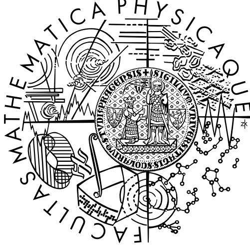

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Gocník

Truffle based .NET IL interpreter and compiler: run C# on Java Virtual Machine

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Štěpán Šindelář
Study programme:	Computer Science (B1801)
Study branch:	IPSS (1801R048)

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedication. I want to thank everyone who had the patience with me throughout my very long bachelor studies and didn't stop believing I'll finish it one day.

I wouldn't know of this topic without Adam Hornáček, who connected me with his colleagues. Special thanks go to my supervisor, Štěpán Šindelář, who always provided guidance when I needed it and didn't give up on me when I missed a deadline (or two). This work also wouldn't exist without everyone at my employer who allowed me to take my time.

Finally, I wouldn't make it without tremendous support from my family and from Antonín and Lucie. Thank you.

Title: Truffle based .NET IL interpreter and compiler: run C# on Java Virtual Machine

Author: Jan Gocník

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Štěpán Šindelář, Department of Distributed and Dependable Systems

Abstract: Traditionally, to achieve high performance for executing dynamic languages, a hand-crafted Just-In-Time (JIT) compiler was necessary. Such compilers come with several disadvantages, including security issues arising from bugs in manual speculative optimizations. Our work focuses on the issue that these state-of-the-art projects can be prohibitively complicated for students, academics and hobbyists interested in programming language design and implementation. A novel project, the Truffle Language Implementation Framework, uses partial evaluation to convert interpreter-style code into an optimizing compiler. Authors propose that the reduced complexity for implementing languages Truffle offers will allow more languages to benefit from high performance compilation. To validate this claim, we implement BACIL, a Truffle-based runtime for .NET (CLI). While built in an academic setting with inherently limited resources, its peak performance achieves under 10 times slowdown compared to .NET's official runtime. We release the implementation as open-source with the hope it can further promote experimentation with programming languages.

Keywords: partial evaluation Graal Truffle CIL JVM

Contents

Introduction **3**

1 Context **5**

1.1 .NET/CLI 5
1.2 Truffle and Graal 6
1.3 Previous work 8
1.4 Bytecode interpreter vs rebuilding an AST 8

2 Theory **10**

2.1 Partial Evaluation 10
2.2 Tiered compilation 11
2.3 Guards and de-optimizations 12
2.4 Escape analysis and virtualization 12
2.5 The `MERGE_EXPLODE` strategy 14

3 CLI Component parser **19**

3.1 Analysis 19
3.1.1 Design goals 19
3.1.2 Definition of important CLI component structures 20
3.1.3 Complexities of the CLI component format 20
3.2 Parser implementation details 24
3.2.1 Metadata tables parser 24
3.2.2 CLITableRow and CLITablePtr 24
3.2.3 Sequence references 25
3.3 Conclusion 26

4 Runtime **27**

4.1 Analysis 27
4.1.1 Nodes 27
4.1.2 Dynamicity of references 30
4.1.3 Standard libraries 34

4.1.4 BACILHelpers 34
4.1.5 Values and locations 34
4.1.6 CompilationFinal annotation 38
4.2 Debugging performance issues 38
4.2.1 Case study 39

5 Results **43**
5.1 Completeness 43
5.1.1 Library methods 44
5.2 Performance benchmarks 45
5.2.1 Harness 45
5.2.2 Hagmüller's work 46
5.3 .NET runtime JIT benchmarks 50
5.3.1 Warmup concerns 52
5.3.2 Interpreting the results 52

Conclusion **54**
5.3.3 Future work 55

Bibliography **56**

A Compiling and running BACIL **57**
A.1 Building 57
A.2 Running 57
A.3 Full example of running pre-compiled BACIL on amd64 Linux 59

B Opcode implementation status **60**

Introduction

Problem

Traditionally, when implementing a programming language, achieving high performance required a significant development effort and resulted in complicated codebases.

While writing an interpreter for even a fairly complicated language is achievable for a single person interested in the topic (as proved by the abundance of language implementation theses available), creating state-of-the-art optimizing compilers usually took several years spent by large teams of developers at the largest IT companies. Not only was kick-starting such a project unthinkable for an individual, but even introducing changes to an existing project is far from simple.

For example, as of 2022, Google's state-of-the-art JavaScript engine V8 has two different JIT compilers and its own internal bytecode. An experiment of adding a single new bytecode instruction to the project can mean several days of just orientating in the codebase. Google provides a step-by-step tutorial for adding a new WebAssembly opcode to v8[1], which admits that a lot of platform-dependant work is necessary to get a proper implementation: "The steps required for other architectures are similar: add TurboFan machine operators, use the platform-dependent files for instruction selection, scheduling, code generation, assembler."

As cybersecurity becomes a more important topic, another factor to consider is that creating manual optimizations in JITs is prone to bugs which can have grave security implications. Speculated assumptions of JIT compilers introduce whole new bug families. As saelo [1] says (emphasis added, footnotes stripped):

> JavaScript JIT compilers are commonly implemented in C++ and as such are subject to the usual list of memory- and type-safety violations. These are not specific to JIT compilers and will thus not be discussed further. Instead, the focus will be put on bugs in the

[1] `https://v8.dev/docs/webassembly-opcode`

> compiler which lead to incorrect machine code generation which can then be exploited to cause memory corruption.
>
> Besides bugs in the lowering phases which often result in rather classic vulnerabilities like integer overflows in the generated machine code, many interesting bugs come from the various optimizations. There have been bugs in bounds-check elimination, escape analysis, register allocation, and others. *Each optimization pass tends to yield its own kind of vulnerabilities.*

Implementing a JIT that is not only performant but also secure is proving to be difficult even for state-of-the-art projects.

These factors resulted in academic and hobby experimentation with programming languages being mostly stuck with low-performance simple interpreters. Kwame, Martey, and Chris [2] conclude that

> Interpreters are very good development tools since it [sic] can be easily edited, and are therefore ideal for beginners in programming and software development. However they are not good for professional developers due to the slow execution nature of the interpreted code.

In recent years, frameworks appeared that promise to deliver performance comparable to state-of-the-art JIT compilers while requiring only a simple interpreter-style implementation. Examples of such frameworks are RPython[2] and the Truffle language implementation framework[3]. Würthinger et al. [3] concluded that Truffle's performance "is competitive with production systems even when they have been heavily optimized for the one language they support".

As the performance aspects of language implementations made by experts (sometimes even designers of these frameworks themselves) are well understood, in this work we want to focus on testing another claim of Würthinger et al. [3]: the "reduced complexity for implementing languages in our system [that] will enable more languages to benefit from optimizing compilers".

Is it feasible to achieve the promised performance benefits with an academic interpreter-style implementation of a language runtime? In order to answer this question, we implement BACIL, a runtime for .NET.

[2] `https://rpython.readthedocs.io/`

[3] `https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/`

Chapter 1

Context

1.1 .NET/CLI

We chose .NET as a platform to implement, mostly because:

- Languages targeting .NET consistently rank high on popularity surveys.
- We have experience with .NET internals and the internally used bytecode.
- No comparable truffle-based implementations were already published for .NET.

While .NET is a well-recognized name, it is a marketing/brand name whose meaning changed through history. Our implementation follows the ECMA-335 Common Language Infrastructure (CLI) standard[4] which does not mention the .NET brand at all. We will use the names defined in the standard throughout this work. We include all references to specific implementations/brand names only to aid understanding with no ambition to be accurate, mainly for .NET vs .NET Core vs .NET Framework vs .NET Standard nomenclature.

"The Common Language Infrastructure (CLI) provides a specification for executable code and the execution environment (the Virtual Execution System) in which it runs."[4] .NET languages (like C#) are compiled into "managed code"[1] — instead of targeting native processor instruction sets, they target the CLI's execution environment.

Using the definitions of the standard, BACIL is actually a Virtual Execution System (VES):

[1] They can also be ahead-of-time compiled to native binaries, but that is out of scope for this work.

> The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding).[4]

.NET Framework's VES is called the Common Language Runtime (CLR) and in .NET Core, it is known as CoreCLR. "To a large extent, the purpose of the VES is to provide the support required to execute the [Common Intermediate Language (CIL)] instruction set"[4].

The CIL, historically also called Microsoft Intermediate Language (MSIL) or simply Intermediate Language (IL), is the instruction set used by the CLI. Interpreting (a subset of) this instruction set was the primary goal of this work.

Another large part of the framework is the standard libraries — the base class library, which has to be supported by all implementations of the CLI, comprises 2370 members over 207 classes. As the focus of the work was on the core interpreter, we largely ignore this part of the standard and delegate to other standard library implementations where possible.

1.2 Truffle and Graal

To implement a high-performance CLI runtime, we employ the Truffle language implementation framework[2] (henceforth "Truffle") and the GraalVM Compiler[3]. These two components are tightly coupled together and we will mostly be referring to them interchangeably, as even official sources provide conflicting information on the nomenclature.

The Graal Compiler is a general high-performance just-in-time compiler for Java bytecode that is itself written in Java. It is state-of-the-art in optimization algorithms — according to official documentation[4], "the compiler in GraalVM Enterprise includes 62 optimization phases, of which 27 are patented".

Truffle is a framework for implementing languages that will be compiled by Graal. From the outside, it behaves like a compiler: its job is to take guest language code and convert it to the VM's language, preserving as much intrinsic metadata as possible. Unlike a hand-crafted compiler, Truffle takes an interpreter of the guest language as its input and uses "Partial evaluation" (see Section 2.1) to do the compilation, performing a so-called "first Futamura projection".

[2] `https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/`

[3] `https://www.graalvm.org/22.1/docs/introduction`

[4] `https://www.graalvm.org/22.1/reference-manual/java/compiler/#compiler-advantages`

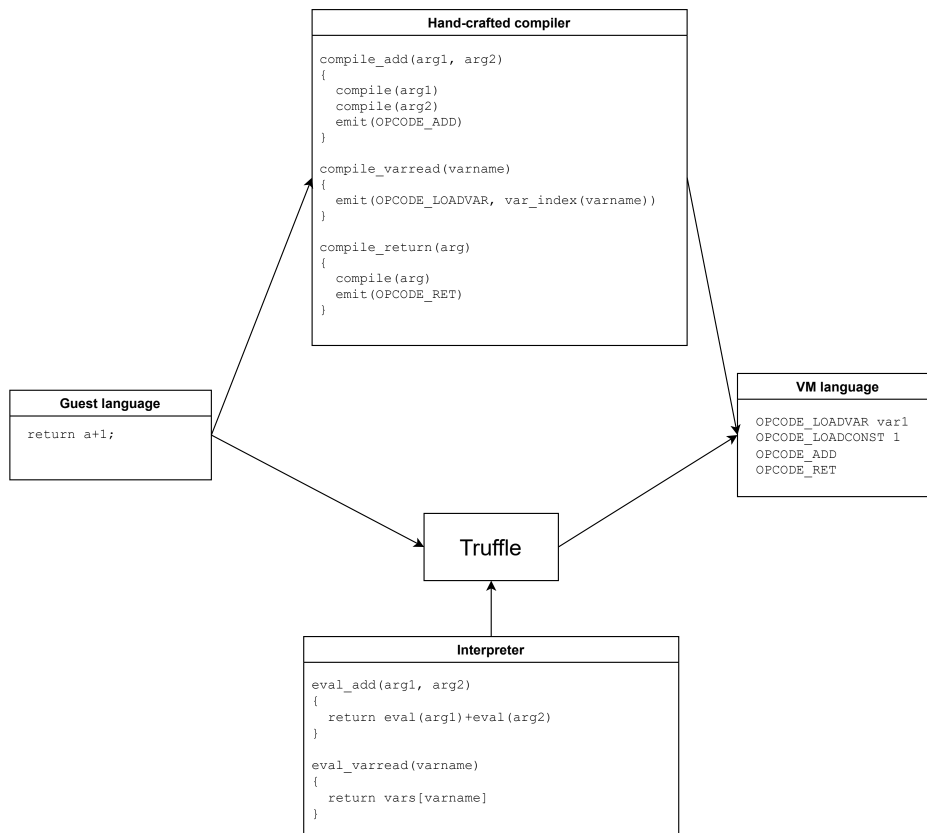

Figure 1.1 A flowchart comparing a traditional hand-crafted compiler (upper path) and Truffle compilation based on an interpreter (lower path)

Truffle also provides several primitives that the language implementation can use to guide the partial evaluation process, allowing for better results.

