The Journal of Systems and Software 215 (2024) 112088

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

When the dragons defeat the knight: Basilisk an architectural pattern for platform and language independent development☆,☆☆,★

Francesco Bertolotti [a], Walter Cazzola [a,*], Dario Ostuni [b], Carlo Castoldi [a]

[a] Università degli Studi di Milano, Computer Science Department, Milan, Italy
[b] Università degli Studi di Verona, Computer Science Department, Verona, Italy

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.11058650

Keywords:
Transpilation
Programming languages

ABSTRACT

In this work, we introduce Basilisk, a high-level architectural pattern designed to facilitate interoperability among various languages, platforms, and ecosystems. The pursuit of *language-independent* software development is highly desirable, enabling developers to utilize existing software products with most programming languages. Achieving *platform independence* is equally advantageous, allowing code deployment on different platforms effortlessly. While the development community has often aimed for either language or platform independence, Basilisk aims to combine both into a single product. To realize this dual objective, Basilisk employs two fundamental components. The first is a *transpilation infrastructure* used to render software products language-independent. The second is an *abstraction layer* over platforms, enabling the creation of platform-independent software products. To illustrate Basilisk's potential, we introduce Hydra, a one-to-many, declarative transpilation infrastructure. Hydra has been utilized to develop transpilers from HydraKernel (source language) to various target languages, including D, C++, C#, Scala, Ruby, Hy, and Python. Additionally, we instantiate the abstraction layer in Wyvern, a low-level embedded domain-specific language for GPU programming, supporting any Vulkan-compatible GPU. With the Hydra transpilation infrastructure, Wyvern becomes available for D, C++, C#, Scala, Ruby, Hy, and Python. We evaluate Basilisk through the instantiation of Hydra and Wyvern, writing five algorithms from the Rodinia suite for the seven available languages, totaling 35 benchmarks. These benchmarks are executed on four different hardware platforms.

1. Introduction

Problem Statement. Programming libraries stand as pivotal elements in a language's ecosystem (Pang et al., 2018). Unfortunately, these libraries are often limited to a handful of languages, exemplified by the popular scientific computing library NumPy (Harris et al., 2020), exclusive to Python. Consequently, when selecting a development language, developers must consider library availability, creating a situation where language choice is driven more by ecosystem considerations than intrinsic language characteristics (Pang et al., 2018).

Introducing new languages to the developer community becomes a daunting task due to the years required for ecosystem maturity. Additionally, as language ecosystems evolve independently, the presence of overlapping libraries with high replication levels is common. For instance, various programming languages offer libraries for random number generation, each with its unique features and bugs. Python's `random`[1] module supports various distributions, while Java's `Random`[2] class only covers the normal distribution. Consequently, transitioning between languages involves grappling with these differences.

As the community gravitates towards newer languages, older software becomes obsolete, leading to the loss and replication of functionally useful software in newer ecosystems. Simultaneously, programming libraries should strive for independence from the execution platform, whether it is a hardware architecture (CPU/GPU/TPU), an

☆ Editor: Lingxiao Jiang.

☆☆ The title pays homage to the famous "Dragon" Book (Aho et al., 1986) where on the cover stands a knight (the programmer) who defeats a dragon (the compiler). The inside joke is that Basilisk (a dragon) can limit the need for a new transpiler to support a software product in a different language/platform demonstrated by Hydra and Wyvern (other two dragons) instantiating Basilisk.

★ This work was partly supported by the MUR, Italy project "T-LADIES" (PRIN 2020TL3X8X).

* Corresponding author.
E-mail addresses: bertolotti@di.unimi.it (F. Bertolotti), cazzola@di.unimi.it (W. Cazzola).

[1] https://docs.python.org/3.10/library/random.html

[2] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html

https://doi.org/10.1016/j.jss.2024.112088
Received 22 May 2023; Received in revised form 24 April 2024; Accepted 28 April 2024
Available online 20 May 2024
0164-1212/© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

operating system (OS), or any other software platform like a DBMS (as different DBMS could become more appropriate as both the DBMS and software product evolve Poltavtseva, 2019). For example, Python's `os`[3] library provides a mostly OS-independent API to interact with the host OS. Platform-independent software empowers user developers to create a single product running seamlessly across various platforms (Latif and Lakhrissi, 2016). This approach allows developers to interact with different platforms through a homogeneous interface, mitigating the peculiarities of each platform.

While both language and platform independence are highly desirable, achieving these properties is extremely challenging (Bishop and Horspool, 2006; Hatledal et al., 2019). Despite their relevance, software products designed to attain at least one of these properties remain scarce.

State-of-the-Art. While discussions regarding platform-independent software are often domain-specific (Draxler and Jänsch, 2004; Wang et al., 2018; Zhang et al., 2022), programming languages themselves consistently aim for platform independence. This objective is typically achieved through the use of a virtual machine or a middleware layer. Examples include GraalVM (Würthinger et al., 2013; Webb et al., 2021) and the .NET common language runtime (Box and Sells, 2002).

The ongoing conversation about language interoperability lacks a framework that effortlessly decouples software products from both the underlying platform and user-end programming languages. Existing approaches often focus solely on translating one language into another, e.g., Albrecht et al. (1980) translate Ada to Pascal, Coco et al. (2018) translate Java to Python, and Seymour and Dongarra (2003) translate Fortran to JVM bytecode. Alternatively, inter-process communication is explored in works like (Vinoski, 1997; Slee et al., 2007), but this method introduces data marshaling between cross-language calls, limiting performance (Grimmer et al., 2018). Multi-language run-times such as (Box and Sells, 2002; Grimmer et al., 2018) provide environments to allow for a high degree of interoperability between different languages using the concept of virtual machines. However, this approach while effective is often constrained by the respective virtual machines. For example, the JVM provides interoperability only between those programming languages running on the JVM as Java, Kotlin, and Scala.

A more comprehensive approach to the problem is demonstrated by HaXe (Ponticelli and McColl-Sylveste, 2008), capable of performing source-to-source translation to multiple languages. However, it lacks a mechanism for easily extending the pool of supported target languages.

Proposed Solution. In this work, we introduce Basilisk, a high-level architectural pattern designed to address the challenge of creating language- and platform-independent software. To contextualize the problem, we focus on two primary interactions of a software product: (i) interaction with a user and (ii) interaction with the platform. Our focus is on software products that engage through programming languages. Such products expose two interfaces: (i) the user-to-product interface, and (ii) the product-to-platform interface.

The user-to-product interface enables users to interact with the software product, and the product-to-platform interface is used by the software product to communicate with platforms. Notably, the product-to-platform interface serves as an abstraction layer, effectively decoupling the software product from the underlying platform. This abstraction can be implemented using established software design patterns like the adapter pattern (Gamma et al., 1995) or other formalisms such as (Selic, 2005).

Instead, the user-to-product interface is crucial for user communication. To avoid tying the software system to a specific programming language, the user-to-product interface is translated into various languages using a transpiler. This transpiler, reusable across different scenarios, underscores the necessity for a source-to-source transpilation infrastructure such as (Quinlan and Liao, 2011; Keep and Dybvig, 2013; Bertolotti et al., 2024).

Implementations. To illustrate the potentiality of the Basilisk architectural pattern, we instantiate its components into three distinct entities: Hydra, Wyvern, and MinPy.

Hydra is a source-to-source, one-to-many, and extensible transpilation infrastructure implemented in Scala. Utilizing a declarative approach, Hydra simplifies the definition of transpilations from HydraKernel to various programming languages. We demonstrate Hydra's capabilities by developing seven transpilers from HydraKernel to D, C++, C#, Scala, Ruby, and Hy, as well as Python.

Wyvern is a simple *embedded domain specific language* (EDSL) (Mernik et al., 2005; Dubochet, 2011) for GPGPU programming. Or, in other words, Wyvern is a domain specific language (Kosar et al., 2008, 2016) accessible from a library API for the GPGPU programming application domain. Wyvern considers different GPUs and CPUs as distinct platforms for executing Wyvern programs. Consequently, Wyvern acts as an abstraction layer for low-level GPU programming with CPU support. Notably, Wyvern is natively available only for Rust, limiting its usability to Rust programmers.

To further illustrate the pattern, we developed **MinPy**, a small array computing library. MinPy can be executed on two different back-ends: NumPy and Torch, representing different platforms.

Research Questions. To understand whether the Basilisk architecture enables the desired language- and platform-independence, we will try to answer the following research questions:

RQ_1. Is the Basilisk architectural pattern successful in rendering a software product language-independent?

The answer should be yes only if the software product can be extended to support new languages via the Basilisk pattern.

RQ_2. Is the Basilisk architectural pattern successful in rendering a software product platform-independent?

The answer should be yes only if the software product can be extended to support new platforms via the Basilisk pattern.

RQ_3. What is the cost of introducing new languages and/or platforms?

Answered in terms of lines of code and number of files required. As Basilisk is a pattern build around the software product, its code impact should be minimal.

Contributions. Our contribution is summarized in the following three points:

- the Basilisk architectural pattern for the development of language- and platform-independent software, and
- the instantiation of the Basilisk architectural pattern to address issues in the GPGPU programming domain,
- an evaluation of the Basilisk architectural pattern through the answering of the research questions.

The remaining of this work is organized as follows. Section 2 presents the overall Basilisk architecture. Section 3 instantiates one of the Basilisk components—the transpilation infrastructure. Section 4 instantiates the abstraction layer for a small array computing library—MinPy. Section 5 instantiates the abstraction layer for a GPGPU EDSL—Wyvern. We evaluate the resulting instantiations of the Basilisk (Hydra+MinPy and Hydra+Wyvern) pattern in Section 6. Results and research questions are discussed in Section 7. Section 9 discusses related works in the field. Finally, in Section 10 we draw our conclusions.

[3] https://docs.python.org/3/library/os.html

2. Basilisk architectural pattern

Basilisk is an architectural pattern to ease the development of language- and platform-independent software products. It achieves this by enabling the use of the development environment from other programming languages and abstracting away from specific platforms. Where the language represents how users interact with the software product, and the platform represents the way the software product interacts with the ecosystems. Software products that are language- and platform-independent:

- **broader user base**: they attract a larger user base compared to competitors by accommodating the language and platform preferences/constraints of more users;
- **survivability in technological shifts**: these products can endure technological shifts, where older languages and platforms are replaced by newer ones;
- **enhanced stability**: the larger user base provides a substantial pool of testers, contributing to the product's stability;
- **automated interface alignment**: these products eliminate the need to manually align exposed interfaces across implementations, as any changes automatically propagate.

Given these advantages, language- and platform-independent software products become compelling. Nonetheless, their development presents many challenges that require a high-level degree of decoupling and extensibility.

2.1. Architectural pattern overview

A software product encompasses applications or libraries designed for language- and platform-independence. It exposes its functionality through the user-to-product interface, accessible to users—entities that can be either human or other software entities. The software product's capabilities are built upon the functionality provided by a platform, which could be an operating system, hardware architecture, or another software product. Access to necessary functionalities occurs through the product-to-platform interface, implemented by various back-ends. Each back-end connects platform-specific functionalities to those required by the product-to-platform interface. Together, these interfaces and different back-ends form the abstraction layer. The user-to-product interface encapsulates the software product's capabilities. Availability in multiple programming languages is ensured through a transpilation infrastructure.

Fig. 1 illustrates the Basilisk architectural pattern and its components. Users access an interface tailored to their preferred language, generated from the user-to-product interface. The software product implements the user-to-product interface and relies on the product-to-platform interface. This interface is implemented by a back-end designed for a specific platform. From the user's perspective, interacting with the software product in their preferred language is seamless, and they are not required to manually create bindings. Simultaneously, the software product transparently accesses the underlying platform. It is worth noting that while user-side interfaces can be manually crafted, this process is mechanical and prone to errors. Additionally, changes in the user-to-product interface must be propagated to all user-side interfaces, a task that can be challenging to manage manually. Leveraging a transpilation infrastructure automates the availability of the user-to-product interface in multiple languages and ensures seamless propagation of changes. However, changes in the product-to-platform interfaces necessitate manual handling due to the diverse behaviors of different platforms. Each change may require a unique approach depending on the specific platform involved.

