

# PiNodes in the Druid Meta-Compiler

Matias Demare<sup>1</sup>, Guillermo Polito<sup>1</sup>, Nahuel Palumbo<sup>1</sup> and Javier Pimás<sup>2</sup>

<sup>1</sup>Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

<sup>2</sup>Univ. de Buenos Aires, Departamento de Computación, FCEyN, Buenos Aires, Argentina

## Abstract

Conditional branches incur high runtime overheads: they disrupt the processor's pipeline and add complexity to the control flow, preventing optimizations. Moreover, after inlining transformations and runtime-inserted type checks, it is often the case that some of these branches are redundant. Thus, there are opportunities to eliminate such branches without changing the semantics of the program.

In this paper, we explore methods of eliminating such branches in the Druid meta-compiler, a project intended for source-to-source ahead-of-time generation of baseline JIT compilers from a language interpreter. Moreover, we compare our new solution against a pre-existing algorithm in Druid. We extend its intermediate representation with PiNodes, a sparse representation of constraints on SSA variables. Such constraint information is leveraged by the compiler to remove redundant branches. We propose two PiNode-based methods of flow-sensitive analysis: one based on modelling possible constant values for variables, and a re-implementation of the traditional ABCD algorithm.

## Keywords

Meta-compiler, Constraint propagation, Dead-Branch elimination, Optimization, PiNodes, Pharo

## 1. Introduction

Programs contain conditional branches because of programmer-written control flow, the result of inlining functions, and, in high-level languages, because of runtime-inserted safety checks such as bounds and type checks. Conditional branches are detrimental to performance in many ways. For example, they need to execute a potentially expensive and pipeline-disrupting branch operation, and they introduce more control flow, which can prevent optimizations. Additionally, they may significantly increase code size, harming cache locality.

An optimizing compiler can gather flow-sensitive information, leverage it to determine possible runtime values, and optimize the code. The compiler must track and propagate the information learned at conditional branches in the form of *constraints*. Then, with the aid of constraint-solving tools or algorithms, it determines if a constraint is implied by another, or if they contradict each other. To illustrate this, let us consider the code of Listing 1 presenting two nested conditionals. In this example, the inner conditional statement is redundant, since it will always execute the `ifTrue` branch.

```
x < 3 ifTrue: [  
  x < 5 ifTrue: [  
    "reachable"  
  ] ifFalse: [  
    "unreachable"  
  ].  
].
```

Listing 1: Example of an unreachable branch that can be deleted without changing what the program does.

In this paper, we propose to study flow-sensitive constraint-based optimizations using PiNodes in the Druid meta-compiler, a compiler generator program. We enhance the compiler's intermediate language with constraint information, encoded as nodes in the instruction graph, at each branch and merge point in the control flow. Then, we leverage this information to enable more optimization opportunities.

*IWST 2025: International Workshop on Smalltalk Technologies, July 1-4, 2025, Gdansk, Poland*



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

This representation exploits the sparseness of the SSA form: obtaining all constraints imposed on a variable at a certain point in the program simply requires traversing its use-def chain, and collecting the constraints found.

In particular, we propose two different techniques for dead branch elimination, one based on modelling possible constant values ranges for variables, and one based on the traditional ABCD algorithm [1]. We compare them both in terms of optimization speed and optimization quality (measuring number of optimized branches and runtime performance). Additionally, we compare them with a preexisting technique used in the Druid meta-compiler.

Existing work proposes techniques that augment compiler intermediate representations to elide type checks [2, 3] and array bound checks [1]. Jump threading [4] modifies the control flow to make it linear when possible. Message splitting [3] duplicates basic blocks at merge points to specialize them for their predecessors.

Extensions to the IR similar to the one we used have been applied in other implementations. Julia’s PiNodes [2] use the same representation, whereas the ABCD paper [1] and the LLVM PredicateInfo pass [5] also implement extensions to the SSA form, though the constraint is not represented within an SSA instruction.