We want to mention that GraalVM is distributed in two editions, Community and Enterprise. Supposedly, the Enterprise edition provides even higher performance than the Community one. As we want to avoid all potential licensing issues, we only used the Community edition and cannot comment on Enterprise performance at all.

1.3 Previous work

Truffle was originally described as “a novel approach to implementing AST interpreters” by Würthinger et al. [5] and was not directly applicable to our bytecode interpreter problem.

Rigger et al. [6] implemented Sulong, an LLVM IR (bytecode) runtime, and showed “how a hybrid bytecode/AST interpreter can be implemented in Truffle”. This is already very similar to our current work, however, it implemented an unique approach of converting unstructured control flow into AST nodes.

In Truffle version 0.15 (2016)[5], the `ExplodeLoop.LoopExplosionKind` enumeration was implemented, providing the `MERGE_EXPLODE` strategy discussed in Section 2.5.

In GraalVM version 21.0 (2021)[6], an “experimental Java Virtual Machine implementation based on a Truffle interpreter” was introduced. This project is very similar to our work, using the same approaches for implementing a different language.

While Hagmüller [7] also implemented the CIL runtime, they chose a completely different approach, building an AST from the text representation of IL code. Also, as they admit in the conclusion, they “didn’t focus on performance optimization of the different instructions”. The same implementation approach was chosen by truffleclr[7].

1.4 Bytecode interpreter vs rebuilding an AST

There are two main ways to approach the implementation of a bytecode runtime on Truffle. While Hagmüller [7] chose to create a full-grown AST from the

[5] `https://github.com/oracle/graal/blob/master/truffle/CHANGELOG.md#version-015`

[6] `https://www.graalvm.org/release-notes/21_0/`

[7] `https://github.com/alex4o/truffleclr`

bytecode, Java on Truffle[8] was implemented as a more traditional fetch-decode-execute loop.

To skip the problem of parsing the bytecode, Hagmüller [7] uses a textual disassembly of the bytecode, and performs a more traditional lexical and syntactical analysis on it. For obvious reasons, parsing text like that is inherently slower than consuming tightly-packed bytecode. We want to avoid this preformance penalty, therefore we want to consume the bytecode itself.

While building a similar AST from the bytecode itself can be a valid approach, we feel that it is obfuscating the source structure for Truffle — the original source *is* a flat bytecode and not an AST. While Truffle's support for bytecode was historically lacking compared to AST patterns, we feel this is a problem that should be alleviated within Truffle and not by synthesizing arbitrary structures atop the original ones. Java on Truffle being released as a traditional bytecode interpreter shows that this pattern will be supported and therefore we see no reason to add complexity and obfuscation by synthesizing ASTs from the bytecode.

[8] `https://www.graalvm.org/22.0/reference-manual/java-on-truffle/`

Chapter 2

Theory

2.1 Partial Evaluation

The most important technique allowing Truffle/Graal to reach high performance is Partial Evaluation. It is theoretically known for decades, the foundations being laid by Futamura [8], but only advances in computer performance make it practically usable.

The high-level view of partial evaluation offered by Futamura is "specializing a general program based upon its operating environment into a more efficient program".

Consider a program (or its chunk) as a mapping of inputs into outputs. We can divide those inputs into two sets — dynamic inputs and static inputs — denoting the program as

$$prog : I_{static} \times I_{dynamic} \to O$$

The process of partial evaluation is then transforming $prog, I_{static}$ by incorporating the static input into the code itself, resulting in

$$prog* : I_{dynamic} \to O$$

We will call $prog*$ a specialization of $prog$ for I_{static}, sometimes it is also referred to as a residual program, intermediate program, or a projection of $prog$ at I_{static}.

For a simple example, let us consider $f(s, d) = s(s(s+1)+d)$. The specialization of f for $s = 2$ is then $f_2(d) = 2(6+d)$, effectively pre-computing one multiplication. An even more interesting specialization is $f_0(d) = 0$, turning the entire program into a constant expression.

The separation between I_{static} and $I_{dynamic}$ is not rigorous — it is valid both to create a separate specialization for every single input combination or to consider all input dynamic and therefore specialize for an empty set. However, these

extremes do not provide any performance benefits. Partial evaluation is therefore usually guided by heuristics that analyze when a specific input value is used *often enough* to warrant a specialization.

Futamura [8] formulates so-called Futamura projections. Let us define a generic specializer as

$$specializer : prog \times I_{static} \rightarrow prog\star$$

The first Futamura projection is as follows: Let us define an interpreter as a program taking two inputs, the source code and the "inner" inputs for the code.

$$interpreter : source \times inputs \rightarrow outputs$$

Then the result of $specializer(interpreter, source) = executable$ is a fully realized program for the specific source code as if the source code was "compiled" in the traditional sense of the word.

The second Futamura projection observes that

$$specializer(specializer, interpreter) = compiler$$

The resulting tool is a tailored specializer that can transform source code into executables.

The third Futamura projection observes that

$$specializer(specializer, specializer) = compiler\text{-}compiler$$

The resulting program takes an *interpreter* and returns a *compiler*.

In this work, we implement an interpreter and use Truffle to perform the first Futamura projection.

2.2 Tiered compilation

Because more aggressive compilation optimizations result in the compilation taking more time, it is a common practice to use tiered compilation. As a specific code gets called more often, it becomes worth it to recompile it again and better optimize it.

In Truffle/Graal, there is always a fallback of interpreting the code with no partial evaluation and compilation. This fallback is used both before the first compilation happens and when a de-optimization happens (see below).

One reason behind always starting in interpreter before compiling is that the interpreted invocations can already provide observations about the code,

for example branch probability, if such observations are implemented. These observations can be used so that the first compilations are already of high quality.

For our project, the difference between compiled tiers is not too interesting, as they usually have a relatively small performance difference between them. The biggest gap occurs between the interpreted code and the first compiled tier, where the execution time can differ by more than an order of magnitude.

2.3 Guards and de-optimizations

For practical partial evaluation, it is valuable to perform speculative optimizations — compiling the code expecting invariants that can be broken during runtime. One common example of such speculation is optimizations of virtual calls: assuming that the method will always be called on objects of a specific type allows replacing the virtual call with a static one and enables a more aggressive specialization.

Also, it is often useful to exclude some exceptional code paths from the compilation — for example, if dividing by zero should cause an immediate crash of the application with a message being printed out, there is no use in spending time compiling and optimizing the error-message printing code, as it will be called no more than once.

To achieve that, Graal uses guards — statements that, when reached by the runtime, result in de-optimization. De-optimization is a process of transferring evaluation from the compiled variant of the method back to the interpreter at the precise point where it was interrupted and throwing away the already compiled variant, as its assumptions no longer hold.

For an example, see a pseudo-code of what a single-cache virtual call implementation could look like in Listing 1. When partially evaluated with *invariant == true*, the resulting flow will look like in Figure 2.1. As long as the virtual call is effectively static at runtime, we only spend time compiling the actual target function (which can be specialized for the environment) and during invocation only pay the price of a simple equality check. Once the comparison fails, this version of the compiled method is thrown away, and a generic one is created, as shown in Figure 2.2.

2.4 Escape analysis and virtualization

All Java objects traditionally have to be allocated on the heap, as the VM has no concept of stack-allocated structures. However, allocating data on the heap is slow. The solution to this issue is escape analysis: if an object never leaves the current compilation unit, it can be virtualized. A virtualized object is never

Listing 1 Single-cache virtual call (pseudocode).

```
bool cached = false;
bool invariant = true;
type expectedType = null;
funcptr cache = null;

exec(obj, method)
{
  if(!cached)
  {
    cached = true;
    expectedType = obj.Type;
    cache = obj.Type.ResolveVirtualFunc(method)
  }

  if (invariant)
  {
    if(obj.type==expectedType)
    {
      return cache.call()
    } else {
      Deoptimize()
      invariant = false
    }
  }
  obj.Type.ResolveVirtualFunc(method).call()
}
```

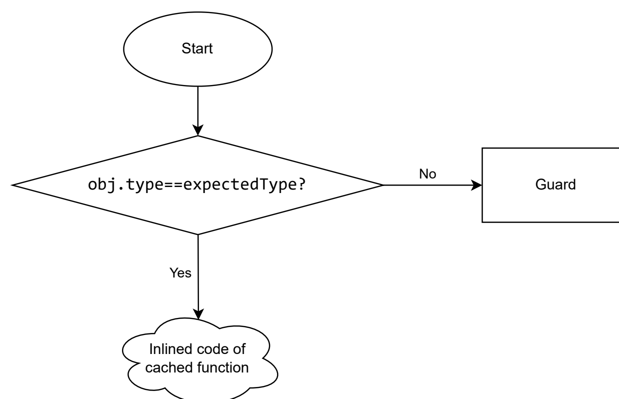

Figure 2.1 A flowchart of a single-cache virtual call (Listing 1) when *invariant* == *true*, with the call target fully inlined.

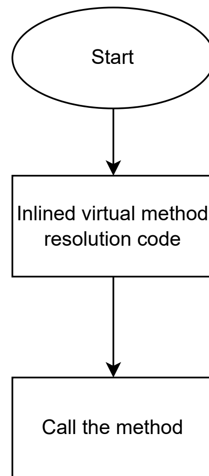

Figure 2.2 A flowchart of a single-cache virtual call (Listing 1) after it was called with multiple different objects so the optimized version in Figure 2.1 was invalidated.

actually allocated but is decomposed to its individual fields, which are then subject to partial evaluation and other optimization methods.

2.5 The `MERGE_EXPLODE` strategy

One of the key elements that allows for implementing partial evaluation friendly bytecode interpreters is the `MERGE_EXPLODE` loop explosion strategy. To quote the documentation[1] (emphasis added):

> like `ExplodeLoop.LoopExplosionKind.FULL_EXPLODE`, but copies of the loop body that have the exact same state (all local variables have the same value) are merged. This reduces the number of copies necessary, but can introduce loops again. *This kind is useful for bytecode interpreter loops.*

To fully appreciate the importance of this strategy, we have to point out the following fact of CLI's design from *I.12.3.2.1 The evaluation stack*[4] (emphasis added):

> The type state of the stack (*the stack depth* and types of each element on the stack) at any given point in a program *shall be identical for all possible control flow paths.* For example, a program that loops an

[1] `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/nodes/ExplodeLoop.LoopExplosionKind.html#MERGE_EXPLODE`

unknown number of times and pushes a new element on the stack at each iteration would be prohibited.

This design choice is not a coincidence, as it is vital also for hand-crafting performant JIT compilers. Regarding `MERGE_EXPLODE`, it means that all copies of the interpreter's inner loop that have the same bytecode offset will also have the same evaluation stack depth and type layout.

Thanks to this if we have, for example, a push immediate 4 instruction somewhere in the code, it can be translated to a simple statement like `stack[7] = 4`, as in every execution of this instruction the stack depth has to be the same. This enables more optimizations, as this constant can be propagated to the next instruction reading `stack[7]`.

To explain the inner working on a more involved example, we manually apply this strategy to

```
for(int i = 0; i < 100; i++) {a = a*a; }; return a;
```

We begin with a psuedo bytecode of this function in Listing 2 and a theoretical interpreter in Listing 3. Thanks to the strategy, only one state per bytecode offset has to be created. Knowing exactly the stack depth, we can partially evaluate the stack positions to constants. This is reflected in Listing 4. As the stack does not leave this method, it will be completely virtualized. Since the stack array is always accessed using constant indices, we can apply aggressive optimization and optimize out the array, reaching the final state in Listing 5.

Even though we started with a big interpreter loop, by merging the instances having the same bytecode offset, the interpreter loop disappears and the original control flow of the method reappears from the flat bytecode.