2.2. Abstraction layer

The abstraction layer serves the crucial role of decoupling the software product from the platforms, each offering its unique set of

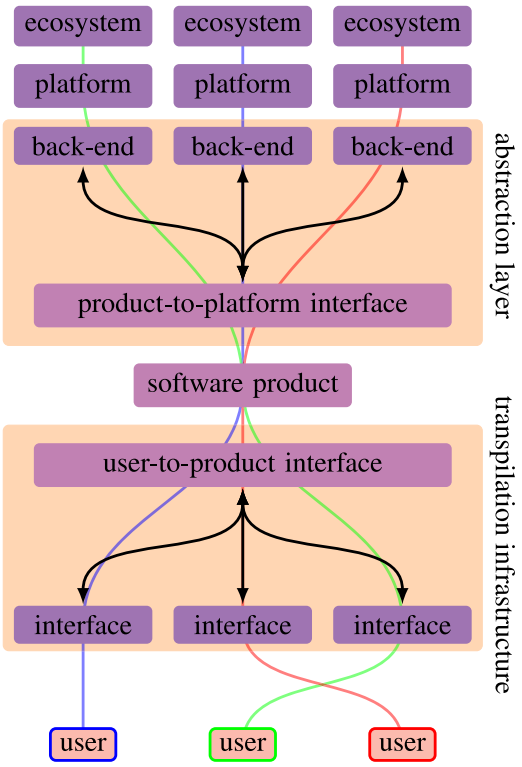


Fig. 1. The Basilisk architecture involves users choosing their preferred language interface. All language interfaces are consistently generated from the user-to-product interface through the language infrastructure. The software product provides the actual functionalities, which may require functionalities from different platforms. These functionalities are declared by the product-to-platform interface, implemented by a back-end. Each back-end utilizes platform-dependent functionalities, effectively decoupling the software product from the language interfaces and the platforms.

functionalities. Platforms may differ significantly in the ways they provide essential functionalities, requiring the software product to adapt accordingly. For instance, a logging library may need to interact with the underlying filesystem for read and/or write operations, but the implementation details vary across different operating systems. To bridge these differences, an abstraction layer is introduced over the available platforms, consolidating variability into a unified interface. The software product exclusively interacts with platforms through the product-to-platform interface, which, in turn, relies on a back-end to access platform-specific functionality.

Consider the example of a logging library with a product-to-platform interface containing abstract methods like read and write. Each back-end then implements these operations for the corresponding platform (e.g., operating system), ensuring the software product accesses the appropriate implementation when using read or write operations.

In cases where an existing abstraction layer, such as Python's `pathlib`, is available, it can be leveraged instead of developing a new one. However, a product-to-platform interface and the required back-ends must be developed when no abstraction layer exists. For effective abstraction layer development, it is advisable to create small product-to-platform interfaces to reduce back-end size, avoid overlap with other back-ends for the same platform, and facilitate composition with functionality requested by other product-to-platform interfaces.

The key points of the abstraction layer development are:

1. to build the product-to-platform interface, i.e., to identify a set of minimal and necessary operations to build the software product;
2. to implement the back-ends implementing the product-to-platform interface for platforms to be supported;

3. to develop (or to adapt) the software product accessing platform functionality only via the product-to-platform interface;
4. to maintain, to integrate with other platforms, and to add new functionality.

To summarize, the abstraction layer represents the interface through which the software product interacts with the existing ecosystem.

2.3. Transpilation infrastructure

The transpilation infrastructure plays a pivotal role in decoupling the software product from the programming language used to access it. While the software product is primarily developed in a specific programming language to offer a set of functionalities, these functionalities are inherently tied to that language. To ensure widespread accessibility, these functionalities should be available to users in various programming languages.

Consider a logging library, initially designed for Java users, but now sought after by Python users. By creating a user-to-product interface and transpiling it into different languages, we can provide the same interface for diverse ecosystems. This ensures that users from different programming languages can seamlessly access the same software product.

It is important to note that with a full language-to-language transpiler, it is possible to transpile the entire stack, including back-ends, the product-to-platform interface, the software product, and the user-to-product interface. However, such transpilers are complex and challenging tools to develop. On the other hand, if the user-to-product interface is designed as bindings to access the software product, the translation process simplifies to transpiling these bindings. This approach significantly eases the development of transpilers, as bindings can be written using a simple, bare-bones language.

Moreover, a transpiler is agnostic to the application domain in which it is deployed, making it highly reusable. For instance, a Java-to-Python transpiler developed for a Java logging library can be applied not only to translate this library into Python but also to other Java Basilisk-compliant software products, showcasing its versatility across different scenarios.

Basilisk. To summarize, using an abstraction layer allows us to make the software product platform-independent. Simultaneously, language-independent software products can be achieved through transpilation, necessitating an infrastructure for building transpilers. Together, these elements enable the development of software that satisfies both properties. It is also worth noting the intrinsic recursiveness of the Basilisk framework. The abstraction layer in the Basilisk framework could itself be another Basilisk-compliant software product, using as its abstraction layer yet another Basilisk-compliant software product, and so on. More precisely, the product-to-platform interface used by a software product may align with the transpilation of the user-to-product interface of another software product. Therefore, Basilisk-compliant software products can be stacked on each other, achieving a higher level of functionality compared to the product on which they are built.

3. The Transpilation Infrastructure: Hydra

Hydra is an extensible, one-to-many, source-to-source, transpilation infrastructure. The transpilation infrastructure's main purpose is to ease the development of transpilers for the user-to-product interface. Source-to-source as the translations are performed from one source file to another. One-to-many as new target languages are always defined starting from an existing source language. Extensible as new target languages can be supported through a declarative approach.

A Hydra transpiler is composed of three components: HydraKernel, Hydra templates, and Hydra plugins. The HydraKernel is the source language used to implement the user-to-product interface. The HydraKernel language features can be extended or changed by adding or removing Hydra plugins. Hydra plugins handle the transpilation of the declared language features. Hydra templates fill the language-dependent knowledge needed by the plugin to perform the translation. For example, a Hydra plugin may declare the `while` language feature so that a HydraKernel source can use the while loop. The Hydra template extending the `while` Hydra plugin needs to declare the `while` syntax of the target language, so that, the plugin can perform the translation. Overall, we develop Hydra plugins necessary to have a simple yet Turing-complete HydraKernel. Next, we develop Hydra templates for D, C++, C#, Scala, Ruby, Hy, and Python languages.

HydraKernel. The HydraKernel, inspired by the concept of RPython (Ancona et al., 2007), represents a restricted subset of the Scala language and serves as the source language for Hydra (as depicted in Fig. 3). It is intentionally designed to incorporate only those language features that are commonly found across a broad range of programming languages. This intentional restriction aims to simplify the translation process for HydraKernel programs. In the current implementation, it supports:

- a few types: `int`, `float`, `boolean`, `unit` (i.e., Scala `void` type), `string`, and any pointer to the previous types;
- a minimum set of operations: *addition*, *subtraction*, *multiplication*, *power*, and *division* on `int` and `float` types; logical `and`, `or`, and `not` on `boolean` types;
- a few type conversions routines: `int-to-string`, `int-to-float`, `float-to-string`;
- string operations: *concatenation*, *length*, and *equality*;
- control-flow statements: `if`, `if-else`, and `while-do`; finally
- a basic object orientation abstraction—classes.

These components represent a small set of language features that ensures Turing completeness while providing a convenient programming language.

Code conditioning directives. Additionally, HydraKernel allows code-conditioning directives (using C-style comments). These directives are analogous to C directives and allow for conditional code compilation. For example, consider the following snippet using the code conditioning directives:

```
/* IF STATIC_ARITH */
def add(op1: T, op2: T): T = return new T(add_(op1,op2))
def sub(op1: T, op2: T): T = return new T(sub_(op1,op2))
def mul(op1: T, op2: T): T = return new T(mul_(op1,op2))
def div(op1: T, op2: T): T = return new T(div_(op1,op2))
/* ELSE */
def add(op: T): T = return new T(add_(T.this.value,op))
def sub(op: T): T = return new T(sub_(T.this.value,op))
def mul(op: T): T = return new T(mul_(T.this.value,op))
def div(op: T): T = return new T(div_(T.this.value,op))
/* ENDIF */
```

In this context, if a language supports operator overloading (e.g., Python), the code-generation process will execute the first branch. Conversely, in languages that do not support operator overloading (e.g., C or Java), the second branch is executed during code generation.

The predefined plugin `HydraOperatorOverloading` defines several code conditioning directives to enable the exploiting of operator overloading language capability. Some of these directives are: `STATIC_ARITH` for arithmetic (e.g., addition and subtraction), `STATIC_BITOP` for bitwise, `STATIC_UNARY` for unary, and `STATIC_LOAD` and `STATIC_STORE` indexing operators.

The Hydra template informs the plugin about language-dependent information. For instance, the Python Hydra template will declare support for `STATIC_ARITH`. It will also specify how the overloading of arithmetic operators is defined in Python, such as by implementing the method `__add__`, as shown by the following snippet:

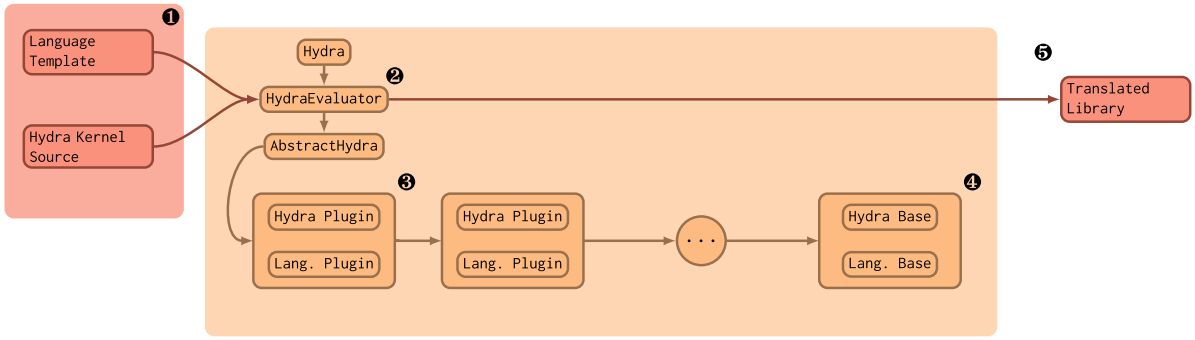


Fig. 2. Hydra architecture overview. ❶ represents the hydra inputs: a language template and a HydraKernel source. The Hydra Kernel source represents the developed user-to-product interface. The Hydra template is a Scala source file filling information required by Hydra Plugins in other to perform the transpilation in a specific target language. ❸-❹ is a customizable chain of plugins. Each plugin represents a small transpilers for a set of language features of the HydraKernel language. Language language-dependent knowledge required by the plugins can be filled from the Hydra templates. ❷ is responsible for parsing and translating the application configuration following the language configuration. ❺ represents the translated source.

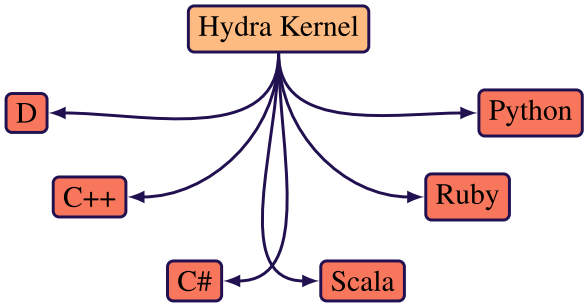


Fig. 3. HydraKernel (source language) can be translated into several other target languages (D, C++, C#, Scala, Ruby, Python).

```
// ...
override val has_static_arith_operator_overloading = true
override def add_constants(
  class_name  : String,
  params      : String,
  return_type : String,
  body        : String) : String =
    s"def __add__($params):\n${indent(body)}\n"
// ...
```

In general, code conditioning directives are treated as simple preprocessing instructions. These support only basic expressions and `if-else` statements. After the first parsing step, if-else code conditioning directives are evaluated. The result of the evaluation depends on the target language definition (declared in a Hydra template). Next, the portion of HydraKernel code irrelevant to the target language is simply dropped. In the previous example, we would maintain only the first 4 functions when transpiling HydraKernel towards C. Instead and the last 4 functions when transpiling HydraKernel towards Python.