Section 2 presents some preliminary definitions, along with the context of our research. In Section 3, we explain how the general framework for PiNodes-based optimizations works, and how we perform the insertion and deletion of PiNodes. In Section 4, we explain how we use this information to perform dead branch elimination. In Section 5, we present a preliminary performance evaluation of our implementation, and we compare it with an existing implementation of dead branch elimination in the Druid compiler infrastructure [6]. Finally, in Section 7, we conclude and explain how we intend to use PiNodes for future research.

## 2. Context

### 2.1. Preliminary Definitions

This Section provides the basic definitions needed to understand the rest of the paper. If the reader is well versed in the SSA representation, they may choose to skip to the next subsection.

**Definition 1** (Control Flow Graph). A **control flow graph** (or **CFG**) is a graph-based representation of a program, in which each node represents a **basic block** (a consecutive group of instructions with no branching), and edges represent jumps between basic blocks.

**Definition 2** (Single Static Assignment form). A control flow graph is said to be in **Single Static Assignment** form [7] (or **SSA form**) if each variable is assigned exactly once. This form uses  $\phi$  functions to represent variables that are assigned a different value depending on the path took along the control flow graph.  $x_3 := \phi(B_1 \rightarrow x_1, B_2 \rightarrow x_2)$  defines  $x_3$  to be equal to  $x_1$  if the last executed block was  $B_1$ , or equal to  $x_2$  if it was  $B_2$ . The SSA form simplifies dataflow analyses, like computing a **use-def chain**.

**Definition 3** (Use-def chain). A **use-def chain**, is a data structure that links each variable usage with all the definitions of that variable that can reach the usage. SSA simplifies finding use-def chains, because for each variable, there is only one place where that variable may have received a value.

**Definition 4** (Domination). Given a control flow graph, and two of its basic blocks  $B_1$  and  $B_2$ , we say that  $B_1$  **dominates**  $B_2$  if every path from the entry node to  $B_2$  must go through  $B_1$ . Variables are available in every block dominated by the block the variable was defined in, and conversely, variable usages are constrained by all the conditions imposed upon them in any block that dominates the usage.

**Definition 5** (Critical Edge). Given a control flow graph, a **critical edge** is an edge whose successor has multiple predecessors, and whose predecessor has multiple successors. Figure 2 shows on the left an example of a critical edge.

## 2.2. The Druid Compiler Infrastructure

Druid<sup>1</sup> is a compiler infrastructure originally born for meta-compilation. Its main usage is for source-to-source ahead-of-time generation of baseline JIT compilers from a language interpreter [6]. Conceptually, Druid works as a compiler generator program (Cogen) [8].

In essence, Druid is an optimizing compiler which optionally targets the Cogit JIT compiler [9]. The DruidIR is a control flow graph in three-address-code SSA form [7]. The unit of meta-compilation is a *bytecode handler* i.e., given an interpreter handler, it generates the corresponding *generator handler*, as illustrated in Figure 1.