Listing 2 Pseudo bytecode of a function used to demonstrate the MERGE_EXPLODE strategy.

```
; i=0
0: OPCODE_LOADCONST 0
1: OPCODE_STOREVAR i

; i < 100
2: OPCODE_LOADVAR i
3: OPCODE_LOADCONST 100
4: OPCODE_JMPIFBEQ @14

; a = a*a
5: OPCODE_LOADVAR a
6: OPCODE_LOADVAR a
7: OPCODE_MULTIPLY
8: OPCODE_STOREVAR a

; i++
9: OPCODE_LOADVAR i
10: OPCODE_LOADCONST 1
11: OPCODE_ADD
12: OPCODE_STOREVAR i

; loop
13: OPCODE_JMP @2

; return a
14: OPCODE_LOADVAR a
15: OPCODE_RET
```

Listing 3 A theoretical interpreter we will apply MERGE_EXPLODE to (pseudocode).

```
pc = 0 //bytecode offset
top = 0 //stack top

while(True)
  opcode = getOpcode(pc)
  switch opcode:
    case OPCODE_LOADCONST:
      stack[top++] = getImmediate(pc+1)
      break
    case OPCODE_STOREVAR:
      vars[getVar(pc+1)] = stack[top--]
      break
    case OPCODE_LOADVAR:
      stack[top++] = vars[getVar(pc+1)]
      break
    case OPCODE_JMPIFBEQ:
      top -= 2
      if(stack[top+1]>=stack[top+2]):
        pc = getImmediate(pc+1)
        continue
      break
    case OPCODE_MULTIPLY:
      top -= 1
      stack[top] = stack[top]*stack[top+1]
      break
    case OPCODE_ADD:
      top -= 1
      stack[top] = stack[top]+stack[top+1]
      break
    case OPCODE_JMP:
      pc = getImmediate(pc+1)
      continue
    case OPCODE_RET:
      return stack[top]

  pc += lenghtOf(opcode)
```

Listing 4 The interpreter from Listing 3 after being specialized for Listing 2 and the MERGE_EXPLODE strategy being applied.

```
; i=0
0: stack[0] = 0
1: vars[i] = stack[0]

; i < 100
2: stack[0] = vars[i]
3: stack[1] = 100
4: if(stack[0]>=stack[1]) goto @14

; a = a*a
5: stack[0] = vars[a]
6: stack[1] = vars[a]
7: stack[0] = stack[0] * stack[1]
8: vars[a] = stack[0]

; i++
9: stack[0] = vars[i]
10: stack[1] = 1
11: stack[0] = stack[0] + stack[1]
12: vars[i] = stack[0]

; loop
13: goto @2

; return a
14: stack[0] = vars[a]
15: return stack[0]
```

Listing 5 The code from Listing 4 after virtualizing the stack array and aggresive optimizations.

```
  vars[i] = 0;

condition:
  if(vars[i]>=100) goto end
  vars[a] = vars[a]*vars[a]
  vars[i] = vars[i]+1
  goto condition

end:
  return vars[a]
```

Chapter 3

CLI Component parser

Before being able to execute any code, it is necessary to read the code from the assemblies. Prior to starting the work, we expected some open source parsers for this format to exist for various languages, including Java. However, the only alternative stand-alone parser (not a component of a full CLI implementation) we found was dnlib[1] targeting .NET framework itself. If even for such a popular runtime there are no suitable parsers implemented in Java, we feel that the parser implementation step is an important part to consider in the whole "Building an experimental runtime" picture.

3.1 Analysis

3.1.1 Design goals

Before we design and implement the parser, we consider what additional constraints have to be put on a parser in order for it to be partial-evaluation friendly. For partial-evaluation friendliness, the key metric is how trivial can every piece of code get after partial evaluation. This metric is most important for a sequence of instructions executed frequently, often referred to as a "hot path". While our goal was for the parser to never be called on a hot path, for some scenarios, including reflection, it would be necessary.

There are two possible extremes for parser design: "fully lazy" where every query for the file causes it to be parsed from the start, and "fully preloaded" where all the data from the file is immediately fully parsed into hierarchies of objects and structures. Practical parsers usually choose a compromise between those two approaches, mainly because the extremes lead to extremely slow runtime or boot-up, respectively.

[1] `https://github.com/0xd4d/dnlib`

Driven by the goal of partial-evaluation friendliness, we design the initial parsing such that:

- trivial queries, e.g. queries for a metadata item at a constant index, will only result in a compilation constant,
- simple queries, e.g. queries for a metadata item at a variable index, will get compiled to a simple offset calculation (multiply and add) and a read from a (compilation constant) byte array,
- all further parsing necessary for more complex queries (creating objects representing metadata concepts etc.) will be lazy and invokers should cache the results themselves.

3.1.2 Definition of important CLI component structures

Most of the metadata are stored in streams, with headers outside of the streams describing their locations. There are two basic types of streams: heaps and tables.

Heaps contain a sequence of bytes, the meaning of which changes based on the specific heap. The specification defines 4 heaps:

- `#Strings` containing values of identifier strings.
- `#US` containing "user strings" — values of strings used by the program code itself during runtime.
- `#Blob` containing variable-length metadata as binary blobs.
- `#GUID` containing GUIDs.

The tables are stored in a stream called `#~`. This is the root of all metadata information. The specification describes 38 tables. Cell values can be a constant or an index. Indices can point to heaps (the value is a byte offset), another table (the value is a row number), or one of multiple tables (the value is a "coded index" specifying both the table and row number).

For an example of references between these structures, Table 3.1 describes a sample row in the metadata table `TypeDef`, which contains definitions of types.

3.1.3 Complexities of the CLI component format

Subjectively, we feel the format used by CLI components is not designed well with regard to supporting different parsing approaches and platforms. To substantiate this claim, we want to highlight several factors that complicate parsing the components and had to be considered in the design.

Column	Raw value	Comment
Flags	0x100000	Constant bitmask specifying TypeAttributes.
TypeName	0x01A9	An offset into the `#String` heap, where the name of the type can be found. In this example, `Program` was written there.
TypeNamespace	0x01DE	An offset into the `#String` heap, where the namespace of the type can be found. In this example, `SampleProject` was written there.
Extends	0x0031	A coded index into `TypeDef`, `TypeRef`, or `TypeSpec`. In this example an index to `TypeRef` table row 12, which is a reference to `System.Object`.
FieldList	0x0002	An index into the `Field` table where the fields for this type start. As in this case the type has no fields, the index points past the end of the `Field` table, which has only 1 row.
MethodList	0x0002	An index into the `Method` table where the methods for this type start. As this type has multiple methods, row 2 of `Method` contains information about `Main`, other methods follow.

Table 3.1 Description of a sample row from the `TypeDef` metadata table.

PE Wrapping

As stated in *II.25 File format extensions to PE*[4]:

> The file format for CLI components is a strict extension of the current Portable Executable (PE) File Format. [...] The PE format frequently uses the term RVA (Relative Virtual Address). An RVA is the address of an item *once loaded into memory* [...].
>
> The RVA of an item will almost always differ from its position within the file on disk. To compute the file position of an item with RVA r, search all the sections in the PE file to find the section with RVA s, length l and file position p in which the RVA lies, ie $s \leq r < s + l$. The file position of the item is then given by $p + (r - s)$.

On Windows and other theoretical platforms where PE parsing is a service provided by the operating system, this allows for the component to be loaded into virtual memory as any other executable file. RVA addresses can then be resolved transparently by the CPU's and operating system's virtual memory mappings. For all other platforms, this adds one more level of indirection that needs to be handled.

Offset	Size	Field	Description
6	1	HeapSizes	Bit vector of heap sizes
8	8	Valid	Bit vectors of present tables
24	4*n	Rows	Array of n 4-byte unsigned integers indicating the number of rows for each present table

Table 3.2 Selected fields from #~ stream header (shortened, for full structure see *II.24.2.6 #~ stream*)[4]

As our parser is platform-agnostic and written in Java, we cannot use any of those services. Therefore, we need to manually perform the sections search and RVA calculations as described in the standard.

Metadata tables format

The biggest complexity we encountered during parser design was the format of metadata tables. These tables contain most of the metadata information of the CLI component.

The data of the tables is stored in the #~ stream. This stream consists of a header followed by a simple concatenation of values of all rows of all tables, with no additional metadata in between.

The header itself contains only a few fields relevant for locating data in the tables, as listed in table 3.2.

The first issue is that no information about table length in bytes is present. This results in *every single parser implementing the format having to implement the format for every single metadata table*, as skipping a table requires knowing the byte length of its rows. This completely prohibits an iterative development cycle that adds support for only the necessary tables. For example, to implement a utility that only outputs names of all the types available in the component, while only data from the TypeDef table is necessary, all 38 tables defined by ECMA-335[4] must be implemented. The BACIL implementation described here only accesses 11 of these tables.

The second complication comes in *II.22 Metadata logical format: tables* and *II.24.2.6 #~ stream*[4]:

> Each entry in each column of each table is either a constant or an index.
>
> [...]
>
> Each index is either 2 or 4 bytes wide. The index points into the same or another table, or into one of the four heaps. The size of each index column in a table is only made 4 bytes if it needs to be for that

particular module. So, if a particular column indexes a table, or tables, whose highest row number fits in a 2-byte value, the indexer column need only be 2 bytes wide. Conversely, for tables containing 64K or more rows, an indexer of that table will be 4 bytes wide.

[...]

If e is a *coded index* that points into table t_i out of n possible tables $t_0, \dots t_{n-1}$, then it is stored as $e << (\log n) | tag\{t0, \dots t_{n-1}\}[t_i]$ using 2 bytes if the maximum number of rows of tables $t_0, \dots t_{n-1}$, is less than $2^{(16-(\log n))}$, and using 4 bytes otherwise.

While the decision that the cell can be either 2 or 4 bytes saves storage size, it means that table row length is not a constant and depends on the row count of other tables. For example, a `TypeDef` table row can be from 14 up to 24 bytes in size.

This means that the parser cannot workaround the first issue by expecting the table row length be constant.

If we were to improve the format to remove these issues, we would add information about the row length of present tables into the header. Even if each size was stored as a full byte (which all tables defined in the standard fit into), in the worst case this would increase each binary's size by 38 bytes and allow for skipping tables without dealing with their internal row format.

Extensive normalisation

File format design is often a compromise between several engineering goals: as described by Syreeni [9], "we face a multitude of engineering goals, some of which are mutually incompatible. [...] Usually tension develops between size and speed, generality and encoding simplicity, and consistence and software complexity". One of the design concepts that apply is normalisation, a concept that each information should be stored only once, removing all redundancy. While such a goal can be beneficiary for other uses of the file format (like writing and modifying), from the point of view of a lightly preloading consumer, it results in non-ideal structures.

- In parent-child relationships, only one node has a direct reference to the other one. Traversing the edge backwards involves enumerating all the nodes and searching for one with the appropriate reference. If such queries are performance sensitive, the invoker has to cache the answers.

- When referencing a sequence of items in a table, only information about the beginning of the sequence is directly stored. The end of the sequence is

Listing 6 A specification of the `TypeDef` table (see sample row in Table 3.1) to be processed by a parser code generator.

```
TypeDef:02
-Flags:c4
-TypeName:hString
-TypeNamespace:hString
-Extends:iTypeDef|TypeRef|TypeSpec
-FieldList:iField
-MethodList:iMethodDef
```

either the last row of the table or the start of the next sequence, as specified by the next row, whichever comes first.

While the complexity this adds usually amounts to a single "if" statement, it crosses the border between cell value semantics and metadata logical format internals — either the parser has to understand the semantics of cells as "sequence indices" to encapsulate resolving the sequence length, or the invoker has to understand the file format's internal detail of row numbers.

3.2 Parser implementation details

3.2.1 Metadata tables parser

As mentioned in Section 3.1.3, parsing any metadata tables requires implementing the internal row format for all tables specified in ECMA-335[4]. Implementing all 38 tables manually would require a sizeable amount of work and make modifications to the parser complicated. Therefore, this problem is a nice match for code generation.

We created a simplified text-file containing information about all the columns in all tables that is also human readable. Listing 6 shows an example specification of the `TypeDef` table.

For simplicity, we write the code generator in plain Java, outputting Java source files. The result is a `CLITableClassesGenerator` class.

3.2.2 CLITableRow and CLITablePtr

We want the implementation of accessing metadata table rows to be as safe and simple-to-use as possible while keeping in mind the design goals for partial

Listing 7 Code pattern for enumeration of all `MethodDef` rows, here printing the method's name.

```
CLIComponent component = ...;
for (CLIMethodDefTableRow methodDefTableRow : component.
    ↪ getTableHeads().getMethodDefTableHead()) {
     System.out.println(methodDefTableRow.getName().read(
        ↪ component.getStringHeap()));
}
```

Listing 8 Code pattern for safe resolving of table references, here resolving a type's first field.

```
CLIComponent component = ...;
CLITypeDefTableRow typeDef = ...;

CLIFieldTypeRow firstField = component.getTableHeads().
    ↪ getFieldTableHead().skip(typeDef.getFieldList());
```

evaluation. The two operations we expect to be most common are enumerating a single table and resolving indices that reference other tables.

