 8 Relative number of safety checks executed for benchmarks compiled with and without SBBV (limit of 2 versions), separately for Bigloo and Gambit. 1.0 corresponds to compilation without SBBV. Lower values indicate fewer dynamic checks with SBBV.

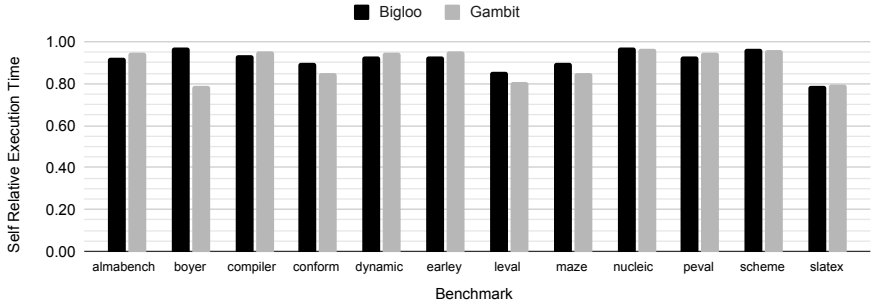


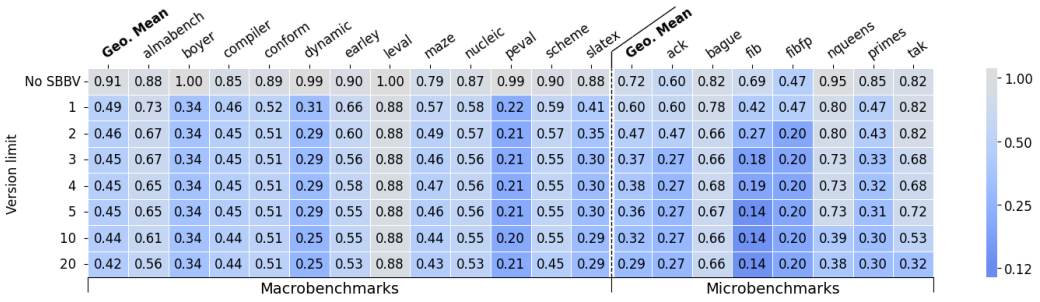
Figure 9 Relative execution time of macrobenchmarks compiled with and without SBBV (limit of 2 versions), separately for Bigloo and Gambit. 1.0 corresponds to compilation without SBBV. Lower values indicate better performance with SBBV.

Although the number of dynamic checks decreases with higher version limits, it does not always lead to a similar reduction of the execution time. Figure 9 shows that all macrobenchmarks execute faster with SBBV and a limit of 2 version (by 10% on average). However, no significant speedup is observed by further increasing the version limit. The relation between the version limit and execution time is discussed further in Section 4.4.3.

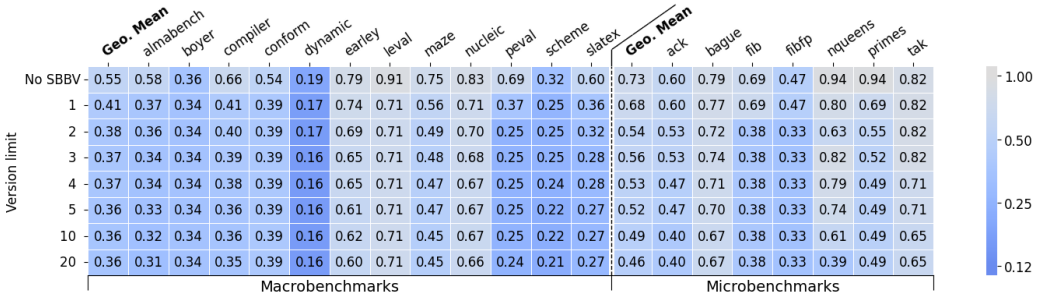
Increasing the version limit allows for more specialized versions. In the following sections, we take into account the impact of the version limit on the removal of dynamic checks, program size, execution speed, and compilation time. Each benchmark is compiled with version limits ranging from 1 to 5, as well as with limits of 10 and 20 versions, and it is compared to a compilation without SBBV. Limits higher than 5 are probably not very practical due to diminishing returns for the added compilation time. We tested with limits of 10 and 20 versions mostly to check the performance in extreme cases.