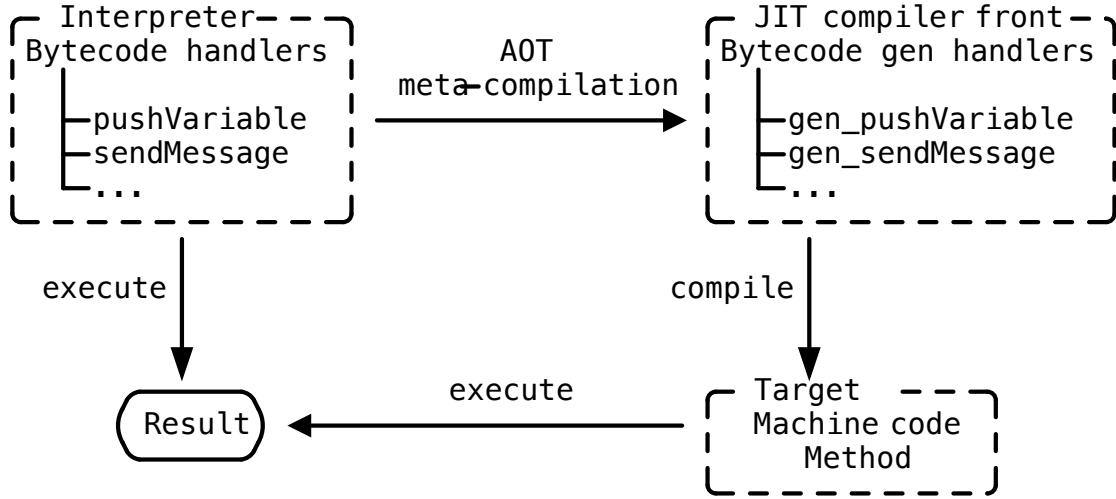


Figure 1: JIT compiler generation approach by meta-compilation of the interpreter.

## 2.3. Druid's Preexisting Constant Dead Branch Elimination Algorithm (old-CDBE)

It is worth mentioning that Druid already implemented an expensive version of constant dead branch elimination, which we will use as a comparison baseline in our experiments. That old implementation works in a similar way to our new constant dead branch elimination method. In fact, it shares the methods used for constraint solving, but represents constraints densely: it attaches constraints to CFG edges, and computes them for all paths a variable is live in. As such, it does not really profit from the sparseness of the SSA form, which makes this optimization expensive. In fact, it can timeout if the number of paths it needs to generate constraints for grows too large. In this case, optimization opportunities are missed. In the rest of this paper we refer to this old constant death branch elimination implementation as old-CDBE.

## 3. Tracking Constraint Information with PiNodes

This Section presents the PiNode framework. PiNodes are nodes in the compiler's intermediate language that attach constraint information to SSA variables. Constraints are of different kinds, encoding ranges of numeric variables, and type information, among others. PiNodes have the same semantics as Copy instructions, with the only difference being that they track the new constraint on the variable. This representation leverages the sparseness of the SSA form, thereby avoiding expensive dataflow analyses.

<sup>1</sup><https://github.com/Alamvic/druid> visited on 2025-03-01.

### 3.1. PiNodes by Example

Let us consider the example code in 2 that we assume is already in SSA form for practical purposes. This code shows a conditional statement with and without PiNodes to the left and right, respectively.

```
"Before"
x1 < y1 ifTrue: [
    x1 doSomething.
] ifFalse: [
    y1 doSomethingElse.
].

"After"
x1 < y1 ifTrue: [
    x2 := Pi(x1, <y1).
    y2 := Pi(y1, >x1).
    x2 doSomething.
] ifFalse: [
    x3 := Pi(x1, >=y1).
    y3 := Pi(y1, <=x1).
    y3 doSomethingElse.
].
```

**Listing 2:** Insertion of PiNodes in conditional branch.

In the example, PiNodes are inserted on each branch. Each PiNode includes the constraints learned from the branch itself, and defines a new version of the SSA variable (e.g., x2 or x3 are defined for x1). Users of the old variable need to be rewritten to use the newly inserted PiNode.

Note that if a PiNode adds a constraint that contradicts the existing restrictions imposed in a variable, it must mean that the branch that leads to it contradicts some parent branch. Therefore, that means that the PiNode's basic block must be unreachable.

### 3.2. The PiNode Framework

Within the PiNode framework, code transformations and optimizations are organized in three phases. In the first phase, control flow is evaluated, and PiNodes are inserted. In the second phase, optimizations are applied, taking benefit from the constraint information available at PiNodes. Finally, PiNodes are removed when lowering to a lower-level code representation (e.g., machine code).

**PiNode insertion step.** For each conditional branch, we insert two PiNode for each variable appearing in the condition: one in each successor of the condition's basic block. The PiNode in the `true` branch stores the condition of the branch over the variable, and the one in the `false` branch stores the negated condition.