Hydra Plugin. Hydra plugins are the extension points of Hydra. Each Hydra plugin is responsible for identifying and translating a specific language feature. For example, `HydraBase` (see Fig. 2-❹) is responsible for translating and identifying the basic imperative programming constructs, such as assignments and control statements. Another plugin, `HydraNativeCalls` is responsible for generating foreign function interfaces (FFIs). Each Hydra plugin is composed of two separate modules: a Hydra module, and a language module. Given an abstract syntax tree (AST) as input, a Hydra module identifies the AST nodes of interest for the plugin and the language module translates the identified AST nodes according to a Hydra template. For example, the `HydraNativeCalls` Hydra module identifies all classes extending `com.sun.jna.Library`. Whereas, the `HydraNativeCalls` language module translates all the class methods as if they were foreign functions. Instead, the Hydra module of `HydraBase` identifies `if` and `while` AST nodes. Whereas, the language module of `HydraBase` translates such nodes by using the information provided by the Hydra templates.

Hydra Template. A Hydra template is a Scala class representing a target language. A Hydra template declares which language features are available and, if necessary their translation pattern. Features are defined in a declarative way, so that, introducing new target languages into the system requires minimal effort. Moreover, target languages can be reused through inheritance. Thus reducing the required development effort. Language features are dictated by Hydra plugins. While Hydra plugins expose a general interface for a given language feature. Hydra templates are responsible for specializing features in their languages. For example, the following snippet declares the translation pattern for the `while-do` and `if` statements for the Python language. It implements the `HydraBase` plugin (Fig. 2-❹) which exposes the method `while_do(cond, body)` and `if_def(cond, body)`.

```
def while_do(cond: String, body: String) :
  String = s"while $cond:\n${indent(body)}\n"
def if_def(cond: String, body: String) :
  String = s"if $cond:\n${indent(body)}\n"
```

A Hydra template should be mostly independent of the application. However, through inheritance, Hydra templates can be easily customized to adapt to specific situations.

Hydra Evaluator. Consider Fig. 2. The code generation is performed by setting the plugin chain (Fig. 2-❸) and by feeding Hydra with a language template and a HydraKernel source (Fig. 2-❶). The Hydra evaluator module reads, parses, and translates the HydraKernel source following the given Hydra template (Fig. 2-❷). The translation is AST-driven, node-by-node, and delegated to one of the plugins (Fig. 2-❸-❹). Each Hydra plugin is supposed to implement one or more language features. By stacking Hydra plugins together, one may add functionality to HydraKernel. For example, `HydraNativeCalls` add an FFI extension. Each template implementing `HydraNativeCalls` will extend the capabilities of the translation to support FFI. Once the AST is translated, it is transcribed into an output file representing the translation. E.g., the following HydraKernel snippet implementing the bubble sort:

```
var i: Int = 0
var j: Int = 0
while (i < array.size) {
  while (j < array.size) {
    if(array(j) > array(j+1)) {
      swap(array,i,j)
    }
  }
}
```

is translated into the Python snippet:

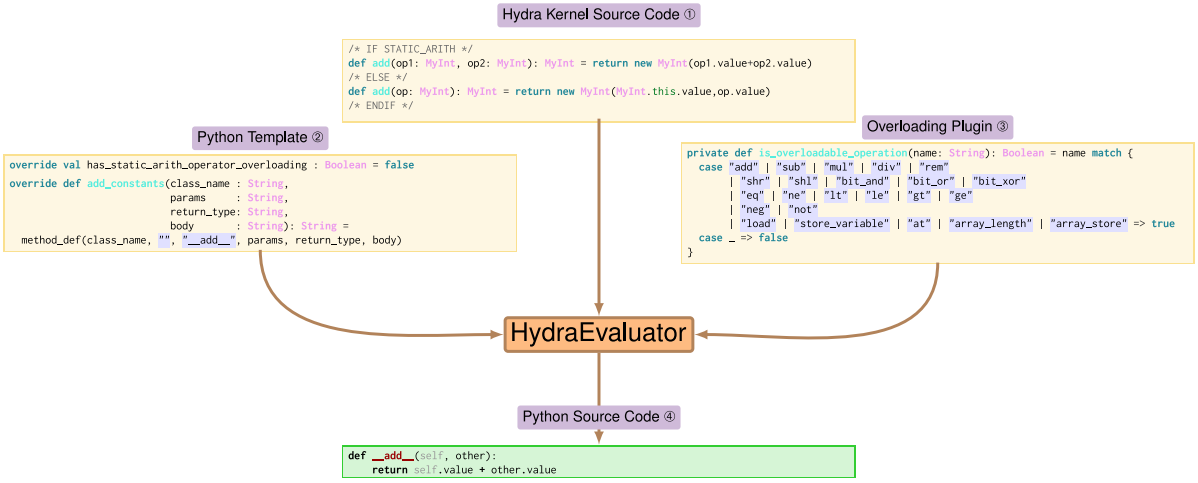


Fig. 4. Code conditioning example. A HydraKernel source is fed to the Hydra evaluator. The Python templates override from a Hydra plugin to declare the type of overloading. Thus, the code conditioning is resolved and the output is a Python snippet.

```
i = 0
j = 0
while i < array.__len__():
  while j < array.__len__():
    if array[j] > array[j + 1]:
      swap(array, i, j)
```

and into the C++ snippet:

```
int i = 0;
int j = 0;
while (i < array.size()) {
  while(j < array.size()) {
    if(array[j] > array[j + 1]) {
      swap(array, i, j)
    }
  }
}
```

Example. Consider Fig. 4. On top, a HydraKernel snippet is shown. This snippet declares the `add` operator (Fig. 4-①) for both C++ style operator overloading and Python style operator overloading. Again, the distinction is rendered explicit through code conditioning directives (in the form of comments). Meanwhile, a Hydra plugin declares that methods named `add` are overloadable (Fig. 4-③). Alongside the Hydra plugin, the Hydra template for the Python language is shown (Fig. 4-②). This Hydra template first declares that Python uses a static operator overloading approach. Next, it declares the translation pattern which is the same as the standard Python method definition, with the name `__add__`. Finally, the Hydra evaluator is responsible for using the information provided by Hydra templates, Hydra plugins, and source HydraKernel. Which results in the translation shown in Fig. 4-④.

4. A First Example: MinPy

For the sake of explanation, consider the NumPy library (Harris et al., 2020), a popular array computing library. Despite its popularity, NumPy is only available to Python users. However, all developers could potentially benefit from a NumPy-like library available in their language of choice. In this section, we instantiate the Basilisk architectural pattern for a small subset of NumPy, which we will refer to as MinPy. The same approach could be applied to the entirety of NumPy but this would not add much to the discussion.

Language Interfaces. With MinPy, we aim to offer a consistent array-based library for D, C++, C#, Scala, Ruby, Hy, and Python programming languages. Of course, the interface needs to be adapted to the specific language capabilities. For example, whenever possible, array addition should be defined with the appropriate overloading operator.

Platforms. In this example, the different platforms are represented by different array-computing libraries. For example, one platform is represented by the NumPy C API itself which we reuse to implement the most common array operations. Another platform is represented by the torch library which is accessed through the Python C API.

The product-to-platform interface. In this case, the product-to-platform interface coincides with a battery of functions to handle array generation and several array operations.

The user-to-product interface. The user-to-product interface provided by MinPy draws heavy inspiration from NumPy itself. It introduces a `NpyArray` class, which manages arrays (creation, destruction, and operators), and a NumPy class to handle array methods (e.g., `zeros`, `ones`, `arange`, and `linspace`). To gain a better understanding of the user-to-product interface, consider the following code snippet that implements a Mandelbrot set using MinPy with its Python interface:

```
size = 1024
x = mnp.linspace(-2, 1, size, npytype=mnp.FLOAT64)
    .reshape((1, size))
    .repeat(size, 0)
y = mnp.linspace(-1, 1, size, npytype=mnp.FLOAT64)
    .reshape((size, 1))
    .repeat(size, 1)
c = x+mnp.complex_(0,1)*y
z = mnp.zeros((size, size), npytype=mnp.COMPLEX128)
m = mnp.ones((size, size), npytype=mnp.BOOLEAN)
for i in range(100):
  z[m] = z[m] * z[m] + c[m]
  m[z.abs() > mnp.float_(2.0)] = mnp.bool_(False)
```

In this example, functions like `linspace`, `repeat`, `zeros`, and `ones` exhibit a behavior similar to the one of the NumPy Python API. Additionally, methods such as `complex_`, `float_`, and `bool_` perform an explicit cast from the input to the corresponding array type. This ensures that operations maintain consistent types.

The user-to-product interface is defined using a small subset of Scala—HydraKernel. Here, we write the library as we would write any other library, then by means of transpilation, we obtain the same interface written for different programming languages. For example, consider the following HydraKernel snippet:

```
trait Minpy extends com.sun.jna.Library {
  // ...
  def arange(
    start  : NativeFloat,
    stop   : NativeFloat,
    step   : NativeFloat,
    npytype: Int): VoidPointer
  // ...
}
```

Here, we are simply defining one extern function—`arange`—that will be accessed through one of the available back-ends. These functions will be rendered accessible through the NumPy class, as shown in the following code snippet:

```
object Numpy {
  // ...
  def arange(start:Float, stop:Float, step:Float,
    npytype:Int): NpyArray = {
    return new NpyArray(
      Minpy.INSTANCE.c_arange(
        start.toNativeFloat,
        stop.toNativeFloat,
        step.toNativeFloat,
        npytype.toNativeInt)
    )
  }
  // ...
}
```

The HydraKernel library will also need to handle special cases such as operator overloading. For example, as mentioned earlier, one language may support operator overloading (such as Python), while another may not support it (such as C). If we wish to support the indexing operator for MinPy arrays whenever possible, we need to provide two different implementations:

```
class NpyArray(npyarray:VoidPointer) {
  //...
  /* IF STATIC_LOAD_ARRAY */
  def at(arr: NpyArray, idx: NpyArray): NpyArray = {
    return NpyArray(getitem(arr, idx.pointer))
  }
  /* ELSE */
  def at(idx: NpyArray): NpyArray = {
    return NpyArray(getitem(this.pointer, idx.pointer))
  }
  /* ENDIF */
  //...
}
```

The first implementation supports the indexing overloading, while the second implementation is used whenever the indexing overloading is disabled. As a result, when generating the interface for different languages we obtain different generation behaviors. For example, when generating the Python interface we obtain:

```
class NpyArray:
  # ...
  def __getitem__(self,index):
    return NpyArray(getitem(self.pointer, index.pointer))
  # ...
```

Meanwhile, when generating the Scala interface, that does not support the indexing operator overloading, we obtain:

```
class NpyArray(npyarray: Long) {
  // ...
  def loc(arr:NpyArray, idx:NpyArray): NpyArray = {
    return new NpyArray(getitem(arr.pointer, key.pointer))
  }
  // ...
}
```

The translation of the `at` method for different languages is specified in the Hydra template corresponding to each language. For instance, the Python template outlines the following translation:

```
class Python3Lang() extends ObjectOrientedLang {
  // ...
  override def at(
    clsname : String,
    params  : String,
    rettype : String,
    body    : String) : String = {
      methodDef(clsname, "__getitem__", params, rettype, body)
  }
  // ...
}
```

This means that the `at` method for Python should be translated into a method definition where the name is changed to `__getitem__`, meanwhile, the rest of the method remains unchanged. Similarly, the Scala template does simply rename the method `at`, however, the chosen method name has no special meaning wrt. the Python `__getitem__`, as shown in the following snippet:

```
class ScalaLang() extends ObjectOrientedLang {
  // ...
  override def at(
    clsname : String,
    params  : String,
    rettype : String,
    body    : String) : String = {
      methodDef(clsname, "loc", params, rettype, body)
  }
  // ...
}
```

Note that, the Hydra template is not directly responsible for the translation of HydraKernel `at` method into the respective Scala (or Python) code. The actual translation is handled by the `HydraOperatorOverloading` plugin. By overloading the `at` method the various templates can customize the code generation of the plugin depending on their specific needs.