4.4.1 Dynamic Checks

Figure 10 shows the proportion of safety checks remaining after SBBV with increasing version limits. We estimate the number of remaining safety checks by subtracting checks in the unsafe version of a benchmark from those in the benchmark compiled with SBBV. To compute the total number of safety checks without SBBV, we apply the same formula to each benchmark compiled with no optimization, which effectively preserves all checks.



(a) Gambit.



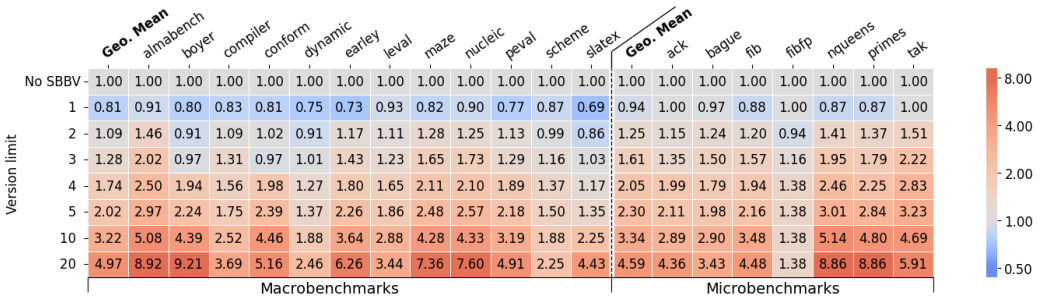
(b) Bigloo.

Figure 10 Effect of SBBV on the removal of dynamic checks (type, overflow, and array bound checks) with increasing version limits. Each cell shows the proportion of safety checks remaining for a given version limit and benchmark when compared to an unoptimized execution. The first row shows checks when SBBV is not applied and only existing optimization techniques are used. A ratio of 1 means that no dynamic checks were removed. Lower values indicate that more checks were removed.

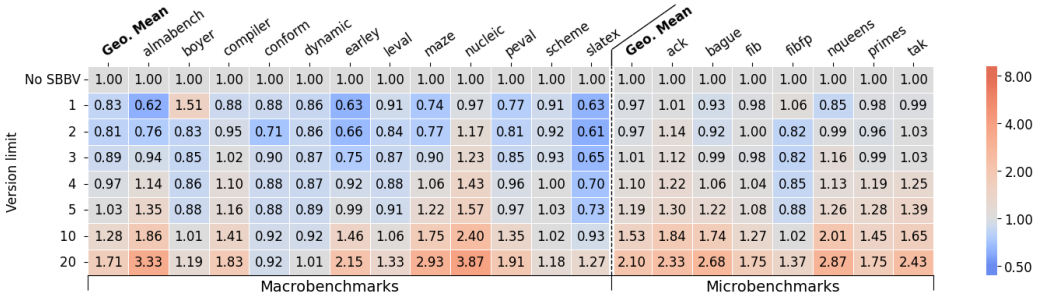
Figures 10a and 10b show, on the first row, the proportion of remaining checks after applying the standard Gambit and Bigloo optimization techniques (*No SBBV*). The following rows display results with SBBV and specific version limits. For all benchmarks, SBBV removes more checks than the standard optimizations. As the version limit increases, more specialized versions are generated, allowing removal of additional checks.

For low version limits, the number of dynamic checks steeply decreases as the limit increases. However, beyond a version limit of about 4, there is a diminishing return for almost all benchmarks. In some benchmarks, an upper bound is rapidly reached beyond which almost no more checks are removed (such as `boyer` at 1 version). In these cases, further increasing the limit contributes to the generation of relatively unimportant versions. Conversely, new useful versions are still discovered when increasing the version limit beyond 10 for some benchmarks (such as `tak` with Gambit).

Increasing the version limit sometimes increases the number of dynamic checks. For instance, in Figure 10a, the number of dynamic checks increases when incrementing the version limit from 2 to 3 in the `nqueens` benchmark with Bigloo. Increasing the version limit delays the merge of excess versions until more candidates are discovered. Given additional choices, the selection function may choose differently, merging a useful version that would have been kept otherwise. This highlights the room for improvement of our selection function.