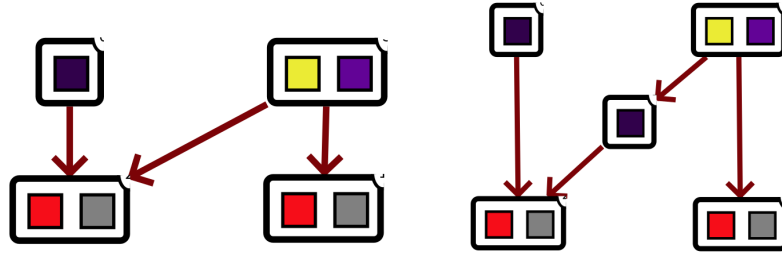
**Variable rename step.** Once PiNodes have been inserted, variable uses are renamed with the objective of making explicit in the use-def chain the dependencies on constrained nodes. Our algorithm follows the CFG domination tree and replaces any usages of the constrained variable that:

- Are dominated by the new variable.
- Do not correspond to PiNodes within the same block, since having dependencies between them would negate the benefit of being able to collect all relevant constraints by traversing the use-def chain.

**Critical Edges.** Special consideration must be taken regarding critical edges, since in that case, the added constraint may not hold true for every possible path. One option is to keep critical edges, but not insert any PiNodes in any block that succeeds a critical edge. Another option is to *break* critical edges: remove them, and insert a new basic block with an unconditional jump to the critical edge's target in their place. This is the path we took, since the information in that branch may be valuable for future work, and in particular for message splitting. An example of this procedure is shown in Figure 2.

**PiNode deletion.** PiNode removal happens by propagating the constrained variable to the users of the PiNode. The main idea is to revert the variable rename step. Internally, this algorithm is similar to a copy-propagation algorithm.

**Constraints.** PiNodes are represented as an instruction, containing as operands the constrained variable, and the constraint imposed on it. In principle, this constraints can represent anything that can be tested with a condition in Druid. However, not all types of constraints are usable by all of our optimization algorithms. Table 1 shows which kinds of constraints are supported by each of our algorithms.



**Figure 2:** A control flow graph with a critical edge (on the left), and the same control flow graph with that critical edge broken (on the right). The colored squares represent instructions, while the larger rectangles represent basic blocks.

	Constant DBE	ABCD DBE
Integer comparisons ( $<$ , $>$ , $\geq$ , $\leq$ )	x	x
Equality ( $=$ , $\neq$ )	x	
Any Mask / None Mask (bitand)	x	
Type constraint (IsType)	x	

**Table 1**  
Constraints supported by each algorithm.

## 4. PiNode-based Dead Branch Elimination

In this Section, we present two algorithms for dead branch elimination using PiNodes. The first one considers constraints against constant values, the second one extends this work by considering comparisons between variable values.

The general structure of the algorithms is the same, and we show its pseudocode in Listing 3: for each PiNode, it checks if its constraint is satisfiable. If it is not, it removes the jump to that block (*i.e.*, it converts the conditional branch instruction in the previous block, into an unconditional jump that points to the other sibling block). Note that, since we removed critical edges during PiNode insertion, blocks with PiNodes have only one predecessor, which ends in a conditional branch instruction, and no optimizations currently break that invariant. Our two new algorithms differ on how the satisfiability check is performed, which impacts the kinds of dead branches they detect, and also determines their performance characteristics.

```

cfg piNodesDo: [ :piNode |
  solver ifNotSatisfiable: piNode do: [
    unreachableBlocks add: piNode basicBlock.
  ].
].
cfg removeJmpsTo: unreachableBlocks.

```

Listing 3: Pseudocode for our new dead branch elimination algorithms.

## 4.1. New Constant Dead Branch Elimination (new-CDBE)

**The algorithm.** This algorithm considers only constraints with constant values. Constraints are represented in a class hierarchy, where each instance can answer whether it includes a specific value, and all values above and below a specific value. Using these methods, they can also answer if a constraint is included in another one. On top of the basic constraints shown in Table 1, there are also classes that represent the intersection and union of constraints.