The abstraction layer. The abstraction layer aims to render the usage of one platform transparent compared to another. With MinPy, our objective is to achieve transparency in the usage of one array-computing library compared to another. We have implemented back-ends for accessing the NumPy C API platform and the back-end for accessing the Torch API. Since both back-ends satisfy the same interface, changing the back-end allows reliance on one platform or the other. In this case, the back-ends are encapsulated in different shared objects one using NumPy and one using Torch. The back-ends can be swapped by swapping one shared object with another. However, it is essential to note that there is not a significant benefit in choosing the NumPy API over the Torch API in this scenario, but we consider such a case for the sake of the explanation.

The result is that the same code can be executed with different platforms and, with minimal differences, can also be executed in different programming languages. For example, consider the Scala implementation of the previously presented Mandelbrot set:

```
val size : Int = 1024
var x = Numpy.linspace(-2, 1, size, Numpy.FLOAT64)
  .reshape(Array(1, size))
  .repeat(size,0)
var y = Numpy.linspace(-1, 1, size, Numpy.FLOAT64)
  .reshape(Array(size, 1))
  .repeat(size,1)
var c = x+Numpy.complex_(0,1)*y
var z = Numpy.zeros(Array(size, size), Numpy.COMPLEX128)
var m = Numpy.ones(Array(size, size), Numpy.BOOLEAN)
for(i <- 1 to 100) {
   z.put(m, z.loc(m) * z.loc(m) + c.loc(m))
   m.put(z.abs() > Numpy.float_(2.0f), Numpy.bool_(false))
}
```

Both Python and Scala implementation share the same structure with an almost identical interface. The few noticeable differences pertain to the distinctions between the Python and Scala languages. For example, Scala does not support the overloading of the indexing operator; therefore, array indexing is done through the methods `loc` and `put`.

```python
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
from array import array
import numpy as np

program = SourceModule("
typedef unsigned u32;
__global__ void add(u32 n, const float* a,
                const float* b, float* c) {
   u32 tid = blockDim.x * blockIdx.x + threadIdx.x;
   u32 tsize = blockDim.x * gridDim.x;
   for(; tid < n; tid += tsize) {
      c[tid] = a[tid] + b[tid];
   }
}
")

N = 4
host_a = array('i', [-(i+10**7) for i in range(N)])
host_b = array('i', [ (i+10**7) for i in range(N)])
host_c = array('i', [0 for _ in range(N)])
dev_a = cuda.mem_alloc(4 * N)
dev_b = cuda.mem_alloc(4 * N)
dev_c = cuda.mem_alloc(4 * N)
cuda.memcpy_htod(device_a, host_a)
cuda.memcpy_htod(device_b, host_b)
add = program.get_function("add")
N = np.uint32(N)
add(N, dev_a, dev_b, dev_c, block=(256, 1, 1))
cuda.memcpy_dtoh(host_c, dev_c)
print(" ".join(map(str, host_c.tolist())))
```

(a) A Python program using PyCUDA to add two vectors. The variable `program` declares the device code to be run. The rest of the program handles data initialization and preparation for the execution of the device code.

```
$ ./add.cuda.py
-10000000 -10000001 -10000002 -10000003
```

(b) Result of the execution of listing 1(a).

```c
const char* source =

"__kernel void add(const uint n,         "
"                  __global const float* a,"
"                  __global const float* b,"
"                  __global float* c) {    "
"   size_t tid = get_global_id(0);       "
"   size_t tsize = get_global_size(0);   "
"   for(; tid < n; tid += tsize)         "
"      c[tid] = a[tid] + b[tid];         "
"}                                       ";

int main(int argc, char* argv[]) {

   const unsigned N = 4;
   int* host_a = calloc(N, sizeof(int));
   int* host_b = calloc(N, sizeof(int));
   int* host_c = calloc(N, sizeof(int));
   for(int i = 0; i < N; i++) {
      host_a[i] = -(i + 10000000);
      host_b[i] =  (i + 10000000);
   }
   cl_mem dev_a = clCreateBuffer(/* arguments */);
   cl_mem dev_b = clCreateBuffer(/* arguments */);
   cl_mem dev_c = clCreateBuffer(/* arguments */);

   // Copy memory from host_{a,b,c} to dev_{a,b,c}
   clEnqueueNDRangeKernel(/* call add */);

   // Copy memory from dev_{a,b,c} to host_{a,b,c}
   for(int i = 0; i < N; i++) printf("%d ", host_c[i]);
   printf("\n");
   return 0;
}
```

(c) A C program using OpenCL to add two vectors. The variable `source` declares the device code to be run. The rest of the program handles data initialization and preparation for the execution of the device code.

```
$ gcc add.opencl.c -lOpenCL -o add.opencl.bin && ./opencl.bin
-10000000 -10000001 -10000002 -10000003
```

(d) Result of the compilation of listing 1(c).

Listing 1 Both PyCUDA and OpenCL fail to detect type mismatch between host and device code.

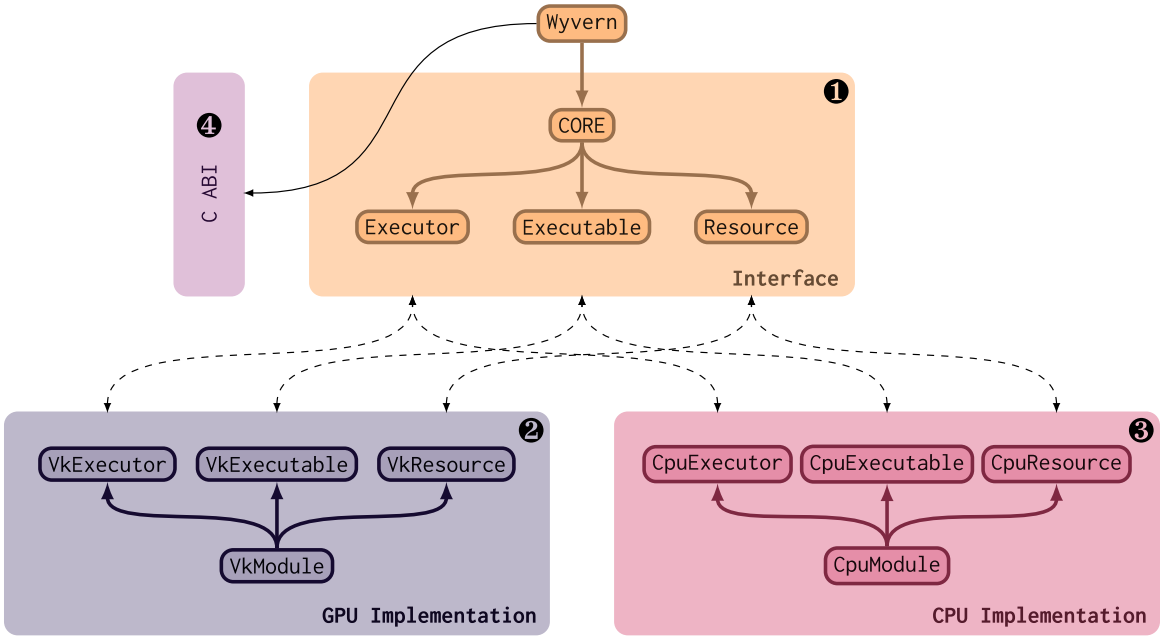


Fig. 5. Wyvern architecture overview. The four main modules of Wyvern: CORE module (❶), Vk module (❷), CPU module (❸) and the C ABI module (❹).

```
1  typedef unsigned u32;
3  __global__ void add(u32 n, const float* a,
4        const float* b, float* c) {
5    u32 tid = blockDim.x * blockIdx.x + threadIdx.x;
6    u32 tsize = blockDim.x * gridDim.x;
7    for(; tid < n; tid += tsize)
8      c[tid] = a[tid] + b[tid];
9  }
11 __global__ void add(u32 n, const int* a
12       const int* b, int* c) {
13   u32 tid = blockDim.x * blockIdx.x + threadIdx.x;
14   u32 tsize = blockDim.x * gridDim.x;
15   for(; tid < n; tid += tsize)
16     c[tid] = a[tid] + b[tid];
17 }
19 int main(int argc, char* argv[]) {
20   const u32 N = 4;
21   // allocate host memory
22   int* host_a = (int*)calloc(N, sizeof(int));
23   int* host_b = (int*)calloc(N, sizeof(int));
24   int* host_c = (int*)calloc(N, sizeof(int));
25   // allocate device memory
26   int* dev_a,* dev_b,* dev_c;
27   cudaMalloc(&dev_a, N * sizeof(int));
28   cudaMalloc(&dev_b, N * sizeof(int));
29   cudaMalloc(&dev_c, N * sizeof(int));
30   // initialize host memory
31   for(int i = 0; i < N; i++) {
32     host_a[i] = -(i + 10000000);
33     host_b[i] =  (i + 10000000);
34   }
35   // mem copy from host_{a,b,c} to dev_{a,b,c}
36   add<<<(N/256)+1, 256>>>(N, dev_a, dev_b, dev_c);
37   // mem copy from dev_{a,b,c} to host_{a,b,c}
38   for(int i = 0; i < N; i++)
39     printf("%d ", host_c[i]);
40   printf("\n");
41   // free memory
42   return 0;
43 }
```

(a) A CUDA code for summing two vectors. This program presents two versions of the same device code with only one small difference. The top kernel function declares input parameters as `float*`. The second one declares input parameters as `int*`.

```
$ nvcc add.cuda.cu -o add.cuda.bin
... "int *" is incompatible with ... "const float *"
... "int *" is incompatible with ... "const float *"
... "int *" is incompatible with ... "float *"
```

(b) Result of Listing 2(a) executed with the top `add` kernel function. Since the top `add` has incompatible parameter types with the inputs on line 33 a compilation error is returned. For brevity, only a portion of the error code is shown.

```
$ nvcc add.cuda.correct.cu -o add.cuda.correct.bin
$ ./add.cuda.correct.bin
0 0 0 0
```

(c) Result of Listing 2(a) executed with the second `add` kernel function. Since the second `add` has compatible parameter types with the inputs on line 33 there is not compilation error and the result is correct.

Listing 2 CUDA detects type mismatch correctly.

5. Wyvern

Overview. Let us delve into a concrete application and motivating example for the Basilisk architectural pattern—Wyvern. Wyvern is a simple EDSL that can be deployed in most programming languages. The Wyvern EDSL offers a low-level interface supporting the *single instruction, multiple threads* (SIMT) paradigm (Zhou, 2015). Code written using the Wyvern EDSL is translated into a Wyvern intermediate representation—referred to as WIR. The WIR represents the interface through which the EDSL interacts with the GPU. The WIR code can have several back-ends implementing its operations, and adding new back-ends will extend the range of supported hardware. Currently, there are two back-ends for WIR: (1) a CPU back-end, and (2) a Vulkan[4] back-end. The CPU back-end enables the execution of Wyvern EDSL on CPUs. In such scenarios, parallelism is only virtual, and the supposed parallel operations are properly sequentialized to be executed on the CPU while maintaining the expected semantics. The Vulkan back-end instead allows the execution of the WIR code on GPUs.