(a) Gambit.



(b) Bigloo.

Figure 11 Program size with increasing version limits, relative to using the standard optimizations (*No SBBV*). Each cell shows the ratio between the program size with and without SBBV for a given version limit and benchmark. Lower values are better.

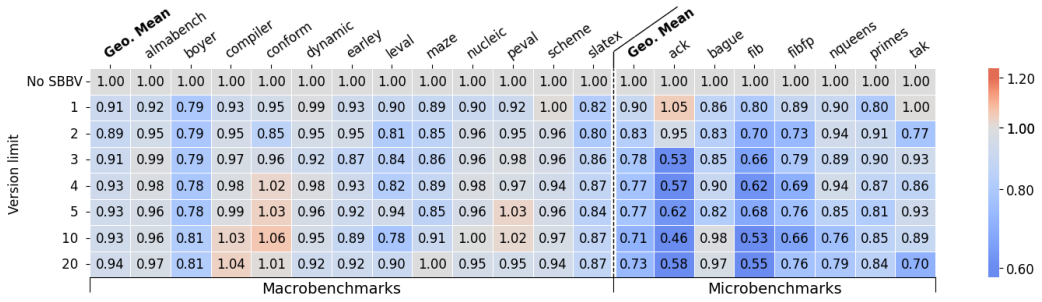
4.4.2 Program Size

We measured the program size of benchmarks compiled with SBBV. The size in bytes of each benchmark is obtained by disassembling its executable and subtracting the position of compiler specific labels. Hence, only the size of the code corresponding to each benchmark, excluding any runtime procedures, is considered.

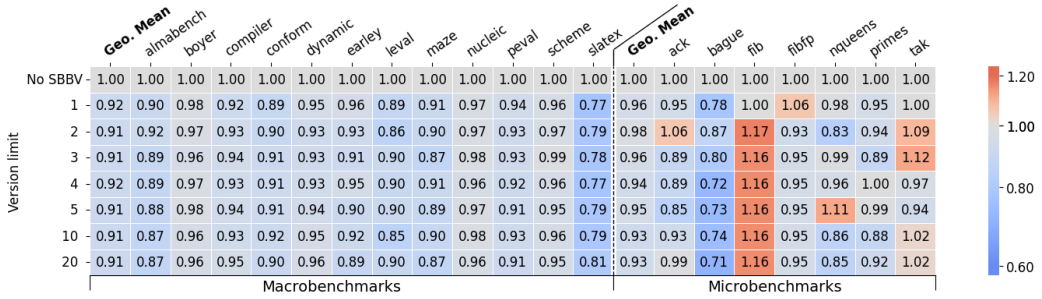
Since SBBV applies code duplication, higher version limits generally generate larger programs. Yet, low version limits may result in smaller executables. In the case of a limit of a single version, this is to be expected because SBBV becomes akin to a static type inference analysis without code duplication, which removes some unnecessary checks. However, for low enough version limits higher than one, SBBV can still reduce program size. In these cases, the removal of dynamic checks outweighs the duplication of basic blocks.

Figure 11b shows the relation between the size of a benchmark and the allotted version limit in Bigloo. On average, macrobenchmarks are smaller up to a limit of 5 versions. In general, the size increases with the version limit, but remains reasonably low with an average growth of about 1.7× on macrobenchmarks with a limit as high as 20 versions.

Figure 11a shows a similar pattern in Gambit, but with a higher growth rate. In the worst case, a growth of about 10× is observed (`boyer`, limit of 20 versions), highlighting the need to select a low enough version limit to curb code bloat. We found a version limit ranging from 2 to 4 to be a good compromise between removed dynamic checks and size.



(a) Gambit.



(b) Bigloo.