The algorithm proceeds as follows:

- For a given PiNode, it collects all constraints on its argument variable (*i.e.*, without including the constraint added by the PiNode itself), by iterating over the SSA use-def chain. The constraints found as direct dependencies are interpreted as a conjunction, whereas dependencies from Phi functions are interpreted as a disjunction of the operand's constraints.
- It checks if the intersection between the collected constraint and the PiNode's constraint is empty or not. To do that, it negates the collected constraint, and checks if that negated constraint includes the PiNode's constraint.
- If it is empty, that means that the new constraint represented by the PiNode is not satisfiable, and therefore its block must be unreachable.

The pseudocode for the algorithm described can be found in Listing 4.

```
isSatisfiable: aDRPiNode  
| collectedConstraint |  
  
collectedConstraint := self collectConstraintsFrom: aDRPiNode operand.  
  
^ (collectedConstraint negated includes: aDRPiNode constraint) not.
```

Listing 4: Pseudocode for the new-CDBE algorithm.

```
x1 < 3  
ifTrue: [  
    x2 := Pi(x1, <3).  
    x2 < 5 ifTrue: [  
        x3 := Pi(x2, <5).  
    ] ifFalse: [  
        x4 := Pi(x2, >=5). " unreachable "  
    ].  
].
```

Listing 5: Unreachable block that is detected by new-CDBE.

**Example.** Let us consider the example in Listing 5, which is the result of performing PiNode insertion on the code from Listing 1:

- The algorithm starts analyzing `x2 := Pi(x1, <3)`. It collects all constraints on `x1` (it has none, so it is represented with a constraint that includes all values), and it checks if it is consistent with the new constraint, `<3`. Since it is, that means that the PiNode is reachable.
- It continues with the following PiNode, `x3 := Pi(x2, <5)`. It collects all constraints on `x2`, which would be `x2 < 3`, and checks if it is consistent with the new constraint, `<5`. Again, that is satisfiable.
- It continues with the next PiNode, `x4 := Pi(x2, >=5)`. It collects the constraints on its argument `x2`, which would be again `x2 < 3`, and checks if the new constraint `>=5` is satisfiable. Since the intersection between `<3` and `>=5` is empty, it marks that block as unreachable.

**Implementation details and limitations.** We extended this algorithm to handle copy propagation and constant propagation. This means that we consider copy instructions of the form  $x := y$  to attach the constraints from  $x$  to  $y$ . The main limitation of this implementation, is to not able to propagate constraints across basic arithmetic operations (like addition and subtraction), nor is it able to handle constraints between variables (for example, constraints of the form  $x < y$ ).

## 4.2. Array Bounds Checks elimination on Demand (ABCD)

To deal with the previous method's limitations, we implemented another method of constraint solving based on ABCD [1]. Their implementation is intended to be used for array bounds checks elimination, but we extended it to detect dead branches of any kind, by leveraging the fact that array bounds checks are just a particular case of conditional branches.

**Overview.** This algorithm uses a weighted directed graph, called the *inequality graph*, to represent the constraints over the variables, allowing it to represent constraints between variables, and to optimize cases like the one shown in Listing 6. Figure 3 shows the corresponding graph. Nodes in the inequality graph represent SSA values. An edge from  $x_1$  to  $y_1$  with weight  $w$  means that  $y_1 - x_1 \leq w$ . Using this model, the algorithm considers that  $x < y$  is a tautology if the shortest path from  $y$  to  $x$  has negative weight. Here, the definition of shortest path is a generalization of the traditional definition with extensions for phi nodes.  $\phi$  functions work as *max nodes*, meaning they consider the longest path for each of its neighbours instead of the shortest one. Conceptually, this means that  $\phi$  functions are bounded by the weakest constraint of its operands.