To support enumeration we make `CLITableRow` implement `Iterable`, allowing for a safe for-each access, completely hiding the internal table details. See Listing 7 for an example of printing all methods defined in a component.

For resolving indices, we made the tables return a `CLITablePtr` wrapped index. Such a pointer can then be directly provided to `CLITableRow`'s `skip` method, which validates that the table ID is correct. The importance of this wrapping is increased by the following fact mentioned in *II.22 Metadata logical format: tables*[4]: "Indexes to tables begin at 1, so index 1 means the first row in any given metadata table. (An index value of zero denotes that it does not index a row at all; that is, it behaves like a null reference.)"

Exposing the indices as raw integers would allow for off-by-one bugs to become prevalent. Providing a wrapped variant that behaves as expected by default helps combat these issues. For an example of safe index resolving, see Listing 8.

3.2.3 Sequence references

As mentioned in Section 3.1.3, sequences of items in a table are stored in a way that requires either implementing column semantics in the parser or the invoker knowing logical table internals. As we generate our table parsers from a definition

Listing 9 Implementation of item sequence semantics by the invoker, which has to resolve the sequence end based on the next row. This example comes from the CLIType class.

```
if(type.hasNext())
{
    methodsEnd = type.next().getMethodList().getRowNo();
    fieldRowsEnd = type.next().getFieldList().getRowNo();
} else {
    methodsEnd = component.getTablesHeader().getRowCount(
        ↪ CLITableConstants.CLI_TABLE_METHOD_DEF)+1;
    fieldRowsEnd = component.getTablesHeader().getRowCount(
        ↪ CLITableConstants.CLI_TABLE_FIELD)+1;
}
```

file, including sequence semantics would require expanding both the generator and the definition file. Instead, we leave the responsibility on the invoker, resulting in code in Listing 9.

3.3 Conclusion

In the end, design and implementation of the parser took a non-trivial portion of the development time. Even though straightforward code-size indicators are controversial, they demonstrate the substantiality of the parser, being about 30% of the whole project. Excluding generated code it comprises 119594 bytes over 3609 lines of code, while the rest of the language package comprises 266010 bytes over 7534 lines.

Chapter 4

Runtime

4.1 Analysis

This chapter focuses on the overall design of our interpreter and the approaches required to achieve acceptable performance.

4.1.1 Nodes

Bytecode nodes

The smallest compilation unit of Truffle is a RootNode. This nomenclature comes from Truffle's original AST-based design, where nodes represented actual nodes in the syntax tree. The supported pattern was that the nodes were small, typically representing a single operation — for example, an `AddNode` that has two child nodes and adds them together.

However, this design is not applicable to our bytecode interpreter. Inside one method, the bytecode does not have any tree structure we could replicate with the nodes. The most straightforward solution is to have one node per one bytecode chunk, which in the CLI subset we implement corresponds to one method. We call such a node the `BytecodeNode`.

Having big nodes with several instructions has its tradeoffs, mainly the fact that without additional work (described below) once a node starts executing in a specific performance tier (see Section 2.2), it has to finish running in that specific tier. Truffle always starts executing code immediately in interpreted mode and only later considers compiling it. This can lead to poor results.

For example, let us consider the code in Listing 10 running in a one-node-per-method implementation. The runtime will immediately enter the `Main` node and start executing it in interpreter mode. As the entire execution time is spent in this one node, the runtime will never get a chance to run a single compiled statement.

Listing 10 Example of long-running code that will be compiled to a single node, bringing performance issues.

```
static int Main()
{
    int result = 0;
    for (int i = 0; i < 20000; i++)
    {
        for (int j = 0; j < 20000; j++)
        {
            result += i * j;
        }
    }
    return result;
}
```

This significantly affects the performance (see Section 5.3.1 for measurements of the slowdown in interpreted mode).

To solve this issue, a feature called On-Stack Replacement (OSR) is a part of Truffle[1]:

> During execution, Truffle will schedule "hot" call targets for compilation. Once a target is compiled, later invocations of the target can execute the compiled version. However, an ongoing execution of a call target will not benefit from this compilation, since it cannot transfer execution to the compiled code. This means that a long-running target can get "stuck" in the interpreter, harming warmup performance.
>
> On-stack replacement (OSR) is a technique used in Truffle to "break out" of the interpreter, transferring execution from interpreted to compiled code. Truffle supports OSR for both AST interpreters (i.e., ASTs with LoopNodes) and bytecode interpreters (i.e., nodes with dispatch loops). In either case, Truffle uses heuristics to detect when a long-running loop is being interpreted and can perform OSR to speed up execution.

While current Truffle versions support OSR for both AST interpreters and bytecode interpreters, OSR support for bytecode interpreters was only introduced in Graal 21.3 released in October 2021, which means it was not available during our design phase.

[1] https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/OnStackReplacement/

Listing 11 A small C program containing a loop. Rigger et al. [6]

```
void processRequests () {
    int i = 0;
    do {
        processPacket();
        i++;
    } while (i < 10000);
}
```

Listing 12 LLVM IR of the C program. Rigger et al. [6]

```
define void @processRequests () #0 {
; (basic block 0)
    br label %1

; <label>:1 (basic block 1)
    %i.0 = phi i32 [ 0, %0 ], [ %2, %1 ]
    call void @processPacket()
    %2 = add nsw i32 %i.0, 1
    %3 = icmp slt i32 %2 , 10000
    br i1 %3, label %1, label %4

; <label >:4 ( basic block 2)
    ret void
}
```

Rigger et al. [6] described a method of separating the bytecode chunk into "basic blocks", each containing only instructions that do not affect the control flow. This method allowed them to take advantage of the OSR support for ASTs in what is actually a bytecode interpreter. A "block dispatcher" controls the flow between these basic blocks, selecting the adequate basic block to continue the execution with. This approach is summarized in Listings 11 and 12 and figure 4.1 from their work.

Our design is not compatible with the bytecode interpreter OSR — to support it, the whole execution state has to be stored in Truffle's frames, while we only use frames to pass/receive arguments and store the rest of the state in plain variables. However, moving this state into the frame should be the only major step necessary to support OSR. Because of the significance of OSR, if we were designing the runtime again, we would definitely focus on supporting it.

We also did not implement the "basic blocks" pattern as we decided against synthesizing any AST structures, as discussed in Section 1.4.

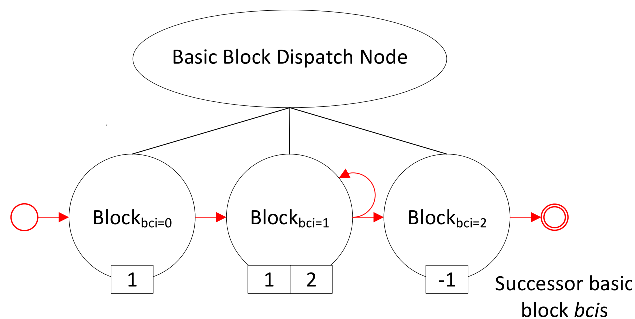

Figure 4.1 Basic block dispatch node for the LLVM IR. Rigger et al. [6]

Instruction nodes

Some instructions require values that can be pre-calculated. A typical example in CIL are instructions that have a token as its argument — a token is a pointer into metadata tables and requires calling into the parser to resolve. We want to perform this resolution only once and cache it for future executions.

For that, we will use a process of nodeization[2] — we create a node representing the instruction with the data already pre-computed and patch the bytecode, replacing the original instruction with a BACIL-specific `TRUFFLE_NODE` opcode. When the interpreter hits this instruction, it calls the respective child node.

4.1.2 Dynamicity of references

One of the additional things to consider when implementing a partial-evaluation friendly interpreter is dynamicity of references, whereby dynamicity we mean how often the reference changes its state. This metric is important because, effectively, the dynamicity of a chain of references will be equal to the most dynamic of the references. As a result, what would traditionally be considered bad design patterns is sometimes necessary to divide the chain into more direct references, in order to make each object reachable with the lowest dynamicity possible. Figures 4.2 and 4.3 show the refactoring in a generic case.

For a case study from the BACIL implementation, let us consider the design decisions behind `LocationDescriptor` and `LocationHolder`. Each location has a type and a value. While the value itself (and the type of the value) changes based on the running code, the type of the location never changes. This is a perfect example of two pieces of information with different dynamicity.

Even from regular development patterns, it makes sense to divide location values and location types into separate classes — store the location type information in the metadata as a “prototype” for later creating the value storage

[2] called “quickening” by Espresso, the Java bytecode interpreter for GraalVM

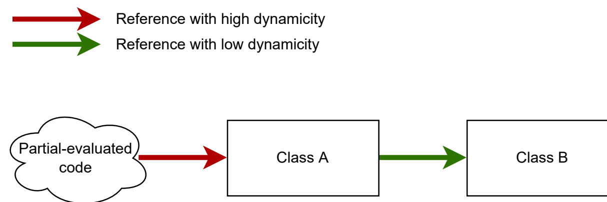

Figure 4.2 Generic scenario 1: Reference chain results in class B being accessible with high dynamicity and therefore not being effectively partially evaluated.

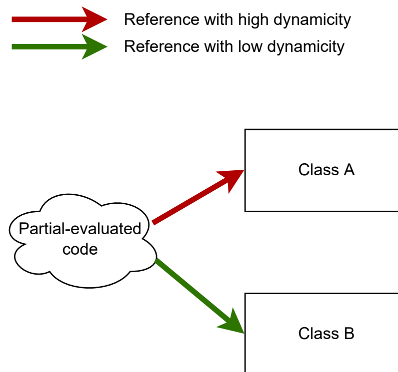

Figure 4.3 Generic scenario 2: Class B is accessible with a low dynamicity reference, resulting in more effective partial evaluation.

Listing 13 A naive implementation of LocationsHolder resulting in non-optimal dynamicity chain.

```
public class LocationHolder {
    private final LocationDescriptor descriptor;

    private final Object[] refs;
    private final long[] primitives;

    public LocationsHolder(LocationDescriptor descriptor) {
        this.descriptor = descriptor;
        refs = new Object[descriptor.getRefCount()];
        primitives = new long[descriptor.getPrimitiveCount()];
    }

    public Object locationToObject(int locationIndex)
    {
        return descriptor.locationToObject(this, locationIndex);
    }

}

//Accessing a field of an object
Object fieldValue = ((StaticObject)object).getLocationsHolder().
    ↪ locationToObject(0);
```

based on it. In BACIL, `LocationDescriptor` contains the type information and `LocationHolder` contains the actual values.

It is always necessary to know the location type to work with the values, mainly to differentiate between ValueTypes and references. The rule of encapsulation would dictate that the consumer does not need to know that there is a `LocationDescriptor` tied to the `LocationHolder`, as it is an internal detail. Such an implementation would look something like Listing 13. However, using such code results in a non-optimal dynamicity chain and ineffective partial evaluation, as illustrated in Figure 4.4.

In order to make this more effective, we have to hold a separate reference to a `LocationDescriptor`. As every location-accessing instruction (in the implemented subset of CLI) will always use the same `LocationDescriptor`, this results in effective partial evaluation. The new implementation is in Listing 14 and the dynamicity is illustrated in Figure 4.5.

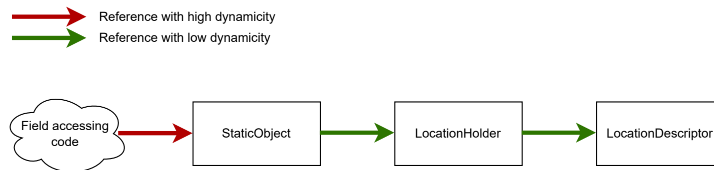

Figure 4.4 Case study cenario 1: As a LocationHolder is unique per object instance/method invocation/ etc., the reference to it is highly dynamic. The LocationDescriptor is only unique per object type/method definiton, but can only be reached through a dynamic chain.