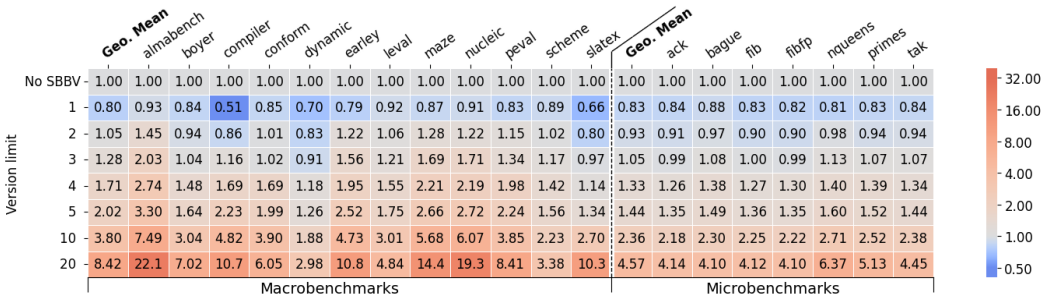
Figure 12 Execution time with SBBV and increasing version limits, relative to the standard optimizations (*No SBBV*). Each cell shows the ratio between the execution time with and without SBBV for a given version limit and benchmark. Lower values are better.

4.4.3 Execution Time

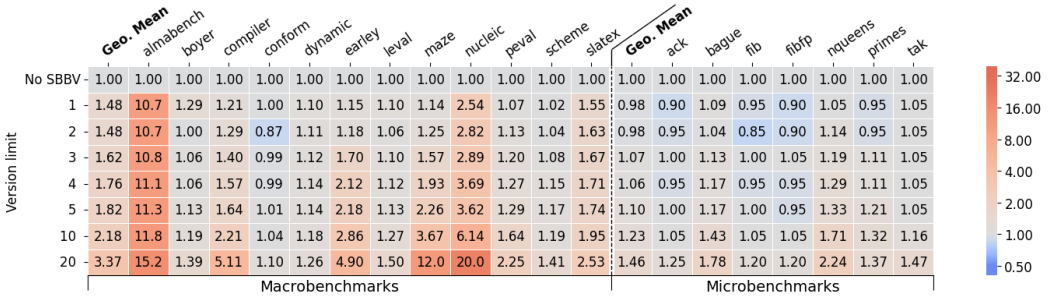
Figure 12 shows how the execution time of each benchmark varies with the version limit. Comparing performance to the number of dynamic checks from Figure 10 shows that a lower number of checks is not correlated to a faster execution in general. With Bigloo (Figure 12b), macrobenchmarks compiled with SBBV execute, on average, about 10% faster than without SBBV regardless of the version limit. With Gambit (Figure 12a), a similar speedup is observed for version limits below 4. Beyond this limit, execution speed still benefits, albeit to a lesser extent. We suspect that this discrepancy is caused by the increased code bloat observed with Gambit for high version limits, which reduces the performance of the instruction cache.

While some benchmarks benefit from a high version limit, the code speed and version limit is only vaguely correlated and contains noise. When optimizing for code speed, using a default version limit for all programs is suboptimal and it is good to give the programmer a manual control over the limit to explore the tradeoffs.

We explain this in part by hard-to-predict hardware optimizations by modern processor architectures. In particular, branch prediction makes dynamic type checking extremely cheap in typical code where a type check frequently returns the same result. Moreover, Bigloo and Gambit implement inexpensive type checks using pointer tagging. We hypothesize that SBBV would have a higher performance impact in implementations with more costly dynamic checks, such as NaN tagging, or object representations that need a memory access to check the type, such as BiBOP and object-oriented languages such as Java, Python, and Ruby.



(a) Gambit.



(b) Bigloo.

Figure 13 Compilation time with increasing version limits, relative to using the standard optimizations (*No SBBV*). Each cell shows the ratio between the compilation time with and without SBBV for a given version limit and benchmark. Lower values are better.

This highlights the need to refine the merge selection function. In the future, we intend to explore the space of possible merge heuristics, including nonlocal heuristics. We also wish to explore dynamic version limits, for instance by increasing the version limit of basic blocks that are likely to be in megamorphic code, as is done by YJIT [7, 8]

4.4.4 Compilation Time

We measured the compilation time for each benchmark and version limit and compared it to the compilation time without SBBV. Figure 13 shows the effect of the version limit on compilation time. In general, compilation time increases with the version limit. The reasons for this increase are twofold: firstly a higher version limit leads to more specialized versions of basic blocks within a control flow graph, secondly the increased size of the C code generated by Bigloo and Gambit leads to increased compilation time by the C compiler. Consequently, a lower program size is correlated to a shorter compilation time.