**Example.** In the example in Figure 3, which shows the inequality graph corresponding to the code in Listing 6, the shortest path from  $y_2$  to  $x_3$  has negative weight. Thus,  $x_3 < y_2$  is a tautology and the `ifFalse` branch is unreachable. Note that the example cannot be optimized by constant dead branch elimination, since it requires modelling the relation between variables.

```
x1 <= y1
  ifTrue: [
    x2 := Pi(x1, <=y1).
    y2 := Pi(y1, >=x1).
    x3 := x2 - 10.
    x3 < y2 ifTrue: [
      x4 := Pi(x3, <y2).
      y3 := Pi(y2, >x3).
    ] ifFalse: [
      x5 := Pi(x3, >=y2). " unreachable "
      y4 := Pi(y2, <=x3).
    ].
  ].
```

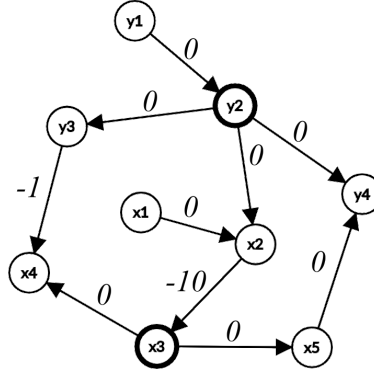
Listing 6: Unreachable block that is detected by the ABCD algorithm.

## 5. Evaluation

We wanted to perform an initial profitability analysis for the new algorithms, contextualized to the main use case of Druid, which is compilation of VM primitives. Our objective was to answer the following questions:

- Do the new methods detect dead branches?
- Do they detect more or less than the existing one?
- Is the ABCD algorithm more powerful than the others?
- Is there a reason to use more than one of these algorithms in combination?





**Figure 3:** ABCD inequality graph for code in Listing 6.

With that purpose, we compared the three algorithms (the one pre-existing in Druid, the new implementation of constant dead branch elimination, and the implementation of ABCD) in three different ways: meta-compilation speed, number of dead branches eliminated, and runtime speed.

**Experimental setup** All benchmarks were ran on a Macbook Pro with an Apple M3 Pro chip, and 18 GB of memory, running macOS 15.6. The laptop was plugged in, and no unnecessary programs were running in the background.

### 5.1. Meta-compilation Speed

To measure meta-compilation speed, we designed a benchmark that consists of analyzing and optimizing a selection of 50 VM primitives supported by Druid. We computed the time to run using the high-resolution clock of the machine, exposed by `Smalltalk highResClock`. We recorded both the average and standard deviation, for three runs of compiling all the measured primitives.

Do note that in the case of the new algorithms, PiNode insertion time is included, and in the case of the old one, the time to generate and attach the constraints to the edges is also included. Moreover, when we run both of the new algorithms, it is only necessary to perform PiNode insertion once. Results are shown in Table 2.

Configuration	Average cycles	Std dev
old-CDBE	$17.68 \times 10^7$	$5.158 \times 10^6$
new-CDBE	$1.510 \times 10^7$	$1.729 \times 10^6$
ABCD	$2.139 \times 10^7$	$1.915 \times 10^6$
new-CDBE & ABCD	$2.069 \times 10^7$	$1.118 \times 10^5$

**Table 2**

Total optimization time for primitive methods under different configurations.

**Conclusion.** As we can see, our early results show that both the new-CDBE and ABCD methods are substantially faster than the old-CDBE implementation, resulting in faster compile times. The new-CDBE algorithm is roughly an order of magnitude faster, while the ABCD implementation is about 4x faster.

We can see that running both the new-CDBE and ABCD methods in conjunction is faster than just running the ABCD method. This is because the constant dead branch elimination method is able to significantly simplify the CFG, reducing the size of the graph that the more expensive method, ABCD,



has to deal with. Additionally, PiNode insertion only needs to be performed once before running both methods.

Finally, it is also worth noting that in 3 out of the 50 primitives methods used for benchmarking, the old dead branch elimination algorithm timed out due to their complexity. This has an impact on the optimization opportunities it is able to find, as we will see in the next subsection.

## 5.2. Number of Dead Branches Eliminated

Using the same selection of primitives, we compared the total number of dead branches detected by each algorithm. We also compared what happened if we ran both of our new algorithms, one after the other, to see how they complemented each other, and determine if they were finding different sets of dead branches. Results are shown in Table 3.

Configuration	Killed Branches	Basic blocks removed
old-CDBE	434	224
new-CDBE	507	236
ABCD	3	4
new-CDBE & ABCD	508	239

**Table 3**

Number of branches and blocks killed under different configurations.