Listing 14 A partial-evaluation friendly implementation of LocationsHolder.

```
public class LocationHolder {

    private final Object[] refs;
    private final long[] primitives;

    public LocationsHolder(int refCount, int primitiveCount) {
        refs = new Object[refCount];
        primitives = new long[primitiveCount];
    }
}

//Accessing a field of an object
//objectType for an instruction never changes!
Object fieldValue = objectType.getLocationsDescriptor().
   ↪ locationToObject(((StaticObject)object).getLocationsHolder
   ↪ (), 0);
```

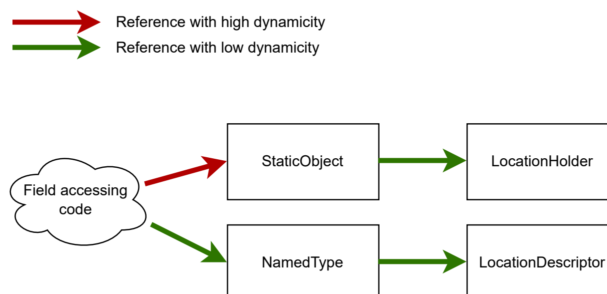

Figure 4.5 Case study cenario 2: While the LocationHolder remains accessible from a highly dynamic chain, the LocationDescriptor is accessible through a static chain. This means the bottom chain will be partially evaluated.

4.1.3 Standard libraries

Our goal was to implement as little of the standard library as possible. When starting with the implementation, we hoped parts implemented in CIL and native methods would be well decoupled, so that we could reuse all CIL parts. For the native parts, we would call .NET's native implementation if possible and only implement them in BACIL if not. Unfortunately, the coupling is tight, as the documentation[3] admits: "CoreLib has several unique properties, many of which are due to its tight coupling to the CLR."

The biggest offender are strings. To achieve high performance, .NET's runtime expects the native view (`StringObject`) and managed view (`System.String`) of strings to be identical, so the coupling is extremely tight. To support string operations in BACIL, we would either have to reimplement the operations or implement the strict marshalling that is expected by .NET's native code.

Truffle's official mechanism of calling into native code is called the Native Function Interface (NFI). Unfortunately, at the time of designing BACIL, it was missing key features, for example support of custom ABIs (calling conventions). While the limitations can be bypassed (for example by using custom trampolines provided by BACIL), the necessary marshalling could lead to the native calls imposing a significant performance hit.

In the end, our experiments with calling native .NET runtime code using NFI showed it would significantly complicate the whole codebase with uncertain results, and we decided against it. Unfortunately, we will have to re-implement all necessary native code ourselves in BACIL.

4.1.4 BACILHelpers

To provide additional BACIL APIs, we need to expose our own "native" BACIL functionality to C# code. For that, we create an assembly called BACILHelpers. This assembly has two implementations: a proper .NET variant (written in C#) for running on .NET, and a "virtual" assembly leading to BACIL's internal methods that BACIL silently injects.

Listings 15 and 16 show an example of how we can implement a `BACILConsole.Write` method in the C# variant and in BACIL, respectively.

4.1.5 Values and locations

As specified in ECMA-335[4], values can have the following "homes":

[3] `https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/corelib.md`

Listing 15 An implementation of the BACILConsole helper class in C#.

```
public class BACILConsole
{
    public static void Write(Object value)
    {
        Console.Write(value);
    }
}
```

Listing 16 An implementation of the BACILConsole helper class in BACIL itself.

```
public class BACILHelpersComponent extends BACILComponent {
    public Type findLocalType(String namespace, String name) {
        if(name.equals("BACILConsole"))
            return new BACILConsoleType();
    }
}

public class BACILConsoleType extends Type {
    public BACILMethod getMemberMethod(String name, MethodDefSig
      ↪ signature) {
        if(name.equals("Write"))
            return new BACILConsoleWriteMethod();
    }
}

public class BACILConsoleWriteMethod extends JavaMethod {
    public Object execute(VirtualFrame frame) {
        CompilerDirectives.transferToInterpreter();
        System.out.print(frame.getArguments()[0]);
        return null;
    }
}
```

I.12.1.6.1 Homes for values

The **home** of a data value is where it is stored for possible reuse. The CLI directly supports the following home locations:

- An incoming **argument**
- A **local variable** of a method
- An instance **field** of an object or value type
- A **static** field of a class, interface, or module
- An **array element**

[...]

In addition to homes, built-in values can exist in two additional ways (i.e., without homes):

1. as constant values (typically embedded in the CIL instruction stream using `ldc.*` instructions)
2. as an intermediate value on the evaluation stack, when returned by a method or CIL instruction.

As Truffle requires Java objects to be passed on the node boundary, BACIL also has an additional state where the value is a Java object.

In .NET, all locations are typed (*I.8.6.1.2 Location signatures*[4]). While evaluation stack slots are also typed, they use a different and more coarse type system.

To avoid boxing and unboxing numbers (integers and floating-point numbers), we cannot just store all values in an `Object[]`. Therefore, it is necessary to always have separate storages for primitives, best implemented by a `long[]`.

Locations usually exist in multiples (local variables, arguments, fields, etc.) and are always statically typed — one location will always have one type through its lifetime and only ever contain values type-compatible with its type. We divide the location into two parts: a descriptor and a holder.

The holder is actually extremely simple: it only has an `Object[] refs` and a `long[] primitives` that are big enough to hold all the values required by the descriptor. The holder knows nothing of the types or identities of values inside. This represents one instance of a value storage.

The descriptor represents the "shape" of the locations, knowing the type of each location and its position in the holder.

One feature of ECMA-335 is so-called user-defined ValueTypes, structures that have the semantics of a primitive. The idea is that two integers (x,y) and a Point structure (with x,y fields) will look exactly the same on the stack, instead of the latter turning into an object reference. Our implementation will follow that

Listing 17 Prototypes of state transition operations implemented by Type.

```
public void stackToLocation(LocationsHolder holder, int
    ↪ primitiveOffset, int refOffset, Object ref, long primitive)
public void locationToStack(LocationsHolder holder, int
    ↪ primitiveOffset, int refOffset, Object[] refs, long[]
    ↪ primitives, int slot)
public Object stackToObject(Object ref, long primitive)
public void objectToStack(Object[] refs, long[] primitives, int
    ↪ slot, Object value)
public Object locationToObject(LocationsHolder holder, int
    ↪ primitiveOffset, int refOffset)
public void objectToLocation(LocationsHolder holder, int
    ↪ primitiveOffset, int refOffset, Object value)
```

example, as we will “flatten” the structure, reserving space in the ValueType’s parent for each of its fields.

The evaluation stack is a bit more complicated: while at each point in time the type of the evaluation stack field is known, it changes throughout execution. Each stack slot therefore has to exist as both a reference slot and a primitive slot and we will have to keep track of which one to use. To achieve that, we by default expect the value to be in the refs slot. In case it is not, we repurpose the refs slot to hold a “marker” object describing the stack-type of the value in the primitive slot. An object will be stored in `(ref, primitive)` as `(obj, undefined)`, a native int 42 will be stored as a `(EvaluationStackPrimitiveMarker.EVALUATION_STACK_INT, (long)42)`.

State transitions

Keeping in mind the dynamicity of references (see Section 4.1.2) and the fact that all locations are typed (and the type of a location never changes), it follows to make the Type objects responsible for implementing the state transitions. There are 6 possible transitions between objects, evaluation stack and locations implemented as methods of Type, as declared in listing 17. The Type class will provide the default transitions for reference types, while subclasses of this class can provide special variants for primitives. A `LocationDescriptor` can then resolve the (compilation constant) location type and use it to transition the value, for example like in Listing 18.

One factor to keep in mind is that to follow the standard, the state transitions are coupled with widening or narrowing operations. For example, according to *I.12.1 Supported data types*[4] “Short numeric values (int8, int16, unsigned int8,

Listing 18 Example implementation of transitioning a value from a location to the evaluation stack, as implemented in `LocationDescriptor`.

```
public void locationToStack(LocationsHolder holder, int
    ↪ locationIndex, Object[] refs, long[] primitives, int slot)
{
    locationTypes[locationIndex].locationToStack(holder, offsets
        ↪ [locationIndex], refs, primitives, slot);
}
```

and unsigned int16) are widened when loaded and narrowed when stored". We also need to perform our own housekeeping because we store all primitives in a flat `long[]`. Each class representing a primitive implements its own widening and narrowing as necessary.

4.1.6 CompilationFinal annotation

As explained in Section 2.1, one of the key decisions is separating inputs into two sets — dynamic inputs and static inputs. Java's `final` keyword is therefore integral for achieving performance, as it guarantees that the variable will be considered a static input.

There is an issue that, for arrays, marking them as `final` only means that the reference to the array does not change, while the contents of the array can change freely. The solution is the `CompilerDirectives.CompilationFinal` annotation, which can mark arrays such that the compiler considers reads with a constant index as constants. Unlike the built-in `final` keyword, the compiler cannot actually enforce that no writes happen to the array. It is the responsibility of the implementation to always invalidate the current compilation when modifying a `CompilationFinal` array.

4.2 Debugging performance issues

To achieve high performance, the ability to debug performance issues is necessary. Unfortunately, traditional methods (like sampling) do not provide the required insight for outputs of Graal compilation — during partial evaluation, the code gets transformed too much for these methods to work properly. A single instruction can result from a partial evaluation of several methods and, as such, cannot be attributed properly to one.

Internally, the Graal compiler represents the code during compilation in graphs. The various optimization phases are then transformations on these

graphs. Graal allows to dump the current graph in various stages of the compilation pipeline by using the `graal.Dump` VM argument. These graphs are key to understanding results of the partial evaluation, mainly which code was eliminated (by constant folding) and which remained. For our analysis, the "After TruffleTier" phase is the most important, as it reflects the graph state after partial evaluation.

The official tool for analyzing these graphs is the Ideal Graph Visualizer[4]. However, getting it requires "accepting the Oracle Technology Network Developer License", which contains strict limitations for allowed use. We do not consider the tool suitable for general use, as using it during development may limit the future uses of the project.

Fortunately, an MIT licensed open source project Seafoam[5] provides all the necessary functionality. Compilation graphs in this work were all generated by Seafoam.

The most common issue we hit when analyzing those graphs was that a piece of code we expected to be eliminated by partial evaluation was still included in the compilation — we designed it to be optimized out, but from the view of the compiler it could not be. To debug these issues, we used the `CompilerAsserts.partialEvaluationConstant` method. It allows us to express our belief that something should be a partial evaluation constant to the compiler and get an error message with detailed information about the compiler's view of the expression when it is not.

4.2.1 Case study

For a case study, let us look into optimizing a specific parser call. As specified in section 3.1.1, we designed the parser so that trivial queries, e.g. queries for a metadata item at a constant index, would only result in a compilation constant. Is that the case? See Figure 4.6 for the compilation graph (before optimizations[6]) after TruffleTier for a function returning `method.getComponent().getTableHeads()`↪ `.getTypeDefTableHead().skip(1).getFlags()` (henceforth referred to as "the case study parser query") — it definitely contains more than just a constant. Using `CompilerAsserts.partialEvaluationConstant` (see Listing 19) and checking the errors[7] leads us to an error in the second expression. From this we can discern that our implementation of `skip` is at fault.

Our skip method contains just one statement, `return createNew(tables, cursor+count*getLength(), rowIndex+count);`. We begin by validating

[4]`https://www.graalvm.org/22.1/tools/igv/`

[5]`https://github.com/Shopify/seafoam`

[6]commit 42e5cb2e6e34956aca75be0c4c71ac7eb0f4bea8

[7]enabled with `--engine.CompilationFailureAction=Print`

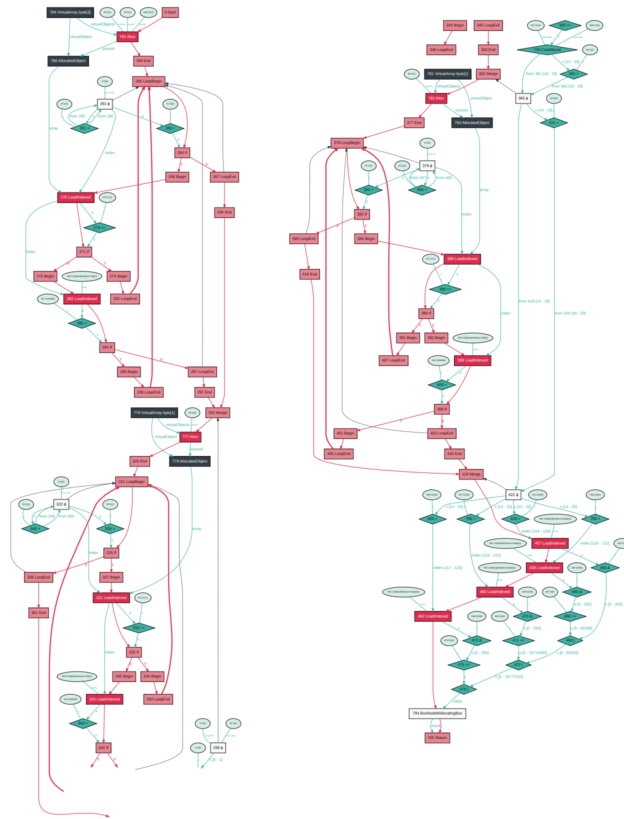

Figure 4.6 The compilation graph of the case study parser query before being optimized (wrapped, included for overview — not expected to be readable).