In the extreme case of a limit of 20 versions, compilation time increased by about 22× in the worst case (`almabench` with Gambit). However, in Section 4.4.1 we showed that there is a diminishing return from increasing the limit beyond about 4 versions. Choosing a limit of 2 caps the worst observed compilation time increase to about 1.5× with Gambit (`almabench`) while reaping most of the benefit of SBBV. On the same benchmark, Bigloo has a large compilation time increase starting at a limit of 1. This is due to the combined effect of a large function with multiple dispatch points and a live variable recalculation by the compiler that is not done by Gambit (register allocation is done before SBBV in the case of Gambit, and after SBBV in the case of Bigloo).

Program	Node.js 21.7.1	Bigloo 4.6a		Gambit 4.9.4-377		Chez 9.5.1	Racket 7.2
		No SBBV	SBBV	No SBBV	SBBV		
`almabench`	**6.31**	15.68	14.46	14.64	13.86	17.69	19.30
`boyer`	40.31	7.40	7.22	8.21	**6.49**	8.09	12.38
`earley`	56.75	14.90	13.83	10.87	10.34	**9.53**	24.73
`leval`	18.42	7.53	**6.47**	12.03	9.74	7.16	16.96
`maze`	10.07	7.47	6.70	6.75	**5.73**	12.01	12.12
`bague`	305.04	12.90	**11.19**	15.54	12.93	21.01	19.33

Figure 14 Average execution times of the benchmarks from [30] in seconds. Bold numbers indicate the fastest execution time for each benchmark. Lower is better.

4.5 Putting the Results in Context

The significance of the above results can be best appreciated through a comparison with other systems whose performance is more widely known. Our goal is to show that both Gambit and Bigloo are competitive with some of the leading implementations of dynamically typed languages, and thus that using SBBV is attractive in the context of high-performance implementations to increase performance further.

The Node.js system is a good comparable given that the JavaScript V8 JIT compiler on which it is based has been extensively engineered and it executes a dynamically typed programming language with similar core constructs as Scheme, to which prototype object-oriented features are added. The Chez Scheme [26] and Racket [25] systems are also interesting as high-performance representatives of the Lisp/Scheme family.

In order to do a fair comparison of systems for different languages, we use benchmark programs that have been translated to both Scheme and JavaScript in previous work [30]. Those programs are a subset of those used in the previous sections. Because of the similarity of JavaScript and Scheme, a systematic translation of the constructs is possible while avoiding important stylistic changes that would compromise the validity of the comparison. We have used the latest versions of those systems available at the time of writing through the Debian package manager (Node.js 21.7.1, Chez Scheme 9.5.1, and Racket 7.2). We measure the program execution time and, for Gambit and Bigloo, we compile the program without SBBV and with SBBV and a limit of 2 versions. Figure 14 gives the execution times in seconds.

The execution time for Gambit and Bigloo without SBBV is faster than Node.js on all the benchmarks except `almabench`, up to 24× faster for `bague`. Node.js does better on `almabench` than any other system because V8 has special optimizations for floating point numbers and arrays that are used by `almabench`. Gambit and Bigloo without SBBV are in the same ballpark as Chez Scheme, faster on roughly half the benchmarks. Racket is typically slower than Chez Scheme, on which it is built internally. When SBBV is used, both Bigloo and Gambit are consistently faster than without SBBV, including on benchmarks on which they already outperformed other compilers. This reinforces our belief that SBBV allows some optimizing compilers for dynamically typed programming language to generate even better code.

5 Related Work

Basic Block Versioning (BBV) was introduced by Chevalier-Boisvert and Feeley [5] as a technique for type check removal in JavaScript. The *lazy* BBV variant, suitable for JIT compilers, only generates versions that are executed, so it limits code bloat. On the other

hand, when the version limit is reached a fully generic version must be used, which negatively impacts type check reduction. SBBV avoids falling back on a generic version by selecting a set of versions that cover all cases but that are at least somewhat specialized. As shown in their Figure 7, a version limit of 2, which is their setting *maxvers=1*, shows a modest reduction of type checks for many benchmarks when compared to *maxvers=5* because the fallback on the fully generic version is reached too quickly. SBBV achieves good performance with lower version limits. A variant of lazy BBV is used in production in the YJIT compiler inside CRuby [7, 8], also falling back on a fully generic version upon reaching the version limit (which can be 4, 10, or 20 depending on the situation).