GPU Programming Background. Usually, a program for GPU computing is composed of two different portions of code: the *host code* and the *device code*. The host code is executed on the CPU. It handles data loading/preparation for the device code. For example, this snippet of CUDA code:

```
// Allocate memory on the device
cudaMalloc(&d_a, size);
cudaMalloc(&d_b, size);
cudaMalloc(&d_c, size);
// Copy host vectors to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
// Call a kernel function to be executed on the device
add<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

shows the allocation and initialization of three device vectors alongside a kernel function call—`add`. Instead, the device code is executed on the GPU. This is composed of one or more *kernel* functions, i.e., functions that will run on the GPU.

One of the widely used programming languages for GPGPU is CUDA. CUDA (Kirk, 2007; Soyata, 2020) is an extension of the C programming language that introduces syntax for interacting with GPU hardware (the device). Despite its efficiency, CUDA has two significant limitations: it is essentially closed source, and it is exclusive to NVIDIA GPUs.

For example, this snippet of CUDA code:

```
__global__ void add( int* a, int* b, int* c, int n ) {
   int id = blockIdx.x * blockDim.x + threadIdx.x;
   if ( id < n ) c[id] = a[id] + b[id];
}
```

declares a kernel function to sum two vectors (`a` and `b`) into a third one (`c`).

Of course, different frameworks handle the definition of the device code differently. For example, CUDA supports only NVIDIA GPUs with C, C++, and Fortran as host code. OpenCL, by the Khronos group, supports a wide range of GPUs, but it only supports C and C++ for the host code (OpenCL-C and OpenCL-C++ (Gaster and Howes, 2013)). Whereas, PyCUDA (Klöckner et al., 2012) is a third-party binding for CUDA in Python. In PyCUDA, the host code is written in Python whereas the device code is written in CUDA. PyCUDA supports only NVIDIA GPUs. Unfortunately, both in OpenCL and in PyCUDA, the device code is encoded as a string with all the well-known limitations for similar approaches, e.g., lack of static analysis and type checking (Khabibullin et al., 2015; Saffran et al., 2021).

Code As Text. Both OpenCL and PyCUDA encode device code into a string, which heavily limits the static analysis of the device code (Eghbali and Pradel, 2020). This situation leads to cases where running the same program with different frameworks yields different results. Listings 1(a) and 1(c), and 2(a) present the same GPGPU program developed with different frameworks: PyCUDA, OpenCL, and CUDA,

[4] Vulkan is a low-level graphics API developed by the Khronos Group. It provides a cross-platform and high-performance interface for rendering graphics, widely used in applications ranging from video games to professional graphics software.

respectively. The device code consists of a single kernel function: lines 9–16 in Listing 1(a), lines 3–11 in Listing 1(c), and lines 3–8 in Listing 2(a). All kernels perform vector addition between arrays `a` and `b`, storing the result in the array `c`. Let N be the array size. The host code, for all frameworks, allocates and initializes array `a` with -10^7 to $-(10^7 + N)$ entries and array `b` with 10^7 to $10^7 + N$ entries. One would expect the array `c` to be made up of 0 entries after addition, as shown in Listing 2(c). However, the results are quite different. Both PyCUDA and OpenCL frameworks result in an array `c` of -10^7 to $-(10^7 + N)$ entries. Notice that neither an error nor a warning is issued by either PyCUDA or OpenCL. Only the CUDA version issues an error about type incompatibility, shown in Listing 2(b). Without such an error, one may incorrectly assume that PyCUDA and OpenCL perform automatic casting from the host-declared type (`int`) to the device-declared type (`float`). However, upon close inspection, the `int` representation of 10^7 is interpreted as `float` numbers as 1.40130×10^{-38}. Meanwhile, the `int` representation of -10^7 is interpreted as `float` numbers as -3.07599×10^{38}. The sum of the `float` representation is practically identical to the latter (-3.07599×10^{38}). Converting back to the `int` representation, we recover the obtained result. Additionally, using consistent types fixes the issue in all frameworks, returning the correct results shown in Listing 2(c). While this situation is extremely undesirable as it can lead to disastrous outcomes, it could have been avoided by simply using an extra layer of abstraction on the device code declaration instead of simple strings.

Language Interfaces. With Wyvern, we aim to offer a consistent GPGPU programming EDSL library for D, C++, C#, Scala, Ruby, Hy, and Python programming languages. Of course, the interface needs to be adapted to the specific language capabilities. For example, whenever possible, operations should be overloaded with the appropriate operator.

Platforms. In this context, different platforms are represented by various processing hardware. For example, a CPU represents one possible platform. Additionally, GPU accelerators (and possibly *tensor processing unit*s, TPUs) represent potential targeted platforms for Wyvern.

Product-to-platform Interface. In Wyvern, the Product-to-platform interface is represented by the WIR intermediate representation (shown in Fig. 5-❶). WIR is a code representation for kernel functions that stands between the high-level Wyvern EDSL and the low-level SPIR-V code. WIR is in single static assignment form (Braun et al., 2013), operating on variables and buffers of types unsigned 32-bit integer (`U32`), signed 32-bit integer (`I32`), IEEE 754 binary32 number (`F32`), and boolean value (`Bool`). It provides logic-arithmetic operations, type conversions, indirect buffer access, and structured control flow. Only the constructs **`if`** and **`while`** are supported. WIR also has a well-defined JSON serialization generated automatically by `serde` (Scott et al., 2020). To support new platforms, developers need to provide a code generation module (composed of a `Executor`, `Executable`, and `Resource`) for each of the WIR constructs. Nonetheless, Wyvern already supports a wide range of GPUs by implementing the SPIR-V (Kessenich et al., 2021) platform. SPIR-V is a standardized low-level binary representation for GPGPU programs. Vulkan mandates all its implementors to provide a way to compile SPIR-V programs to their native GPU executable machine code. Its diffusion combined with its high performance makes Vulkan/SPIR-V the perfect platform target.

For example, the following code snippet provides the Vulkan implementation for the WIR `Op::Add` instruction:

```
Op::Add(r, a, d) => {
    match get_const_datatype(r) {
      U32  => {vk_builder.iadd(types.u32,&r,&a,&d)}
      I32  => {vk_builder.iadd(types.i32,&r,&a,&d)}
      F32  => {vk_builder.fadd(types.i32,&r,&a,&d)}
      _    => unreachable!(),
    };
}
```

The implementation utilizes the `rspirv` builder[5] (`vk_builder`) to construct the SPIR-V code depending on the operand types. Here, both unsigned and signed 32 bit utilize the same SPIR-V instruction, `iadd`, while 32 bit floating point numbers utilize the `fadd` instruction. This code snippet is part of the GPU back-end shown in Fig. 5-❷. Similarly, the CPU back-end's equivalent code snippet for the Add WIR instruction has the following implementation:

```
Op::Add(r, a, b) => {
  let v = match get_const_datatype(r) {
    U32 => {U32(Add::add(x, y))}
    I32 => {I32(Add::add(x, y))}
    F32 => {F32(Add::add(x, y))}
    _   => unreachable!(),
  };
}
```

The main difference is the direct execution instead of the indirect compilation towards SPIR-V. This snippet is part of the GPU back-end shown in Fig. 5-❸.

User-to-product Interface. The user-to-product interface provided by Wyvern is directly accessible to Rust users. As mentioned earlier, this interface is represented by a concise SIMT EDSL for executing kernel functions on both GPUs and CPUs. For instance, the following snippet demonstrates the addition of two arrays, `a` and `b`, into a third array, `c`, using the Wyvern EDSL:

```
let builder = ProgramBuilder::new();
// ...
let tid = Variable::new(&builder);
tid.store(builder.worker_id());
builder.while_loop(|_| tid.load().lt(100), |_| {
  let i = tid.load();
  c.at(i).store(a.at(i).load() + b.at(i).load());
  tid.store(i + builder.num_workers());
});
```

Developers interact with the EDSL via the `ProgramBuilder` class. For example, here, we access the while loop through the corresponding method `while_loop`. This method requires a conditional expression and a body to be executed and implements the relative loop semantics. Parallelism is achieved by employing multiple workers (with incremental `worker_id`s) that are concurrently executed to perform the addition on different array elements.

The main improvement, over a string kernel implementation (for PyCUDA and OpenCL in Figures 1(a) and 1(c)), is that the type correctness can be checked at compile time, so that, the previous error can be reported immediately preventing from being silently ignored.

Similarly to the previously presented case of MinPy, Wyvern EDSL is only accessible through the Rust programming languages. Developers who wish to use Wyvern need to implement an FFI interface for their language of choice accessing the C ABI provided by Wyvern (Fig. 5-❸). While effective for a single programming language, such a procedure becomes mechanical and error-prone when developers need to implement the FFI interface for different programming languages. Furthermore, different language interfaces may introduce different features rendering one language interface drastically different from the other. Instead, we use a one-to-many transpilation infrastructure, such as Hydra, to transpile the same language interface to all the interested programming languages.

The HydraKernel interface will need to handle all the EDSL components, such as `while_do` but also, among others, the executable components (representing the kernel programs) and the Program/ConstantBuilder components used to build the kernel programs.

```
trait Wyvern extends com.sun.jna.Library {/*...*/}
class ProgramBuilder {/*...*/}
class ConstantBuilder {/*...*/}
class Executable(executor: VoidPtr, program: String) {/*...*/}
// ...
```

[5] https://docs.rs/rspirv/latest/rspirv/dr/struct.Builder.html

Despite most of the heavy lifting being done by Wyvern the resulting language interface is still demanding. Consider that the HydraKernel code for implementing the Wyvern user-to-product interface consists of circa 1000 lines. Thus being able to transpile the resulting interface in different programming languages becomes extremely valuable. For instance, the resulting translation for Python will produce the classes that handle the program components, the executable components, and the building components.

```
import ctypes
libwyvern = ctypes.CDLL("libwyvern.so")
# ...
class ProgramBuilder:
  def add_cmd(self, op): # ...
class ConstantBuilder:
  def while_do(self,cnd,body): self.program.add_cmd(...)
  def if_else(self,cnd,body1,body2): self.program.add_cmd(...)
  def if_then(self,cnd,body): self.program.add_cmd(...)
class Executable:
  def run(self,): libwyvern.run(...)
```

Then, this components can be used like any other Python library. For example, the previously discussed addition of two vectors into a third one becomes:

```
builder = ProgramBuilder()
# ...
tid = Variable(builder);
tid.store(builder.worker_id());
def cond(): return tid.load() < 100
def body():
   i = tid.load()
   c[i] = a[i] + b[i]
   i = i.load() + builder.num_workers()
builder.while_loop(cond, body)
```

Abstraction Layer. The Wyvern abstraction layer aims to render the usage of one computing platform transparent to the developers. Wyvern currently supports a back-end for Vulkan-compatible GPGPUs and one for classical CPUs. Since both back-ends implement the same product-to-platform interface, as in MinPy, using one back-end wrt. the other is simply a matter of using one shared object wrt. the other. The result is that the same Wyvern kernel code can be run on both CPU and the GPU with little or no effort.

6. Evaluation

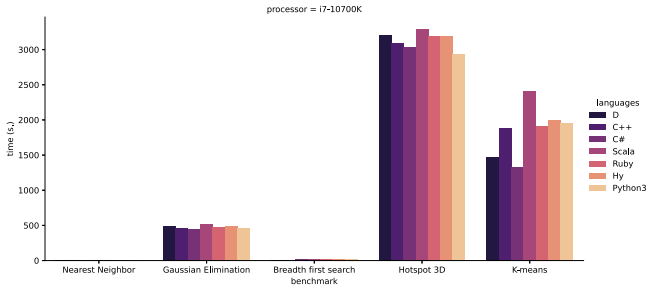
Overview. In this section, we will perform an evaluation of Wyvern, MinPy, and Hydra. Wyvern is evaluated by comparing time executions in different scenarios generated by varying: benchmarks, language interfaces, and platforms. Similarly, MinPy is evaluated by varying: language interfaces and platforms. Meanwhile, the latter, Hydra, is evaluated by measuring the cost of implementing a transpiler in terms of *lines of code* (LOCs).

We do not aim to compare Wyvern with other GPU computing frameworks (e.g., CUDA and OpenCL) or MinPy with other array computing libraries (e.g., ArrayFire[6]), as our focus is on language- and platform-independent software products. However, for completeness, when evaluating Wyvern, we provide CUDA and OpenCL benchmarks as a reference for target performance. Similarly, when evaluating MinPy, we provide benchmarks with direct Numpy and Torch implementations.

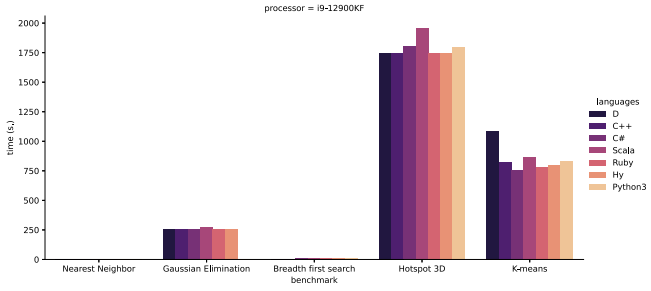
Hardware Setup. We conduct the evaluations on two different hardware setups:

- **Alpha**: A workstation equipped with an NVIDIA GPU RTX4080 and an Intel i7-10700K CPU.
- **Beta**: A workstation equipped with an NVIDIA GPU RTX3090 and an Intel i9-12900KF CPU.

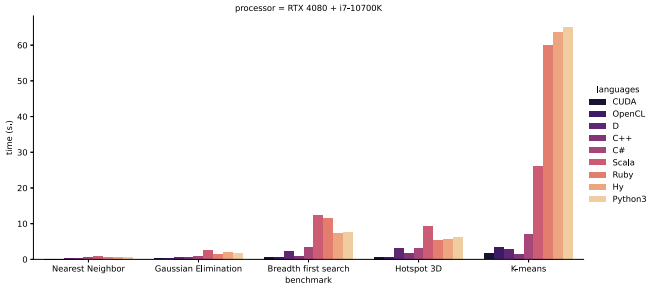
[6] https://arrayfire.org



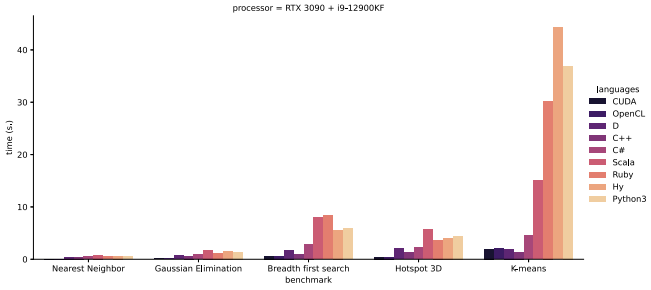
(a) Alpha CPU-only benchmark



(b) Beta CPU-only benchmark



(c) Alpha CPU+GPU benchmark



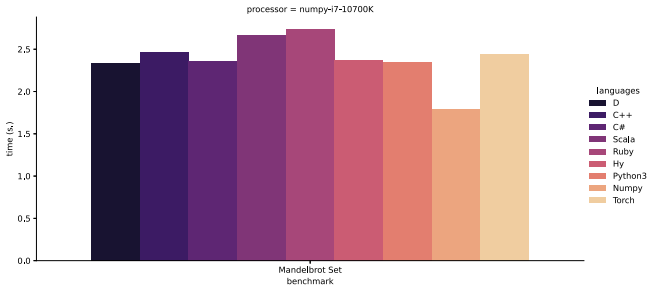
(d) Beta CPU+GPU benchmark

Fig. 6. Benchmark results for languages CUDA, OpenCL, D, C++, C#, Scala, Ruby, Hy, and Python for algorithms Nearest Neighbor, Gaussian Elimination, Hotspot 3D, Breadth First Search, and K-means from the open source Rodinia suite with two machines: Alpha and Beta.

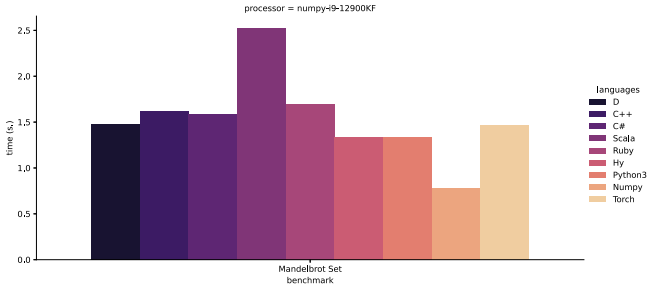
Platforms. Given the available hardware setups, to evaluate Wyvern software product we performed experiments on four platforms:

- CPU-only with the Alpha workstation,
- CPU-only with the Beta workstation,
- CPU+GPU with the Alpha workstation, and
- CPU+GPU with the Beta workstation.

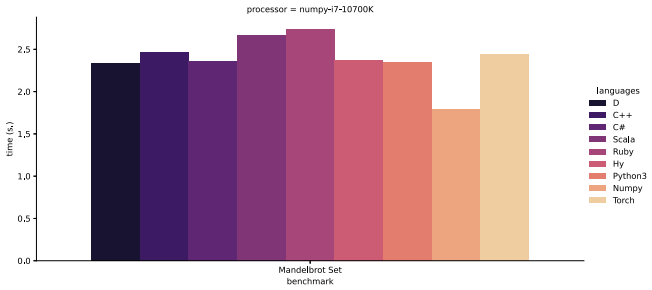
Instead, given the available hardware setups, to evaluate the MinPy software product we performed experiments on two platforms:



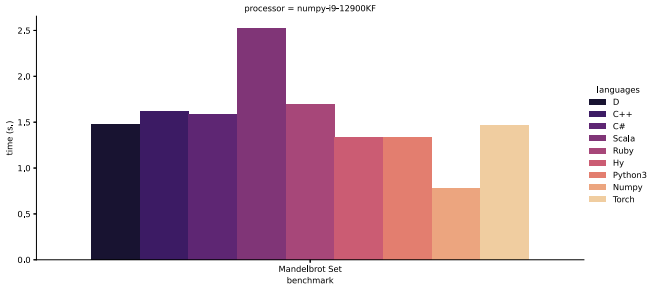
(a) Alpha with Numpy back-end



(b) Beta with Numpy back-end



(c) Alpha with Torch back-end



(d) Beta with Torch back-end

Fig. 7. Benchmark results for languages D, C++, C#, Scala, Ruby, Hy, Python, and NumPy for the Mandelbrot set algorithm with the NumPy C API back-end and the torch back-end with the high-end workstation Alpha and Beta.

- CPU-only with the Alpha workstation.
- CPU-only with the Beta workstation.

Language Interfaces. To represent a diverse range of programming paradigms and language features, we selected seven distinct programming languages for utilizing the Wyvern EDSL: D, C++, C#, Scala, Ruby, Hy, and Python. These languages embody various paradigms, including:

- object-oriented, e.g., Python;
- functional, e.g., Hy;
- dynamically typed, e.g., Ruby;
- statically typed, e.g., C#; and
- garbage-collected, e.g., Scala;
- non-garbage-collected, e.g., C++.

All these language interfaces are generated from a single HydraKernel interface, which has been translated into seven programming languages. These EDSLs are compared against the same CUDA and OpenCL benchmarks. However, performance discrepancies are to be expected, as both CUDA and OpenCL target highly performant programming languages such as C and C++. Meanwhile, Wyvern targets both performant and non-performant languages (e.g., C++ vs. Python), additionally introducing a level of indirectness with FFI.

Benchmarks. To demonstrate the capabilities of Wyvern, we chose 5 GPU benchmarks from the Rodinia benchmark suite[7]:

- nearest neighbor,
- Gaussian elimination,
- hotspot 3d,
- breadth-first search, and
- k-means.

For each benchmark, we developed the equivalent versions for CUDA, OpenCL, D, C++, C#, Scala, Ruby, Hy, and Python (for a total of 45 benchmarks).

Instead, the capabilities of MinPy are measured against a single array computing benchmark: the array computation of the Mandelbrot set.

Results for Wyvern. The results of the Wyvern evaluation are displayed in Fig. 6. CPU-only platforms are significantly slower than their CPU+GPU counterparts, with the former taking up to an hour to complete compared to the latter's maximum of 1 min. This behavior is to be expected as the CPU back-end cannot apply any kind of parallelization or vectorization. Furthermore slowdowns can be blamed on the simulation of the kernel execution. Thus, algorithms that require fewer executions of the kernel, such as the breadth-first search algorithm, perform better. Additionally, compiled languages such as D, C++, and C# are usually faster compared to others which is most noticeable in the CPU+GPU benchmarks where Python, Hy, and Ruby result to be the slowest languages. Notably, while OpenCL and CUDA offer the highest performance, Wyvern remains competitive with high-performance languages such as D and C++.

Regardless of the performance, apart from CUDA and OpenCL, benchmarks are implemented using the same API generated from the HydraKernel source code. Therefore, the benchmarks exploit a coherent API to implement the respective algorithms. For example, the following snippets (from the C# and Ruby implementations for the Hotspot3d benchmark) showcase the usage of API in different languages.

```
c.IfElse(
   () => {return condition0;},
   () => {tmp0.Store(lexpr0); return null;},
   () => {tmp0.Store(C); return null;});
```

```
c.if_else
  -> {return condition0},
  -> {tmp0.store(lexpr0); return nil},
  -> {tmp0.store(tC); return nil};
```

Results for MinPy. The results for the MinPy evaluation are shown in Fig. 7. Most notably, the direct NumPy implementation is faster wrt. other implementation. This is because the language interfaces use NumPy C Array API which requires a degree of interoperability with the Python C API. Therefore, the generated interface uses a higher level of indirectness wrt. the direct NumPy implementation. Nonetheless,

[7] https://github.com/yuhc/gpu-rodinia

Table 1
Line of code to define each target language. Hy is 0 as it reuses the Python target language.

Language	D	C++	C#	Scala	Hy	Python	Ruby	AVG.
LoC	315	302	319	307	0	279	306	261.14

the NumPy interface performs on par with the direct Torch implementation. Therefore, while there is surely an introduced overhead, it remains fairly controlled. Furthermore, a direct C implementation wrt. using either the Python C API or the NumPy C API would surely provide more efficiency.

Regardless of performance, the same algorithm is implemented consistently using the same API in all seven programming languages which results in consistent implementations, as showcased in the following snippets (D and Hy):

```
auto x = linspace(-2, 1, size, FLOAT64)
   .reshape([1, size])
   .repeat(size,0);
```