**Conclusion.** These findings lead us to conclude:

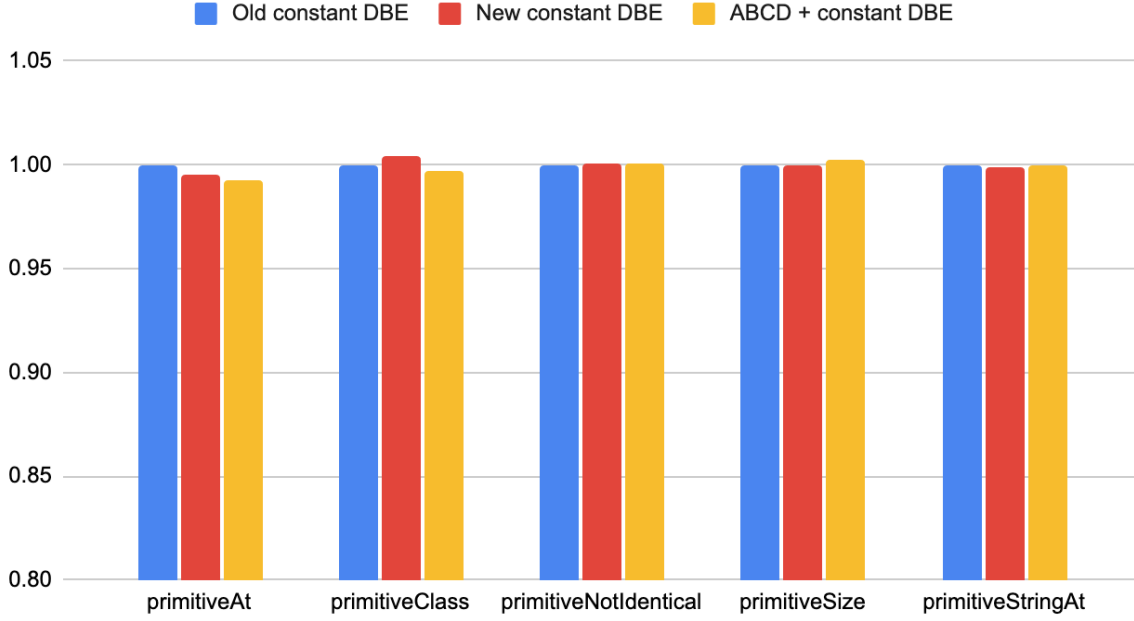
- Even though the old and new CDBE should be mostly equivalent, the new method removed more branches. This is because the old method bails out in some cases while performing its analysis, due to an excessively complex control flow. In those cases, it does not remove any dead branches, even though those are in general the methods with the most dead branches.
- ABCD does not eliminate many branches. This is probably due to the fact that the VM methods do not perform bounds checks, which are the cases this method is designed to detect. While the algorithm does use equality constraints inferred from conditionals to build the inequality graph, it cannot infer whether a check of that type is redundant.
- If we run both constant elimination and ABCD, we get better results than with just constant elimination, which shows that ABCD detects some dead branches that the other method does not.
- If we ignore the cases in which the old algorithm bails out, in general the new and the old CDBE methods remove the same number of branches, with a couple of exceptions in which the new one performs marginally worse. We believe this is due to small implementation details we still have to iron out.

## 5.3. Runtime Speed

To assess runtime speed differences between the different methods, we replaced four VM primitives with versions optimized by each algorithm, and we ran microbenchmarks designed to stress test each of them. We present the results in Figure 4.

We should note that we were unable to run benchmarks optimizing only with the ABCD algorithm. Since it did not eliminate enough branches, the size of IR graphs usually increased, which in turn caused an increase in stack spills. Those spills in some cases made the stack grow too big, crashing the VM and preventing us from running the desired benchmarks. Therefore, we only ran ABCD in conjunction with the new-CDBE, to see if using it provided any advantage.