The *eager* BBV variant described in [5] is suitable for AOT compilers, like SBBV, but suffers from large code bloat so it was deemed impractical and not explored further [5]. In comparison, SBBV with a version limit of 2, which achieves a comparable level of dynamic check removal, causes an average code size increase of 9% for Gambit and a decrease of 19% for Bigloo.

SBBV can be explained by the theory of abstract interpretation introduced by Cousot and Cousot [10]. The authors presented a theoretical framework for building lattice-based fixed point algorithms. Their work ensures that the SBBV algorithm converges. Furthermore, their seminal paper introduced *union with widening*. Widening not only ensures that the algorithm converges, but also that it converges fast enough to be practical.

SBBV generalizes well-known optimization techniques such as loop-unrolling [2], constant-folding [2], and tail duplication [24]. Tail duplication replicates the code after conditional branches instead of merging the control flow to a single basic block. This permits propagating the information acquired from a condition beyond the body of each branch, allowing further optimizations.

Determining when to apply tail duplication remains a challenge. Leopoldseder *et al.* proposed a simulation-based approach to determine which duplications are the most promising in term of optimization opportunities while minimizing code size [18]. This is analogous to how the choice of a selection function impacts the efficiency of SBBV.

More recently, D'Souza *et al.* applied tail duplication in TASTyTruffle, a Scala JIT compiler using Truffle. TASTyTruffle performs tail duplication at the AST level to generate guarded versions of polymorphic functions that are then specialized by the Graal compiler [13].

Our work is related to occurrence typing that is in particular used in Racket [33]. Occurrence typing is a type system that allows a context-sensitive refinement of variable types during static analysis, for instance by typing a variable differently in each branch of a conditional statement. In the context of our work, occurrence typing synergizes well with tail duplication to further propagate context-sensitive type information.

Partial function inlining can also be done if SBBV is applied interprocedurally. We have not done this in the current work because it is tricky to extend the versioning contexts to track code pointers for function entry and return points. This will require future work.

Code optimization by duplication and specialization has been extensively studied and used for various programming languages [12, 3, 29, 4, 9]. Recently Flückiger *et al.* [15] optimized the compilation of R programs by function specialization. They show that this technique makes programs run 1.7× faster on average. Subsequently a study by Mehta *et al.* [21] has used a mechanism that keeps multiple versions of a given function specialized for contexts encountered at runtime by JIT compilers. Specialized versions of a function are stored in an external repository, allowing switching between versions when de-optimization occurs and to reuse versions of functions across executions and programs. SBBV is related to these techniques but the granularity of cloning is finer as it clones basic blocks while all

those previous works clone whole function bodies. This enables SBBV to better control the code expansion and to spend the cloning budget on relevant specializations without falling back on a fully generic version or using de-optimization, which is not an option in an AOT context. This sort of fine optimization tuning is out of reach for techniques that specialize at the level of whole function bodies.

6 Conclusion

Previous work has shown that the lazy variant of Basic Block Versioning (BBV) is effective in practice for optimizing dynamic checks in JIT compilers for dynamic languages [5, 27, 7, 8]. The *static* BBV (SBBV) approach that we have described in this paper is a variant of BBV that determines, through a fix-point program analysis, a set of basic block versions that are appropriate for the program and that covers all possible contexts without exceeding some versioning limit. This gives control over the code bloat induced by the multiple specializations of individual basic blocks in a way that avoids falling back on an unoptimized generic version of the basic block when the versioning limit is reached. SBBV is thus particularly interesting for use in AOT compilers and consequently it does not suffer from a warmup time.

At the core of the SBBV algorithm is a heuristic to drive the merging of previously generated versions to keep the number of versions within the allowed limit. We have shown through experiments that even a simple merge heuristic removes dynamic checks effectively in practice.