Listing 19 Code using compiler asserts to check compilation issues in the case study parser query. Running it reveals that the second expression is at fault.

```
CLITypeDefTableRow row = method.getComponent().getTableHeads().
    ↪ getTypeDefTableHead();
CompilerAsserts.partialEvaluationConstant(row.getFlags());
CompilerAsserts.partialEvaluationConstant(row.skip(1).getFlags()
    ↪ );
```

Listing 20 Code using compiler asserts to check compilation issues in CLITableRow.skip. Running it reveals an error in getLength.

```
public final T skip(int count)
{
    CompilerAsserts.partialEvaluationConstant(tables);
    CompilerAsserts.partialEvaluationConstant(cursor);
    CompilerAsserts.partialEvaluationConstant(getLength());
    return createNew(tables, cursor+count*getLength(), rowIndex+
      ↪ count);
}
```

that all used arguments are constants, as in Listing 20. We get an error[8] when checking the `getLength()` statement. We then check `getLength()`, enhancing it with asserts to see if `isStringHeapBig()` or `areSmallEnough` are at fault, as seen in Listing 21. We get an error on a call to `areSmallEnough`.

We continue digging into that method, eventually realising that inside `areSmallEnough(byte... tables)`, the `tables[0]` is not a constant! Turns out that even though the calls to `areSmallEnough` look to just be providing constant integers, as the function is using varargs a *new array is allocated* for those constants that is then passed to the function. As we explain in Section 4.1.6, array elements are not considered to be constant unless the array is properly annotated as `CompilationFinal`. Our inplace arrays have no way to be annotated. We modified `areSmallEnough` to take a `byte[]` instead of varargs (to make what is happening more obvious) and instead of using inplace arrays used constant fields annotated with `@CompilerDirectives.CompilationFinal(dimensions = 1)`.

After two additional small changes (adding a field to cache `tableData` with the `CompilationFinal` annotation and annotating `areSmallEnough` with `@ExplodeLoop`), the graph for the case study parser query after TruffleTier changed to Figure 4.7.

This case study provides a great example of how choices that are functionally equivalent in regular Java can provide vastly different compilation results when partially evaluated. In the end, we mostly only had to add annotations to make a big difference.

We used this process of checking if graphs look as expected using Seafoam and then using `CompilerAsserts.partialEvaluationConstant` to express our desires about constants multiple times during development.

---

[8] `Partial evaluation did not reduce value to a constant, is a regular compiler node: 516|ValuePhi(459 515, i32) (513|Merge; 459|ValuePhi(401 458, i32); 515|+; )`

Listing 21 Code using compiler asserts to check compilation issues in CLITypeDefTableRow.getLength. Running it reveals an error in areSmallEnough.

```
public int getLength() {
    int offset = 14;
    CompilerAsserts.partialEvaluationConstant(tables.
      ↪ isStringHeapBig());
    CompilerAsserts.partialEvaluationConstant(areSmallEnough(
      ↪ CLITableConstants.CLI_TABLE_TYPE_DEF, CLITableConstants
      ↪ .CLI_TABLE_TYPE_REF, CLITableConstants.
      ↪ CLI_TABLE_TYPE_SPEC));
    if (tables.isStringHeapBig()) offset += 4;
    if (!areSmallEnough(CLITableConstants.CLI_TABLE_TYPE_DEF,
      ↪ CLITableConstants.CLI_TABLE_TYPE_REF, CLITableConstants
      ↪ .CLI_TABLE_TYPE_SPEC)) offset += 2;
    if (!areSmallEnough(CLITableConstants.CLI_TABLE_FIELD))
      ↪ offset += 2;
    if (!areSmallEnough(CLITableConstants.CLI_TABLE_METHOD_DEF))
      ↪  offset += 2;
    return offset;
}
```

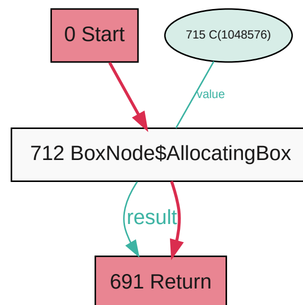

Figure 4.7 The compilation graph of the case study parser query after being optimized.

Chapter 5

Results

5.1 Completeness

Due to time constraints, several areas of the standard were ignored. The development focused on being able to run simple calculation programs and being able to run benchmarks from Hagmüller [7].

In total, ECMA-335[4] defines 219 opcodes, consisting of 6 prefixes and 213 instructions. Of those, our runtime contains code handling 151 instructions and no prefixes.

Notable missing features include:

- exceptions, overflow checking instructions
- interfaces
- generics
- casting and type checks
- visibility enforcement
- general arrays — only SZArrays (single dimensional, zero-based array) are supported
- operations requiring 64-bit unsigned integers
- unmanaged pointers and localloc

To validate the proper implementation of instructions, we used .NET's CodeGenBringUpTests[1]. According to the documentation[2], they are the recommended test suit to target when porting RyuJIT (.NET's JIT engine) to a new platform:

> **Initial bring-up**
>
> [...]
>
> Implement the bare minimum to get the compiler building and generating code for very simple operations, like addition. Focus on the CodeGenBringUpTests (src\tests\JIT\CodeGenBringUpTests), starting with the simple ones.

They are perfect for testing edge cases of implemented instructions. One example of a bug uncovered in this test suite that would be very hard to find manually was a missing int32 truncation[3].

After stubbing out `Write`, `WriteLine`, `ToString` and `Concat` (to get rid of the unsupported operations used by debug prints in case of failure), the BACIL implementation presented in this work passed 85%, e.g. 133 out of the 155 tests, included in the `v6.0.6` tag. All the failed tests were because of missing features and not bugs in implemented features, see Table 5.1 for details.

5.1.1 Library methods

While we pass library calls to the .NET runtime implementations, most of them either require generics or use native methods. We only implemented the following native methods:

- `System.Runtime.CompilerServices.RuntimeHelpers.↪ InitializeArray(Array, RuntimeFieldHandle)` used for constant array initializatons (like `int[] a = new int[] 0, 1`).
- Methods `System.Math.Abs(Double)`, `System.Math.Cos(Double)` and `System.Math.Sqrt(Double)` required by some float instruction tests.
- `System.ValueType.GetHashCode()` required by user-defined value types, as they override this virtual method.

[1] `https://github.com/dotnet/runtime/tree/main/src/tests/JIT/CodeGenBringUpTests`

[2] `https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/porting-ryujit.md`

[3] fixed in commit 056640ec276376434f5cb32ac70c3f9eb26c4881

Test	Reason for failure
ArrayExc	Missing exception support
ArrayMD1	Missing multi-dimensional array support
ArrayMD2	Missing multi-dimensional array support
div2	Missing exception support
DivConst	Missing exception support
FPConvI2F	Missing `conv.r.un` instruction implementation (due to unsigned 64-bit integers usage)
FPMath	Missing generics support (in `System.BitConverter::DoubleToInt64Bits` internally used by `System.Math::Round`)
LngConv	Missing generics support internally used by `System.IntPtr`
Localloc* (8 tests)	Missing support for localloc and raw pointers
ModConst	Missing exception support
RecursiveTailCall	Missing generics support
Rotate	Missing `volatile.` prefix
StructReturn	The test (incorrectly) compares floats using == instead of checking their difference against an epsilon
UDivConst	Missing exception support
UModConst	Missing exception support

Table 5.1 Reasons why specific `CodeGenBringUpTests` failed.

5.2 Performance benchmarks

We performed all benchmarks mentioned here on a laptop with an AMD Ryzen 7 PRO 4750U with 8 cores and 16 virtual threads, a base clock of 1.7GHz and boost up to 4.1GHz, featuring 32 GB of RAM and running Windows 10.

We ran the tests on GraalVM version 22.1.0[4] and .NET runtime version 6.0.6[5].

5.2.1 Harness

As mentioned in Section 4.1.4, our way of exposing additional functionality comprises implementing them in the `BACILHelpers` assembly, which .NET calls directly and BACIL replaces with its own implementation. To facilitate benchmarks, we add two new methods: `StartTimer` which starts a timer and `GetTicks` which return the number of ticks since the start. The API was inspired by the API

[4]openjdk 11.0.15 2022-04-19
OpenJDK Runtime Environment GraalVM CE 22.1.0 (build 11.0.15+10-jvmci-22.1-b06) OpenJDK 64-Bit Server VM GraalVM CE 22.1.0 (build 11.0.15+10-jvmci-22.1-b06, mixed mode, sharing)

[5]Version: 6.0.6
Commit: 7cca709db2

Listing 22 An implementation of the benchmark timer in C#.

```
static Stopwatch stopWatch = new Stopwatch();
static long nanosecPerTick = (1000L * 1000L * 1000L) / Stopwatch
    ↪ .Frequency;

public static void StartTimer()
{
    stopWatch.Restart();
}

public static long GetTicks()
{
    stopWatch.Stop();
    return stopWatch.ElapsedTicks * nanosecPerTick;
}
```

of `System.Diagnostics.StopWatch`, which is what the .NET implementation uses, as shown in Listing 22. On the BACIL side, we use `System.nanoTime()`, saving a value on start and the subtracting it from the value on end. Combining with the console writing capabilities, this our final harness is in Listing 23 (`DoCalculation` and the iteration count being replaced as necessary).

5.2.2 Hagmüller's work

One goal was to compare with the implementation of Hagmüller [7]. We received copies of the benchmark programs used in the work and could therefore run them against our implementation. When researching the .NET runtime JIT benchmarks in Section 5.3, we discovered that the used benchmarks are based on code from the repository. However, to keep the comparison fair, we used the provided modified versions, only changing out the harness. We did not have access to the interpreter itself, so we could not replicate the original benchmarks, and only use the numbers provided in their work. To receive comparable numbers, we followed the methodology outlined by Hagmüller [7]:

> The discussed Truffle CIL Interpreter, was evaluated by running a set of different programs. All benchmarks were executed on an Intel i7-5557U processor with 2 cores, 4 virtual threads featuring 16GB of RAM and a core speed of 3.1 GHz running macOS Catalina(64 bit).
>
> We parametrized each benchmark so that its execution results in high workload for our test system. In order to get a performance reference to compare with, we executed the benchmark programs in

Listing 23 Harness used for the benchmarks.

```
static void report(int iteration, long ticks, int result)
{
    BACILHelpers.BACILConsole.Write("iteration:");
    BACILHelpers.BACILConsole.Write(iteration);
    BACILHelpers.BACILConsole.Write(" ticks:");
    BACILHelpers.BACILConsole.Write(ticks);
    BACILHelpers.BACILConsole.Write(" res:");
    BACILHelpers.BACILConsole.Write(result);
    BACILHelpers.BACILConsole.Write("\n");
}

public static void Main(String[] args)
{
    int r;
    for(int i=0;i<1500;i++)
    {
        BACILHelpers.BACILEnvironment.StartTimer();
        var result = DoCalculation();
        long duration = BACILHelpers.BACILEnvironment.GetTicks()
            ↪ ;

        report(i, duration, result);
    }
}
```

	Debug BACIL	Debug .NET	Release BACIL	Hagmüller [7]
Binarytrees	2.573	2.003	2.649	24
Sieve of Eratosthenes	2.172	4.696	1.698	226
Fibonacci	1.635	4.449	1.512	38
Mandelbrot	5.612	2.666	4.779	38
N-Body	7.237	5.000	6.763	194

Table 5.2 Slowdown of benchmarks from Hagmüller [7] relative to .NET in release configuration.

> the mono runtime. We ran the benchmark programs in our Truffle CIL Interpreter on the top of the Graal VM. To find out how much our Truffle CIL Interpreter benefits from the support of compilation by Graal, we also ran the tests in an interpreter only mode, by using the standard Java JDK, instead of Graal. Because Graal optimizes functions which are called a certain number of times, we executed each program in a loop a several amount of times. For our evaluation we wanted to ignore the warm up phase of the compilation, so we just took the last 10 iterations of the execution loop. For each iteration the execution time is measured. For these 10 iterations we calculated the arithmetic mean. In order to reduce statistical outliers we repeated this 10 times and calculated the geometric mean over the arithmetic means.