## Runtime relative to "Old constant DBE" (lower is better)



**Figure 4:** Speed relative to the old method, in multiple microbenchmarks.

**Conclusion.** As we can see, the runtime performance between all three methods is comparable, with the difference between them being under 1%. Therefore, we can conclude that:

- The new and old CDBE methods have similar capabilities for detecting dead branches.
- Using them in conjunction with the ABCD method does not make a significant difference in execution time.

## 6. Prototype Implementation Details and Limitations

In this Section we briefly explain some implementation details of our algorithms. We implemented these algorithms in the Druid compiler infrastructure [6].

Currently, none of the optimizations implemented on Druid invalidates the PiNodes. This means that it is possible to apply multiple optimizations after PiNode insertion.

During the insertion phase, we did not take any special considerations for the points where control flow merges. Merge points are not dominated by either branch, thus, variable usages will not be replaced by the new variables. Moreover, this also means that we lose constraint information at merge points. This loss of constraint information does not prevent the two optimizations presented in this article. However, it prevents opportunities for message splitting. We intend to insert Phi functions at merge points, making explicit the merge of constraint information.

At the moment of writing this article, our ABCD prototype only supports upper bounds checks elimination. Lower bounds checks elimination works similarly, but the building of the inequality graph follows different rules.

## 7. Conclusion and Future Work

In this paper, explored augmenting the SSA IR from the Druid meta-compiler with constraint information, to aid context-sensitive optimizations. We implemented two dead branch elimination techniques using

this representation, and we compared them with the existing technique in the Druid meta-compiler.

Our results show that the new method of constant dead branch elimination is considerably faster than the existing method in terms of meta-compilation speed, while still detecting approximately the same dead branches. This makes VM build times faster, without any significant difference to runtime performance.

On the other hand, the ABCD algorithm does not seem fit for optimizing VM primitives. While it is still faster than the old dead branch elimination method, its is designed to eliminate a kind of redundant conditions that does not appear to be very common in the VM's code, and therefore it cannot eliminate enough branches for it to be useful.

We are currently working on other methods of constraint solving for dead code elimination, that would allow the compiler to solve more complex constraint systems. Our current goals include using the Z3 constraint solver [10]. Additionally, we intend to explore the usage of PiNodes to inform opportunities for message splitting.

## Acknowledgments

This project is financed by the ANR JCJC project convention ANR-25-CE25-0002-01.

## Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

## References

- [1] R. Bodik, R. Gupta, V. Sarkar, Abcd: eliminating array bounds checks on demand, in: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 2000, pp. 321–333.
- [2] Julia ssa-form ir, <https://docs.julialang.org/en/v1/devdocs/ssair/#Phi-nodes-and-Pi-nodes>, 2025. Accessed: 2025-04-07.
- [3] C. Chambers, D. Ungar, Iterative type analysis and extended message splitting; optimizing dynamically-typed object-oriented programs, in: Proceedings of the ACM SIGPLAN 1990 conference on Programming Language Design and Implementation, 1990, pp. 150–164.
- [4] Llvm jump threading, <https://llvm.org/docs/Passes.html#jump-threading-jump-threading>, 2025. Accessed: 2025-04-07.
- [5] Llvm predicate info, [https://llvm.org/doxygen/PredicateInfo\\_8h.html](https://llvm.org/doxygen/PredicateInfo_8h.html), 2025. Accessed: 2025-04-07.
- [6] N. Palumbo, G. Polito, S. Ducasse, P. Tesone, Meta-compilation of baseline jit compilers with druid, The Art, Science, and Engineering of Programming 10 (2025). URL: <http://dx.doi.org/10.22152/programming-journal.org/2025/10/9>. doi:10.22152/programming-journal.org/2025/10/9.
- [7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (1991) 451–490.
- [8] L. Birkedal, M. Welinder, Hand-writing program generator generators, in: Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming, PLILP '94, Springer-Verlag, Berlin, Heidelberg, 1994, p. 198–214.
- [9] E. Miranda, The cog smalltalk virtual machine, in: Proceedings of VMIL 2011, 2011.
- [10] L. De Moura, N. Bjørner, Z3: an efficient smt solver, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, p. 337–340.