As a second contribution, we have shown in this paper how to extend the BBV approach to implement optimizations that go beyond the elimination of dynamic type checks. We have shown how to use it to remove integer overflow checks and bound checks effectively. By doing so, we have shown that the BBV approach can be viewed as a general programming analysis methodology that can be used to implement various optimizations that otherwise are implemented in isolation using dedicated techniques.

References

1 R7RS benchmarks. `https://github.com/ecraven/r7rs-benchmarks`, April 2024.

2 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. `doi:10.1145/197405.197406`.

3 Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA):120:1–120:23, 2018. `doi:10.1145/3276490`.

4 Craig Chambers and David M. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pages 146–160. ACM, 1989. `doi:10.1145/73141.74831`.

5 Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 101–123. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPICS.ECOOP.2015.101`.

6 Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 7:1–7:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPICS.ECOOP.2016.7`.

7 Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. *YJIT: a basic block versioning JIT compiler for CRuby*, pages 25–32. ACM, 2021. doi:10.1145/3486606.3486781.

8 Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. Evaluating YJIT's performance in a production context: a pragmatic approach. In Rodrigo Bruno and Eliot Moss, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, pages 20–33. ACM, 2023. doi:10.1145/3617651.3622982.

9 Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117, 1993. doi:10.1016/0096-0551(93)90005-L.

10 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

11 Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In David B. Wortman, editor, *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS), Raleigh, North Carolina, USA, March 28-30, 1977*, pages 77–94. ACM, 1977. doi:10.1145/800022.808314.

12 Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 42–47. ACM, 2009. doi:10.1145/1565824.1565830.

13 Matt D'Souza, James You, Ondrej Lhoták, and Aleksandar Prokopec. TASTyTruffle: Just-in-time specialization of parametric polymorphism. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1561–1588, 2023. doi:10.1145/3622853.

14 Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981. doi:10.1145/322234.322235.

15 Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):220:1–220:24, 2020. doi:10.1145/3428288.

16 ECMA International. Standard ECMA-262 - ECMAScript language specification, June 2015. 6th edition. URL: http://www.ecma-international.org/ecma-262/6.0/.

17 Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998. doi:10.1145/290229.290234.

18 David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O'Boyle, editors, *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 126–137. ACM, 2018. doi:10.1145/3168811.

19 Manuel Serrano. Bigloo. http://www-sop.inria.fr/indes/fp/Bigloo/, 2024.

20 Marc Feeley. Gambit. https://gambitscheme.org, 2024.

21 Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1176–1197, 2023. doi:10.1145/3622839.

22 Olivier Melançon, Marc Feeley, and Manuel Serrano. An executable semantics for faster development of optimizing Python compilers. In João Saraiva, Thomas Degueule, and Elizabeth Scott, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23-24, 2023*, pages 15–28. ACM, 2023. doi:10.1145/3623476.3623529.

23 Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009. doi:10.1137/1.9780898717716.

24 Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, pages 56–66. ACM, 1995. doi:10.1145/207110.207116.

25 PLT Inc. Racket. https://racket-lang.org/, 2024.

26 R. Kent Dybvig. Chez Scheme. https://www.scheme.com/, 2024.

27 Baptiste Saleil and Marc Feeley. Interprocedural specialization of higher-order dynamic languages without static analysis. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 23:1–23:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ECOOP.2017.23.

28 Baptiste Saleil and Marc Feeley. Building JIT compilers for dynamic languages with low development effort. In Stephen Kell and Stefan Marr, editors, *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2018, Boston, MA, USA, November 4, 2018*, pages 36–46. ACM, 2018. doi:10.1145/3281287.3281294.

29 Manuel Serrano. JavaScript AOT compilation. In Tim Felgentreff, editor, *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*, pages 50–63. ACM, 2018. doi:10.1145/3276945.3276950.

30 Manuel Serrano. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473575.

31 Manuel Serrano and Marc Feeley. Property caches revisited. In José Nelson Amaral and Milind Kulkarni, editors, *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, pages 99–110. ACM, 2019. doi:10.1145/3302516.3307344.

32 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. doi:10.1145/1328438.1328486.

33 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 117–128. ACM, 2010. doi:10.1145/1863543.1863561.