Our benchmarks had the following differences:

- instead of an unspecified version of the mono runtime, we used .NET 6.0.301 to get the reference performance
- our system was different
- we ignored "interpreter only mode" results — we tailored the interpreter for partial evaluation, so the slowdowns in interpreter mode are usually more than 200x; we do not see value in precisely benchmarking such a glaring difference

It was not obvious if CIL-level optimizations were enabled when compiling the tests in Hagmüller [7]. For that reason, we measured both the debug (unoptimized) and release (optimized) compilation configurations.

The measured slowdowns (relative to .NET in release configuration) are shown in Table 5.2 and figures 5.1 and 5.2.

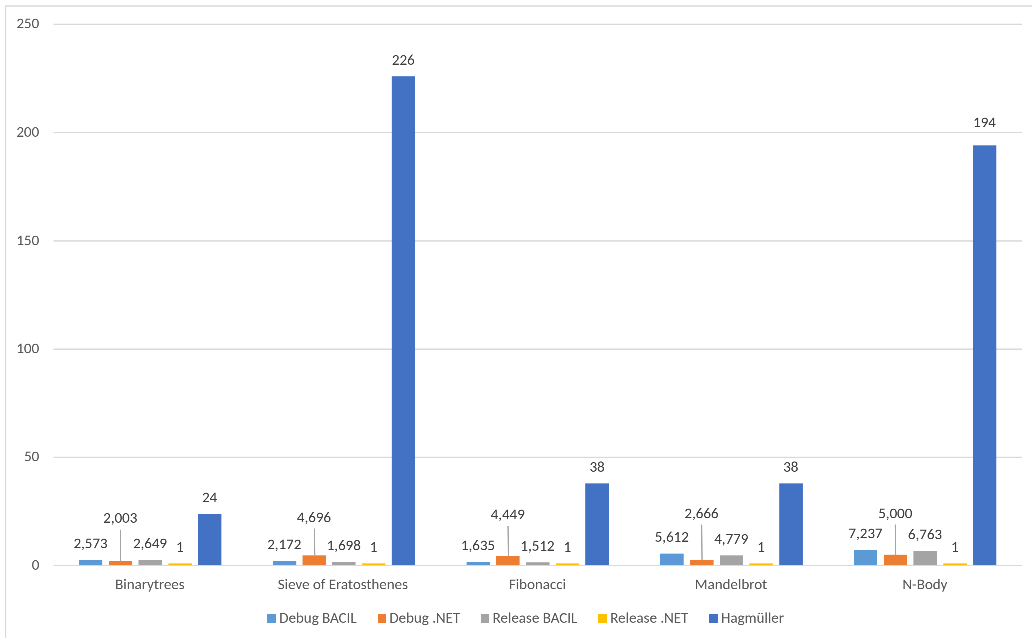

Figure 5.1 Slowdown of benchmarks from Hagmüller [7] relative to .NET in release configuration, including measurement from Hagmüller [7].

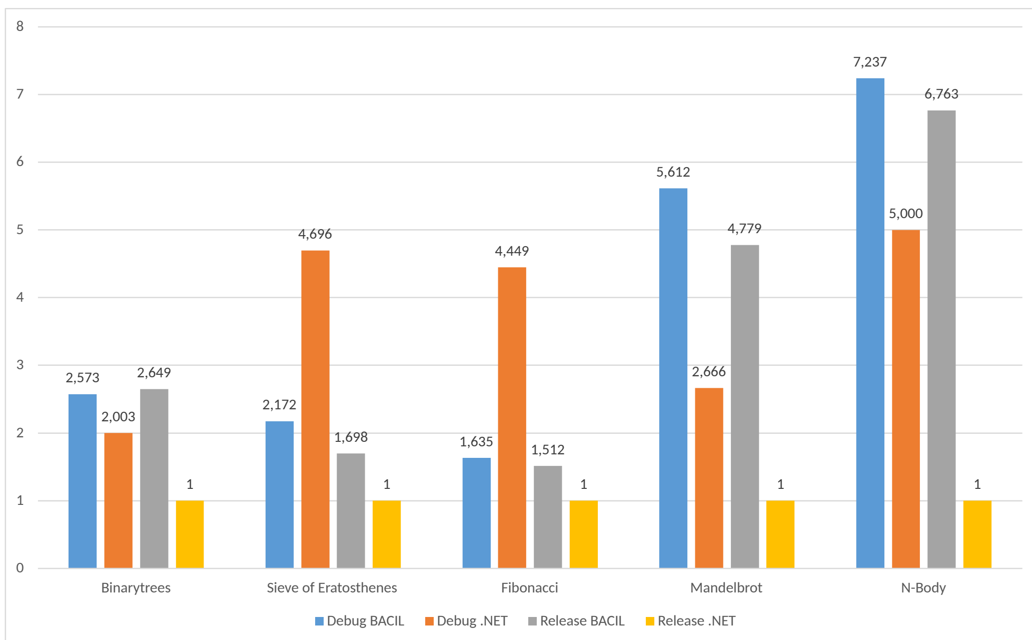

Figure 5.2 Slowdown of benchmarks from Hagmüller [7] relative to .NET in release configuration, without measurement from Hagmüller [7].

Benchmark	BACIL Slowdown
TreeInsert	1.036
Pi	1.125
HeapSort	1.172
Array1	1.322
QuickSort	1.428
Fib	1.519
BubbleSort	1.538
BubbleSort2	1.722
CSieve	1.741
fannkuch-redux-2	1.818
spectralnorm-1	2.063
MatInv4	2.351
8queens	2.355
Permutate	2.595
Ackermann	3.008
TreeSort	3.158
binarytrees-2	3.412
Lorenz	4.600
n-body-3	4.974

Table 5.3 BACIL slowdown of .NET's JIT benchamrks relative to .NET.

5.3 .NET runtime JIT benchmarks

To get more performance comparisons, we used (a subset of) .NET's JIT benchamrks[6]. The subset selection was driven by picking only tests using features the BACIL implements.

Originally, the tests use `Xunit` framework for benchmarks. Apart from switching the `Xunit` harness for our own, we made no other modifications to the code.

Our methodology was driven by our interest in getting results for as many binaries as possible, rather than making sure the comparison is extremely precise. We ran each test once, took the arithmetic average of iterations 250-299 and calculated the slowdown ratio between BACIL and .NET. We used the benchmarks as compiled by the .NET runtime compilation process without changing any settings regarding optimizations. The results are shown in Table 5.3 and figure 5.3.

[6] `https://github.com/dotnet/runtime/tree/main/src/tests/JIT/Performance/CodeQuality`

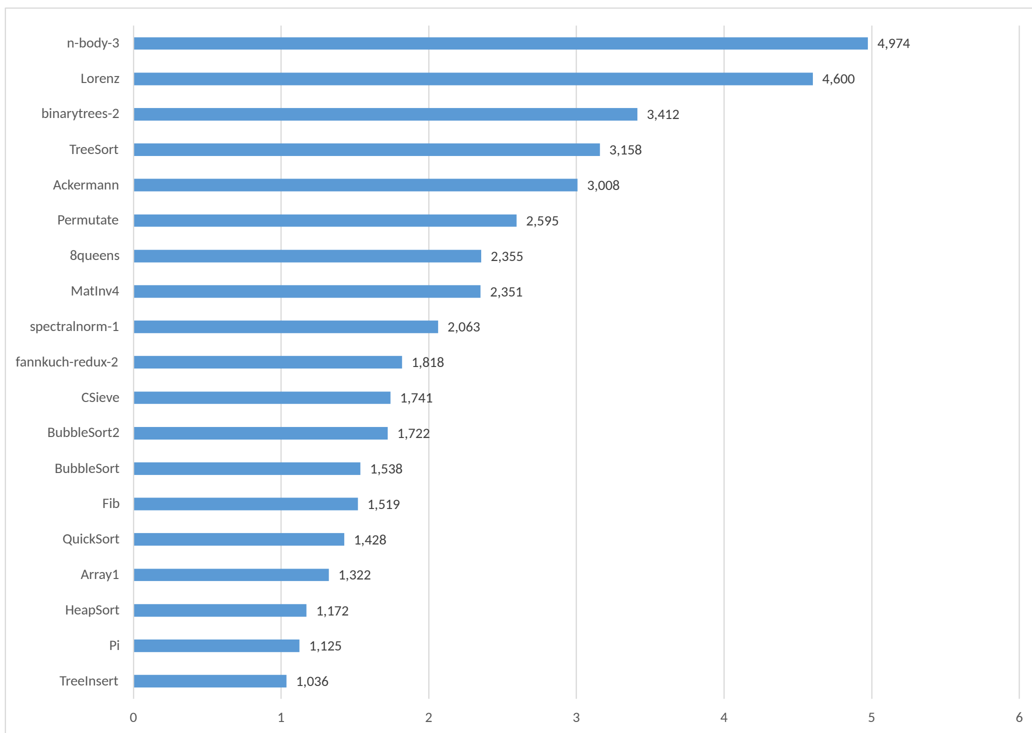

Figure 5.3 BACIL slowdown of .NET's JIT benchamrks relative to .NET.

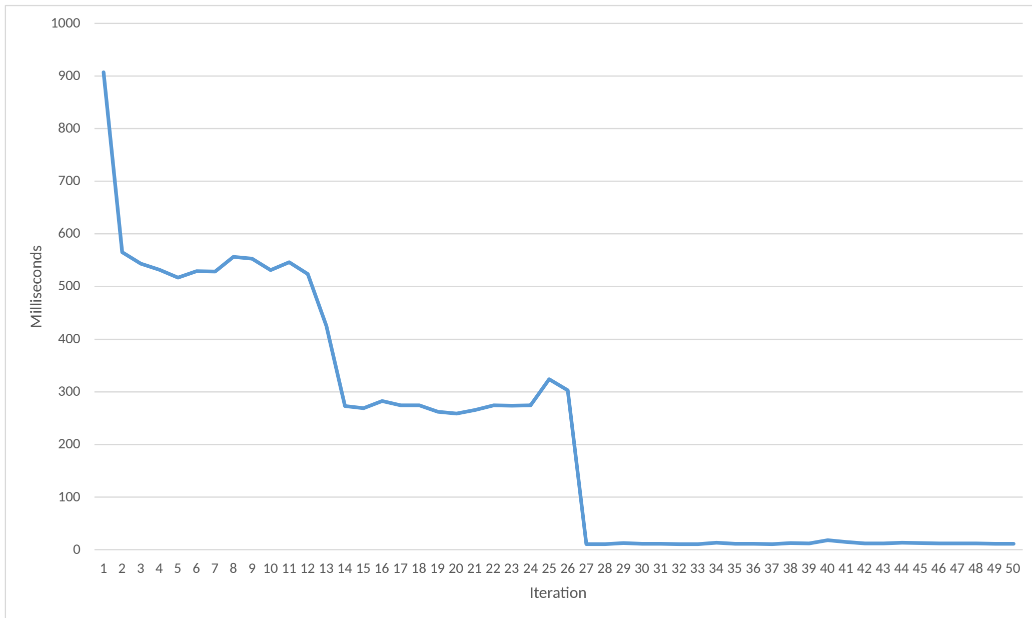

Figure 5.4 Chart showing the warmup of BACIL when running the `MatInv4` .NET runtime benchmark.

5.3.1 Warmup concerns

One fact that is important for real-world performance but our benchmarks ignore is that both the internal workings of GraalVM and our design result in the warmup time (e.g. time before reaching full performance potential is) being significant.

While the cause inherent to GraalVM is the tiered compilation model (discussed in Section 2.2), which has to compromise between the time spent compiling and the quality of the resulting compilation, BACIL has another important performance limitation. As mentioned in Section 4.1.1, for BACIL the smallest compilation unit is a method and we do not support On-Stack Replacement (OSR). Therefore, performance for the first few iterations is (expectedly) terrible.

The example code from Listing 10 runs about 60 times slower on BACIL than on .NET. See Figure 5.4 for an example chart of time-per-iteration when running the `MatInv4` .NET runtime benchmark with default Truffle heuristics. The first iteration was 83 times slower than iterations 30+.