```
(setv x
   (  .
      (  .
         (linspace -2 1 size FLOAT64)
         (reshape #(1 size)))
      (repeat size 0))
   )
```

Here, we create a 1-dimensional array of `size` floats evenly spaced between −2 and 1 using `linspace`. Next, with `reshape` and `repeat`, we repeat the array along a new dimension.

Results for Hydra. The results for the Hydra evaluation are shown in Table 1. We measured the cost of adding new language interfaces, by measuring the lines of code of the respective Hydra template. Firstly, all templates amount to mostly the same LOC. This result is a byproduct of the form-filling style of the target language definition approach of Hydra. Secondly, the whole target language definition lies in less than 300 LOC on average. This means that there are around 300 fields to be filled per language. Of course, the more plugins a Hydra template uses the more fields will need to be filled but the more complete the Hydra kernel will be.

7. Discussion

In the light of these findings, we can try to answer the research questions presented in Section 1.

RQ_1. Is the Basilisk architectural pattern successful in rendering a software product language-independent?

As demonstrated by our experiments with the Rodinia benchmark suite and the Mandelbrot set, the Basilisk pattern enables Wyvern and MinPy, respectively, to be utilized by a variety of programming languages encompassing different paradigms and language features. Moreover, the pool of compatible languages can be expanded by adding new language templates to Hydra. Therefore, we can conclude that the Basilisk architectural pattern is successful in achieving language independence for the software product. The only conceivable limitations concern specific language features. For instance, one requirement for implementing both Wyvern, MinPy and Hydra language interfaces was the availability of FFI in the target language. Without this requirement, the entire software product would need to be implemented in HydraKernel, potentially restricting performance. Another requirement is Turing completeness; otherwise, the user-to-product interface would need to be severely restricted to be translatable into the non-Turing-complete target language.

Developers may face a limitation while implementing the Basilisk architectural pattern due to the different functionalities offered by various platforms. In case the product-to-platform interface requires functionalities that are not available on a particular platform, developers have two options. They can either implement the required functionality directly in the back-end or if this is not feasible or too expensive, they can choose to drop support for the lacking platform.

RQ_2. Is the Basilisk architectural pattern successful in rendering a software product platform-independent?

As illustrated in Figs. 6, and 7, the Wyvern and MinPy software products seamlessly operate on the respective different platforms: different hardware (e.g., CPU and CPU+GPU), and different array computing libraries (e.g., Torch and NumPy). The Wyvern abstraction layer, comprising CPU and Vulkan back-ends, effectively renders the Wyvern EDSL platform-independent. Similarly, the abstraction layer for MinPy renders the code written in one back-end executable with a different one. The primary limitation of these abstraction layers is the manual alignment requirement for different back-ends. Modifications to the product-to-platform interface cannot be automatically propagated to each back-end, leading to potential drawbacks and error-proneness. Notably, a similar issue is addressed for the user-to-product interface by introducing a common language (HydraKernel) from which all language interfaces could be generated. An analogous solution would necessitate the development of an intermediate back-end from which other back-ends could be derived. However, we believe that such an approach would pose significant implementation challenges, as the back-ends are intrinsically intertwined with the application domain, and consequently, the intermediate back-end would need to reflect this inherent dependency.

RQ_3. What is the cost of introducing new languages and/or platforms?

In our scenarios, the cost of adding a new programming language is approximately 300 LOC. However, it is important to note that a significant portion of these lines are reusable, as transpilers can be adapted to be used with different software products. Therefore, we conclude that the Basilisk architectural pattern is largely successful in maintaining low development costs for transpiler definition. However, it is worth mentioning that introducing programming languages significantly different from those presented in this work (e.g., Prolog) may require developers to customize existing plugins. In general, it is always possible to write Hydra plugins to perform the transpilation from HydraKernel to any target language, provided that the target language is Turing-complete. In some cases, developers may not be able to reuse the existing stack, requiring additional work to define the transpiler.

8. Threats to validity

External Validity. In this study, we utilized Hydra as a one-to-many transpilation infrastructure. We developed several transpilers encompassing a variety of language paradigms and language features. However, not all programming paradigms were tested, e.g., we did not investigate the declarative programming paradigm. Ultimately, the effectiveness of the Hydra infrastructure remains uncertain for these untested paradigms. The cost of introducing a new transpiler could be significant, potentially exceeding 300LOC, due to the need to implement additional infrastructure. Additionally, the extent of code reusability across newly introduced target languages is unclear. To mitigate these concerns, we incorporated a diverse range of programming languages into our study: D, C++, C#, Scala, Ruby, Python, and Hy all of which could be developed with a few hundreds of declarative code lines.

Further, the Basilisk architectural pattern may not map well to all application domains, to mitigate this issue, we instantiate the pattern in two different application domains: GPGPU computing and array computing.

Internal Validity. The validity of the reported findings could be compromised due to the implementation of the same benchmark from the Rodinia benchmark suite multiple times in different languages. This approach raises concerns about the introduction of bugs and inconsistencies in the expected behavior of the benchmark across different languages. To mitigate this issue, we selected benchmarks with the simplest and smallest implementations. This choice facilitated a more straightforward one-to-one translation between languages and minimized the likelihood of introducing errors.

The obtained results could be influenced by our familiarity with Basilisk, as we developed instantiations in Hydra, MinPy, and Wyvern. This extensive knowledge may have inadvertently steered us away from potential pitfalls. In an effort to mitigate this bias, we had the benchmarks created by individuals who were not directly involved in the Basilisk development.

9. Related works

In this section, we summarize a few among the many related works that the community has proposed during the years targeting transpilers, and platform-independent software.

Platform independence can be achieved by means of quality code design. While we cited only an approach based on established designed patterns as those found in Gamma et al. (1995), more modern alternatives do exist. For example, Almeida et al. (2003) propose an abstraction layer based on services. They argue that by describing the application-platform interaction as services the software product becomes naturally platform-independent. Also, Selic (2005) treats platforms as service-offering entities. Platform independence is achieved by means of an abstract platform. Blanco and Lucrédio (2021) propose to use a high-level general purpose language, named GPL, to describe the product-to-platform interface. Then each back-end provides a GPL implementation or compiler, so that, the GPL code can be executed seamlessly on different platforms. Instead, He et al. (2005) target web applications by proposing a model-driven approach (Selic, 2003; France and Rumpe, 2007). Another notable example is embodied by Corda et al. (2019), here platform independence is achieved by exploiting the LLVM intermediate representation. Hsieh and Chen (2015) develop a series of ten design patterns to support continuous integration in cross-platform applications. Chadha et al. (2017) develop a semi-automatic approach, with a relevant degree of success, to translate IOS into equivalent Android code and vice versa using popular web-based programming resources.

Transpilers are heavily studied concepts. However, often, the focus is only on translating pairs of languages. For instance, Schultes (2021) focuses on Swift and Kotlin. Meanwhile, Shetty et al. (2019) focus on C/C++ and Rust languages. Related to GPGPU, Tabuchi et al. (2013) discuss a transpiler from OpenACC to CUDA. Albrecht et al. (1980) have focused on Ada and Pascal programming languages.

While not being the focus of Hydra, source-to-source translators have been used to improve performance (Loveman, 1977). For example, Cetus (Schordan and Quinlan, 2003) uses a source-to-source transformation to automatically introduce parallelization, thus improving performance; Krzikalla et al. (2011) also discuss automatic loop vectorization. Alias et al. (2007) introduce a memory contraction implementation using source-to-source translation. Quinlan and Liao (2011) propose ROSE,[8] a compiler infrastructure to support various kinds of software optimizations and verifications. ROSE, during the years, has been used in several research projects for both compilers (Liao et al., 2010; Schmitt et al., 2014) and transpilers (Lidman et al., 2012). Similarly, DMS[9] is a compiler infrastructure based on AST transformations (Baxter et al., 2004). A compilation infrastructure, Nanopass, discussed in Ringo et al. (2023), Keep and Dybvig (2013), Sarkar et al. (2004), is used to develop compilers as a sequence of many small passes rather than a few complicated ones. Similarly, an A* search-based compilation infrastructure is proposed in Bertolotti et al. (2024).

More akin to Hydra, HaXe[10] is one-to-many, source-to-source translator (Ponticelli and McColl-Sylveste, 2008). While more mature and more studied (Štrekelj et al., 2015), it supports only a fixed size number of languages. Meanwhile, Hydra is designed with extensibility in mind. Thus, new languages can be introduced by simply declaring language features (as shown in Section 3).

10. Conclusion

In this work, we proposed Basilisk. An architectural pattern for the development of language- and platform-independent software products. We achieved these properties by means of an abstraction layer which can be achieved through quality software design. Meanwhile, language independence is achieved by means of transpilation. We also instantiate the pattern for the development of SIMT development EDSL. To achieve platform independence we develop Wyvern, the abstract layer over the computational platforms. To achieve language independence, we developed Hydra, a source-to-source transpilation infrastructure, which we used to render Wyvern EDSL available to seven more languages (other than the native one: Rust).

The code of Hydra, Wyvern, and MinPy alongside with the code necessary to reproduce the presented experiments is available at

https://doi.org/10.5281/zenodo.11058650

CRediT authorship contribution statement

Francesco Bertolotti: Writing – review & editing, Writing – original draft, Software, Investigation, Conceptualization. **Walter Cazzola:** Writing – review & editing, Writing – original draft, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Dario Ostuni:** Software. **Carlo Castoldi:** Software.

Declaration of competing interest

We wish to confirm that there are not known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Data availability

A replication package is available on Zenodo

https://doi.org/10.5281/zenodo.11058650

.

Acknowledgments

The authors wish to thanks Ermanno Righini for his help with the experiment.

References

Aho, A.V., Sethi, R., Ullman, J.D., 1986. Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, Massachusetts.

Albrecht, P.F., Garrison, Philip E., Graham, S.L., Hyerle, R.H., Ip, P., Krieg-Brückner, B., 1980. Source-to-source translation: Ada to Pascal and Pascal to Ada. ACM Sigplan Not. 15, 183–193.

Alias, C., Baray, F., Darte, A., 2007. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator ROSE. Sigplan Not. 42, 73–82.

Almeida, J.P., van Sinderen, M., Ferreira Pires, L., Quartel, D., 2003. A systematic approach to platform-independent design based on the service concept. In: Proceedings of the 7th International Enterprise Distributed Object Computing Conference. EDOC'03, IEEE, Brisbane, Australia, pp. 112–123.

[8] http://rosecompiler.org/

[9] http://www.semdesigns.com/Products/DMS/DMSToolkit.html

[10] https://haxe.org/

Ancona, D., Ancona, M., Cuni, A., Matsakis, N.D., 2007. RPython: A step towards reconciling dynamically and statically typed OO languages. In: Costanza, P., Hirschfeld, R. (Eds.), Proceedings of the 3rd International Symposium on Dynamic Languages. DLS'07, ACM, Montréal, Canada, pp. 53–64.

Baxter, I.D., Pidgeon, C., Mehlich, M., 2004. DMS®: Program transformations for practical scalable software evolution. In: Estublier, J., Rosenblum, D. (Eds.), Proceedings of the 26th International Conference on Software Engineering. ICSE'04, IEEE, Edinburgh, Scotland, pp. 625–634.