5.3.2 Interpreting the results

We draw two main conclusions from the performance benchmarks:

- our implementation outperforms Hagmüller's work

- in compiled code, BACIL's peak performance is less than an order of magnitude slower than .NET runtime, with the worst case measured being 7.237 times slower

The last observation we want to make is regarding IL-level optimizations. While for the .NET runtime, the IL optimizations (in Release mode) made it significantly more performant, for BACIL such optimizations were very much insignificant. One interesting fact is that running Hagmüller's binarytrees on BACIL, they performed slightly worse in the optimized Release version than the Debug version. This probably has to do with the fact that the "optimizations" (which are surely tailored for .NET runtimes) resulted in using different instructions that were incidentally less performant on BACIL.

Conclusion

In this work, we set out to answer the following question: *Is it feasible to achieve the promised performance benefits [of Truffle-based implementations] with an academic interpreter-style implementation of a language runtime?* We feel the answer is *yes.*

To achieve the performance benefits, the implementation definitely needs to be designed with partial evaluation in mind, as different designs that would make no tangible difference for "classic" execution can be diametrically different when partially evaluated. This statement is supported both by our case study in Section 4.2.1 and by the difference in performance between our implementation and Hagmüller [7]. In this sense, Truffle's partial evaluation is not a "magic bullet" that will take any functioning interpreter and make it run close to state-of-the-art performance.

Once we understood the theory behind partial evaluation and about half a dozen key APIs provided by Truffle and designed the important concepts around them, the implementation stage consisted mostly of writing straightforward interpreter code. Subjectively, the interpreter style of code is easier to understand, modify and extend than traditional compilers, however, this claim can only be truly tested by other contributors.

Even though we admittedly only scratched the surface of all the services Truffle provides, our interpreter's peak performance is within an order of magnitude from .NET's state-of-the-art performance. Most benchmarks ran less than 2 times slower than .NET, showing great promise of the possible performance.

We hope that Truffle and our work can make the area of language implementation and bytecode interpreters more accessible to students, academics and hobbyists. We release all the code publicly on GitHub[7]. While we cannot promise any future code contributions into the repository from us, we want to keep maintaining the project in case there is interest from other contributors.

[7] `https://github.com/jagotu/BACIL` — this thesis reflects the repository state in commit `34a113d7c1a4bfe3e1567b28e52501c5055fb891`

5.3.3 Future work

The most obvious future work lays in the various missing features, including exceptions, generics, type checks, interfaces and others. Without fully adhering to ECMA-335[4], BACIL will always be useful only for experimentational use. Such conformance also includes all the standard libraries.

Apart from that, Truffle provides several features and intrinsics that we largely ignored and could benefit the project[8]. Examples of these features are:

- On-stack replacement (OSR)[9], which would increase the performance during warmup.
- Various instrumentation and branch prediction callbacks, which could allow increased performance and provide better profiling, debugging and stack traces.
- The Static Object model[10] which could be used to replace our custom location-based type system.
- The "polyglot" (interop) API which would allow for calling CLI-based code from other languages supported by Truffle (for example JavaScript).

[8]Some of them were unfortunately only introduced after we finalized our design.

[9]`https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/OnStackReplacement/`

[10]`https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/StaticObjectModel/`

Bibliography

[1] saelo. "Compile Your Own Type Confusions: Exploiting Logic Bugs in JavaScript JIT Engines". In: *Phrack* 16.70 (2021). URL: http://phrack.org/issues/70/9.html.

[2] Ampomah Ernest Kwame, Ezekiel Mensah Martey, and Abilimi Gilbert Chris. "Qualitative assessment of compiled, interpreted and hybrid programming languages". In: *Communications* 7.7 (2017), pp. 8–13.

[3] Thomas Würthinger et al. "Practical partial evaluation for high-performance dynamic language runtimes". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 662–676.

[4] ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 5th ed. Geneva, Switzerland, 2010. URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[5] Thomas Würthinger et al. "Self-optimizing AST interpreters". In: *Proceedings of the 8th Symposium on Dynamic Languages*. 2012, pp. 73–82.

[6] Manuel Rigger et al. "Bringing low-level languages to the JVM: Efficient execution of LLVM IR on Truffle". In: *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*. 2016, pp. 6–15.

[7] Patrick Hagmüller. "Truffle CIL Interpreter/submitted by Patrick Hagmüller, BSc." PhD thesis. Universität Linz, 2020.

[8] Yoshihiko Futamura. "Partial computation of programs". In: *RIMS Symposia on Software Science and Engineering*. Springer. 1983, pp. 1–35.

[9] Sampo Syreeni. "A brief look at file format design". In: *Hugi* 14 (1999). URL: http://decoy.iki.fi/texts/filefd/filefd.

Appendix A

Compiling and running BACIL

The full source code and a compiled version of BACIL are digital attachments to this work. You can also clone the latest version of the project from GitHub[1].

A.1 Building

For ease of building we use Maven[2], therefore building consists of running `mvn package` in the root of the project.

The resulting artifacts will be the language jar in `language/↪ target/language-1.0-SNAPSHOT.jar` and a launcher jar in `launcher/↪ target/bacil-launcher.jar`.

A.2 Running

For running you'll need to obtain GraalVM[3] and a copy of a .NET standard library[4]. While the target for development was Java 8 and .NET Runtime 5.0.6 on both Linux and Windows, we succesfully ran the project on Java 11 with .NET 6.

You'll also need an assembly to run. If you don't have any handy, you can clone BACIL_examples[5].

Once you have all the prerequisites, you can run BACIL on Windows with .NET Runtime 5.0.6 like so:

```
BACIL>java -version
openjdk version "1.8.0_282"
```

[1] `https://github.com/jagotu/BACIL/`

[2] `https://maven.apache.org/`

[3] `https://www.graalvm.org/downloads/`

[4] `https://dotnet.microsoft.com/download/dotnet`

[5] `https://github.com/jagotu/BACIL_examples`

```
OpenJDK Runtime Environment (build 1.8.0_282-b07)
OpenJDK 64-Bit Server VM GraalVM CE 21.0.0 (build 25.282-b07-
    ↪ jvmci-21.0-b06, mixed mode)

BACIL>java -Dtruffle.class.path.append=language/target/language
    ↪ -1.0-SNAPSHOT.jar -jar launcher/target/bacil-launcher.jar "
    ↪ --cil.libraryPath=c:\Program Files\dotnet\shared\Microsoft.
    ↪ NETCore.App\5.0.6" BACIL_examples\Inheritance\bin\Debug\
    ↪ net5.0\TestHarness.dll
Micka: Meow
Rex: Woof
Returned 4
Returned: 0
Runtime: 110ms
```

Apart from the last 2 lines (added by BACIL to aid debugging), the output should be identical when running the assemblies directly.

```
BACIL>dotnet BACIL_examples\Inheritance\bin\Debug\net5.0\
    ↪ TestHarness.dll
Micka: Meow
Rex: Woof
Returned 4
```

On Linux it's very similar:

```
$ java -version
openjdk version "1.8.0_292"
OpenJDK Runtime Environment (build 1.8.0_292-b09)
OpenJDK 64-Bit Server VM GraalVM CE 21.1.0 (build 25.292-b09-
    ↪ jvmci-21.1-b05, mixed mode)

$ java -Dtruffle.class.path.append=language/target/language-1.0-
    ↪ SNAPSHOT.jar -jar launcher/target/bacil-launcher.jar --cil.
    ↪ libraryPath=../dotnet-runtime-5.0.6-linux-x64/shared/
    ↪ Microsoft.NETCore.App/5.0.6 BACIL_examples/Inheritance/bin/
    ↪ Debug/net5.0/TestHarness.dll
Micka: Meow
Rex: Woof
Returned 4
Returned: 0
Runtime: 348ms

$ dotnet BACIL_examples/Inheritance/bin/Debug/net5.0/TestHarness
    ↪ .dll
Micka: Meow
Rex: Woof
Returned 4
```

Always make sure you are running GraalVM and replace the `libraryPath` with path to the .NET standard library DLLs.

A.3 Full example of running pre-compiled BACIL on amd64 Linux

Run these commands from a directory with `language-1.0-SNAPSHOT.jar` and `bacil-launcher.jar`

```
# Download and extract dotnet and graal
wget https://download.visualstudio.microsoft.com/download/pr/0
    ↪ e83f50a-0619-45e6-8f16-dc4f41d1bb16/
    ↪ e0de908b2f070ef9e7e3b6ddea9d268c/dotnet-sdk-6.0.302-linux-
    ↪ x64.tar.gz
wget https://github.com/graalvm/graalvm-ce-builds/releases/
    ↪ download/vm-22.1.0/graalvm-ce-java11-linux-amd64-22.1.0.tar
    ↪ .gz
mkdir dotnet-sdk
tar xzf dotnet-sdk-6.0.302-linux-x64.tar.gz -C dotnet-sdk
tar xzf graalvm-ce-java11-linux-amd64-22.1.0.tar.gz

# Get the example assembly
git clone https://github.com/jagotu/BACIL_examples

# Run the example assembly
graalvm-ce-java11-22.1.0/bin/java -Dtruffle.class.path.append=
    ↪ language-1.0-SNAPSHOT.jar -jar bacil-launcher.jar --cil.
    ↪ libraryPath=dotnet-sdk/shared/Microsoft.NETCore.App/6.0.7/
    ↪ BACIL_examples/Inheritance/bin/Debug/net5.0/TestHarness.dll
```

Appendix B

Opcode implementation status

Implemented instructions:

```
add
and
beq
beq.s
bge
bge.s
bge.un
bge.un.s
bgt
bgt.s
bgt.un
bgt.un.s
ble
ble.s
ble.un
ble.un.s
blt
blt.s
blt.un
blt.un.s
bne.un
bne.un.s
box
br
brfalse
brfalse.s
br.s
brtrue
brtrue.s
call
callvirt
ceq
cgt
cgt.un
clt
clt.un
conv.i
conv.i1
conv.i2
conv.i4
conv.i8
conv.r4
conv.r8
conv.u
conv.u1
conv.u2
conv.u4
conv.u8
div
dup
initobj
ldarg.0
ldarg.1
ldarg.2
ldarg.3
ldarga.s
ldarg.s
ldc.i4
ldc.i4.0
ldc.i4.1
ldc.i4.2
ldc.i4.3
ldc.i4.4
ldc.i4.5
ldc.i4.6
ldc.i4.7
ldc.i4.8
ldc.i4.m1
ldc.i4.s
ldc.i8
ldc.r4
ldc.r8
ldelem
ldelema
ldelem.i
ldelem.i1
ldelem.i2
ldelem.i4
```

```
ldelem.i8
ldelem.r4
ldelem.r8
ldelem.ref
ldelem.u1
ldelem.u2
ldelem.u4
ldfld
ldflda
ldind.i
ldind.i1
ldind.i2
ldind.i4
ldind.i8
ldind.r4
ldind.r8
ldind.ref
ldind.u1
ldind.u2
ldind.u4
ldlen
ldloc.0
ldloc.1
ldloc.2
ldloc.3
ldloca.s
ldloc.s
ldnull
ldsfld
ldsflda
ldstr
ldtoken
mul
neg
newarr
newobj
nop
not
or
pop
rem
ret
shl
shr
shr.un
starg.s
stelem
stelem.i
stelem.i1
stelem.i2
stelem.i4
stelem.i8
stelem.r4
stelem.r8
stelem.ref
stfld
stind.i
stind.i1
stind.i2
stind.i4
stind.i8
stind.r4
stind.r8
stind.ref
stloc.0
stloc.1
stloc.2
stloc.3
stloc.s
stsfld
sub
unbox.any
xor
```

Unimplemented instructions and prefixes:

```
add.ovf
add.ovf.un
arglist
break
calli
castclass
ckfinite
constrained.
conv.ovf.i
conv.ovf.i1
conv.ovf.i1.un
conv.ovf.i2
conv.ovf.i2.un
conv.ovf.i4
conv.ovf.i4.un
conv.ovf.i8
conv.ovf.i8.un
conv.ovf.i.un
conv.ovf.u
conv.ovf.u1
conv.ovf.u1.un
conv.ovf.u2
conv.ovf.u2.un
conv.ovf.u4
conv.ovf.u4.un
conv.ovf.u8
conv.ovf.u8.un
conv.ovf.u.un
conv.r.un
cpblk
cpobj
div.un
endfilter
endfinally
initblk
isinst
jmp
ldarg
ldarga
```

ldftn
ldloc
ldloca
ldobj
ldvirtftn
leave
leave.s
localloc
mkrefany
mul.ovf
mul.ovf.un
no.
readonly.
Refanytype
refanyval
rem.un
rethrow
sizeof
starg
stloc
stobj
sub.ovf
sub.ovf.un
switch
tail.
throw
unaligned.
unbox
volatile.