Bertolotti, F., Cazzola, W., Favalli, L., 2024. ★Piler: Compilers in search of compilations. J. Syst. Softw. 212, http://dx.doi.org/10.1016/j.jss.2024.112006.

Bishop, J., Horspool, N., 2006. Cross-platform development: Software that lasts. IEEE Comput. 39, 26–35.

Blanco, J.Z., Lucrédio, D., 2021. A holistic approach for cross-platform software development. J. Syst. Softw. 179.

Box, D., Sells, C., 2002. Essential.Net: The Common Language Runtime, vol. 1, Addison-Wesley.

Braun, M., Buchwald, S., Hack, S., Leißa, R., Mallon, C., Zwinkau, A., 2013. Simple and efficient construction of static single assignment form. In: Jhala, R., Bosschere, K. (Eds.), Proceedings of 22nd International Conference on Compiler Construction. CC'13, Springer, Rome, Italy, pp. 102–122.

Chadha, S., Byalik, A., Tilevich, E., Rozovskaya, A., 2017. Facilitating the development of cross-platform software via automated code synthesis from web-based programming resources. Comput. Lang. Syst. Struct. 48, 3–19.

Coco, E.J., Osman, H.A., Osman, N.I., 2018. JPT: A simple java-python translator. Comput. Appl. Int. J. 5.

Corda, S., Singh, G., Awan, A.J., Jordans, R., Corporaal, H., 2019. Platform independent software analysis for near memory computing. In: Konofaos, N., Kitsos, P., Žemv, A. (Eds.), Proceedings of the 22nd Euromicro Conference on Digital System Design. DSD, IEEE, Kallithea, Greece, pp. 606–609.

Draxler, C., Jänsch, K., 2004. SpeechRecorder—A universal platform independent multi-channel audio recording software. In: Calzolari, N. (Ed.), Proceedings of the Fourth International Conference on Language Resources and Evalation. LREC'04, ELRA, Lisbon, Portugal, pp. 559–562.

Dubochet, G., 2011. Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming (Phd thesis). École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

Eghbali, A., Pradel, M., 2020. No strings attached: An empirical study of string-related software bugs. In: Le Goues, C., Lo, D. (Eds.), Proceedings of the 35th International Conference on Automated Software Engineering. ASE'20, IEEE, pp. 956–967, Online.

France, R.B., Rumpe, B., 2007. Model-driven development of complex software: A research roadmap. In: Briand, L.C., Wolf, A.L. (Eds.), Proceedings of Future of Software Engineering. FoSE'07, IEEE Computer Society, Minneapolis, MN, USA, pp. 37–54.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. In: Professional Computing Series, Addison-Wesley, Reading, Ma, USA.

Gaster, B.R., Howes, L., 2013. OpenCL C++. In: Cavazos, J., Gong, X., Kaeli, D. (Eds.), Proceedings of the 6th Workshop on General Purpose Processor using Graphics Processing Units. GPGPU'13, ACM, Houston, TX, USA, pp. 86–95.

Grimmer, M., Schatz, R., Seaton, C., Würthinger, T., Luján, M., Mössenböck, H., 2018. Cross-language interoperability in a multi-language runtime. ACM Trans. Program. Lang. Syst. 40, 1–43.

Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E., 2020. Array programming with NumPy. Nature 585, 357–362.

Hatledal, L.I., Styve, A., Hovland, G., Zhang, H., 2019. A language and platform independent co-simulation framework based on the functional mock-up interface. IEEE Access 7, 109328–109339.

He, C., He, F., He, K., Tu, W., 2005. Constructing platform independent models of web application. In: Bai, X., Xu, J. (Eds.), Proceedings of the International Workshop on Service-Oriented System Engineering. SOSE'05, IEEE, Beijing, China, pp. 77–81.

Hsieh, C.Y., Chen, C.T., 2015. Patterns for continuous integration builds in cross-platform agile software development. J. Inf. Sci. Eng. 31, 897–924.

Keep, A.W., Dybvig, R.K., 2013. A nanopass framework for commercial compiler development. In: Uustalu, T. (Ed.), Proceedings of the 18th International Conference on Functional Programming. ICFP'13, ACM, Boston, MA, USA, pp. 343–350.

Kessenich, J., Ouriel, B., Krisch, R., 2021. SPIR-V Specification. Technical Report, Khronos Group.

Khabibullin, M., Ivanov, A., Grigorev, S., 2015. On development of static analysis tools for string-embedded languages. In: Puntikov, N. (Ed.), Proceedings of the 11th Central and Eastern European Software Engineering Conference in Russia. CEE-SECR'15, ACM, Moscow, Russia, pp. 5:1–5:10.

Kirk, D., 2007. NVIDIA CUDA software and GPU parallel computing architecture. In: Sagly, M. (Ed.), Proceedings of the 6th International Symposium on Memory Management. ISMM'07, ACM, Montréal, Canada, pp. 103–104.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A., 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. J. Parallel Comput. 38, 157–174.

Kosar, T., Bohra, S., Mernik, M., 2016. Domain specific languages: A systematic mapping study. Inf. Softw. Technol. 71, 77–91.

Kosar, T., Martínez López, P.E., Barrientos, P.A., Mernik, M., 2008. A preliminary study on various implementation approaches of domain-specific languages. Inf. Softw. Technol. 50, 390–405.

Krzikalla, O., Feldhoff, K., Müller-Pfefferkorn, R., Nagel, W.E., 2011. Scout: A source-to-source transformator for SIMD-optimizations. In: Proceedings of the Parallel Processing Workshops. Euro-Par'11, Springer, Bordeaux, France, pp. 137–145.

Latif, M., Lakhrissi, Y., 2016. Cross platform approach for mobile application development: A survey. In: Proceedings of the 2nd International Conference on Information Technology for Organizations Development. IT4OD'16, IEEE, Fezm Morocco, pp. 1–5.

Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R., 2010. A ROSE-based openmp 3.0 research compiler supporting multiple runtime libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., Supinski, B.R. (Eds.), Proceedings of the 6th International Workshop on OpenMP. IWOMP'10, Springer, Tsukuba, Japan, pp. 15–28.

Lidman, J., Quinlan, D.J., Liao, C., McKee, S.A., 2012. ROSE::FTTransform—A source-to-source translation framework for exascale fault-tolerance research. In: Mendelson, A., van Moorsel, A., Steininger, A. (Eds.), Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops. DSN'12, IEEE, Boston, MA, USA, pp. 1–6.

Loveman, D.B., 1977. Program improvement by source-to-source transformation. J. ACM 24, 121–145.

Mernik, M., Heering, J., Sloane, A.M., 2005. When and how to develop domain specific languages. ACM Comput. Surv. 37, 316–344.

Pang, A., Anslow, C., Noble, J., 2018. What programming languages do developers use? A theory of static vs dynamic language choice. In: Kelleher, C., Engels, G. (Eds.), Proceedings of the Symposium on Visual Languages and Human-Centric Computing. VL/HCC'18, IEEE, Lisbon, Portugal, pp. 239–247.

Poltavtseva, M.A., 2019. Evolution of data management systems and their security. In: Gvozdetsky, V., Zakharov, V. (Eds.), Proceedings of the International Workshop on Engineering Technologies and Computer Science. EnT'19, IEEE, Moscow, Russia, pp. 25–29.

Ponticelli, F., McColl-Sylveste, L., 2008. Professional HaXe and Neko. John Wiley and Sons.

Quinlan, D., Liao, C., 2011. The rose source-to-source compiler infrastructure. In: Midkiff, S., Eigenmann, R., Bae, H. (Eds.), Proceedings of the Cetus Users and Compiler Infrastructure Workshop. Galveston, TX, USA, pp. 1–3.

Ringo, N., Kramer, L., Van Wyk, E., 2023. Nanopass attribute grammars. In: Degueule, T., Scott, E. (Eds.), Proceedings of the 16th International Conference on Software Language Engineering. SLE'23, ACM, Cascais, Portugal, pp. 70–83.

Saffran, J.a., Barbosa, H., Quintão Pereira, F.M., Vladamani, S., 2021. On-line synthesis of parsers for string events. J. Comput. Lang. 62.

Sarkar, D., Waddell, O., Dybvig, R.K., 2004. A nanopass infrastructure for compiler education. In: Okosaki, C., Fisher, K. (Eds.), Proceedings of the 9th International Conference on Functional Programming. ICFP'04, ACM, Snow Bird, UT, USA, pp. 201–212.

Schmitt, C., Kuckuk, S., Hannig, F., Köstler, H., Teich, J., 2014. ExaSlang: A domain-specific language for highly scalable multigrid solvers. In: Krishnamoorthy, S., Ramanujam, J., Sadayappan, P. (Eds.), Proceedings of the 4th Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing. WOLF-HPC'14, IEEE, New Orleans, LA, USA, pp. 42–51.

Schordan, M., Quinlan, D., 2003. A source-to-source architecture for user-defined optimizations. In: Böszörményi, L., Schojer, P. (Eds.), Proceedings of the Joint Modular Language Conference. JML'03, Springer, Klagenfurt, Austria, pp. 214–223.

Schultes, D., 2021. SequalsK—A bidirectional Swift-Kotlin-Transpiler. In: Abreu, R., Fazzini, M. (Eds.), Proceedings of the 8th International Conference on Mobile Software Engineering and Systems. MobileSoft'21, IEEE, Madrid, Spain, pp. 73–83.

Scott, N.W., Hodson, D.D., Dill, R., Grimaila, M.R., 2020. Using serde to serialize and deserialize DIS pdus. In: Arabnia, H.R., Deligiannidis, L., Tinetti, F.G., Tran, Q.N. (Eds.), Proceedings of the International Conference on Computational Science and Computational Intelligence. CSCI'20, IEEE, Las Vegas, NV, USA, pp. 1425–1428.

Selic, B., 2003. The pragmatics of model-driven development. IEEE Softw. 20, 19–25.

Selic, B., 2005. On software platforms, their modeling with UML 2, and platform-independent design. In: Ghafoor, A., Brinkschulte, U., Ramamritham, K., Pettit, R.G. (Eds.), Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC'05, IEEE, Seattle, WA, USA, pp. 15–21.

Seymour, K., Dongarra, J., 2003. Automatic translation of fortran to JVM bytecode. Concurr. Comput.: Pract. Exper. 15, 207–222.

Shetty, N., Saldanha, N., Thippeswamy, M.N., 2019. CRUST: A C/C++ to rust transpiler using a nano-parser methodology' to avoid C/C++ safety issues in legacy code. In: Emerging Research in Computing, Information, Communication and Applications. In: Advances in Intelligent Systems and Computing, vol. 882, Springer, pp. 241–250.

Slee, M., Agarwal, A., Kwiatkowski, M., 2007. Thrift: Scalable cross-language services implementation. White Paper 5. Facebook.

Soyata, T., 2020. GPU Parallel Program Development using CUDA. CRC Press.
Štrekelj, D., Leventić, H., Galić, I., 2015. Performance overhead of haxe programming language for cross-platform game development. Int. J. Electr. Comput. Eng. Syst. 6, 9–13.
Tabuchi, A., Nakao, M., Sato, M., 2013. A source-to-source openacc compiler for CUDA. In: Proceedings of the Parallel Processing Workshops. Euro-Par'13, Springer, Aachen, Germany, pp. 178–187.
Vinoski, S., 1997. CORBA: Integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. 35, 46–55.
Wang, A., Yan, X., Wei, Z., 2018. ImagePy: An open-source, python-based and platform-independent software package for bioimage analysis. Bioinformatics 34, 3238–3240.
Webb, B.J., Utting, M., Hayes, I.J., 2021. A formal semantics of the graalvm intermediate representation. In: Hou, Z., Ganesh, V. (Eds.), Proceedings of the International Symposium on Automated Technology for Verification and Analysis. ATVA'21, Springer, Gold Coast, QLD, Australia, pp. 111–126.
Würthinger, T., Wimmer, C., Woß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M., 2013. One VM to rule them all. In: Hirschfeld, R. (Ed.), Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. Onward!'13, ACM, Indianapolis, IN, USA, pp. 187–204.
Zhang, Z., Liu, Y., Yu, S., Li, X., Yun, Y., Fang, C., Chen, Z., 2022. UniRLTest: Universal platform-indipendent testing with reinforcement learning via image understanding. In: Smaragdakis, Y. (Ed.), Proceedings of the 31st International Symposium on Software Testing and Analysis. ISSTA'22, ACM, pp. 805–808, Online.
Zhou, H., 2015. SIMT architecture. In: Solihin, Y. (Ed.), Fundamentals of Parallel Multicore Architecture, first ed. CRC Press, pp. 409–426 (Chapter 12).

Francesco Bertolotti is currently a Computer Science Postdoctoral Researcher at Università degli Studi di Milano and a member of the ADAPT Laboratory. He got his Ph.D. in computer science from the Università degli Studi di Milano. Previously, he was an assistant researcher at the same University where he also got his master degree in Computer Science. His research interests are programming languages, software quality, machine/deep learning techniques and their reciprocal cross-fertilization. He can be contacted at francesco.bertolotti@unimi.it for any question.

Walter Cazzola is currently a Full Professor in the Department of Computer Science of the Università degli Studi di Milano, Italy and the Chair of the ADAPT laboratory. Dr. Cazzola designed the mChaRM framework, @Java, [a]C#, Blueprint programming languages and he is currently involved in the designing and development of the Neverlang language workbench. He also designed the JavAdaptor dynamic software updating framework and its front-end FiGA. He has written over 100 scientific papers. His research interests include (but are not limited to) software maintenance, evolution and comprehension, programming methodologies and languages. He served on the program committees or editorial boards of the most important conferences and journals about his research topics. He is associate editor for the Journal of Computer Languages published by Elsevier. More information about Dr. Cazzola and all his publications are available at https://cazzola.di.unimi.it and he can be contacted at cazzola@di.unimi.it for any question.

Dario Ostuni is currently a Ph.D. student in Computer Science at the University of Verona. He received his M.Sc. in Computer Science at the University of Milan, where he was involved in the research activities of the ADAPT laboratory. His research interests concern algorithms and data structures, computational complexity and combinatorial optimization. He can be contacted at dario.ostuni@univr.it for any inquiry.

Carlo Castoldi is resarch associate at CNR. He received his master and bachelor in Computer Science at the University of Milan, where he was involved in the research activities of the ADAPT laboratory. His research interests concern programming languages. He can be contacted at carlo.castoldi@studenti.unimi.it for any question.