Master research Internship

Bibliographic report

Self-Adaptable Virtual Machines

Domain: Programming Languages - Performance

Author:
Gwendal JOUNEAUX

Supervisor:
Benoit COMBEMALE
Olivier BARAIS
IRISA – DiverSE

Abstract: With the growing need to define domain-specific knowledge inside applications, creating Domain-Specific Languages(DSLs) became a logical solution. In this context, multiple tools were developed to ease the creation of DSL interpreters using generic patterns to implement those interpreters. However, this context of development hamper fine-grained language-specific optimizations. We propose in this bibliography to study approaches and architectural patterns to dynamically optimize the performance and energy consumption of programming language interpreters. We reviewed interpreter's design patterns, DSL interpreter performances enhancment technics, dynamic adaptive system structures and evalution methods. We also review energy consumption, but there is no known attempt to introduce generic approach to manage the energy consumption concerns inside domain-specific interpreters, that's why we survey general knowledge on this topic. Finally, we propose a self-healing dynamically adaptive JIT tuning system.

Contents

1 Introduction **1**

2 Programming language design and implementation **1**
2.1 Building-blocks of a programming language 1
2.2 Existing implementation patterns of interpreters 3

3 Performances and energy management in DSL interpreters **4**
3.1 Performance optimizations of DSLs interpreters 4
3.1.1 Approximate computing 5
3.1.2 Multi-stage programming 5
3.1.3 Just-In-Time Compilers 6
3.2 Energy consumption 8

4 Architecture for dynamically adaptable systems **10**
4.1 Feedback Control loops 10
4.2 Self-healing software 10

5 Performance and enrgy consumption benchmarking **11**
5.1 The challenge of benchmarking 11
5.2 Mitigate the non-deterministic behavior 11
5.3 Addressing bias with statistics 12

6 Conclusion **13**

1 Introduction

In the past few years, researchers in software language engineering tried to reduce the costs of programming language creation. This comes with different approaches like improvement of language workbenches usability and language modularity and composition. With those approaches, the design of a new language became an affordable solution to address domain-specific problems. However, such a generic approach hamper fine-grained optimization of the resulting language.

In this context, we might want to make a hand-crafted language with the risk of poor tooling and high development and maintenance costs. To avoid that, we want to create a new solution that benefits from the language workbenches ecosystem with the ability to optimize our language.

In this bibliography, we focus on programming language interpreters because we want to create a dynamic pattern that works in an open world while compilers work in a closed world. Language interpreters are a widely used method to define a language semantic and most of the language workbenches can generate a generic, but empty, implementation of interpreter. We review in this bibliography approaches and architectural patterns to dynamically optimize the performance and energy consumption of programming language interpreters to propose research directions to the following question :

"Can we design a DSL interpreter as a dynamic adaptative system that leverages on domain-specific information and apply optimization at runtime for the sake of performance and energy consumption?"

The rest of this document is structured as follows. First, section 2 recall the basics of programming language design and implementation. Then, in section 3, we address DSL interpreter performances and energy consumption. In section 3.1 present different solutions used to enhance the performances of an interpreter while section 3.2 discuss energy consumption at a language level. A review of relevant dynamic adaptative systems structures is presented in Section 4. Section 5 highlight the difficulty in assessing performance or energy consumption and the possible solution to do reproducible benchmarking. Finally, in Section 6, we discuss how we can relate the general model of dynamic systems to existing optimizations.

2 Programming language design and implementation

In this section, we recall the basics of language design and implementation using object oriented paradigm. Then we will focus on interpreters to see in detail their implementations especially the different patterns used to define them using a simple calculator language as example. We will provide a short comparison of the different implementation technics.

2.1 Building-blocks of a programming language

Nowadays, we use three formalisms to define programming languages in the object oriented paradigm. The abstract syntax defines relations between the concepts of the language, the concrete syntax explicit writing rules for the language and the semantics which is the meaning of what you want to express in the language [1].

The abstract syntax is a meta-model that contains useful information (concepts and relations between them) of the language. This meta-model represents the structure of the data that are represented by our language. Figure 1 show a possible abstract syntax of the calculator language we proposed as an example. We can read this meta-model this way:

- `NumberLiteral` has an integer value
- All operators have two sub-expressions
- `MulOrDiv` save the operator as a string
- All classes extends `Expression`

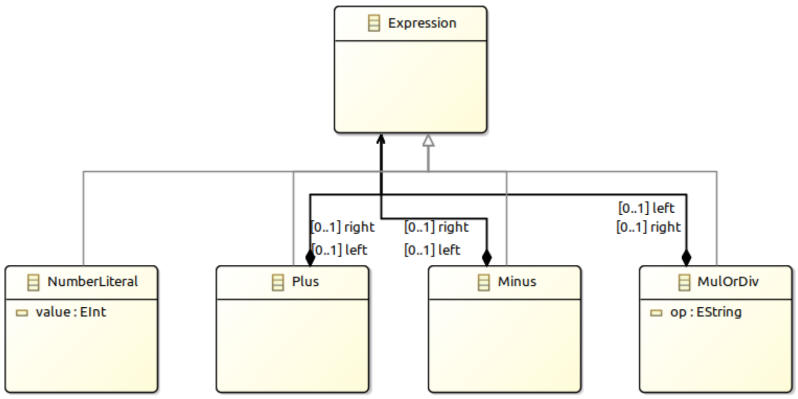
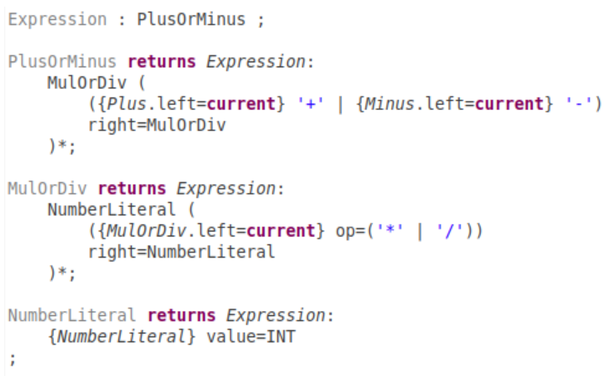


Figure 1: EMF meta-model of a simple calculator language

Nowadays, the Eclipse Modeling Framework [2] is a widely used formalism to define an abstract syntax and come with an entire ecosystem of tools like editors, validators or generators. For example, the language workbench xText has in its latest versions integrated EMF models in its workflow.

The concrete syntax is the definition of the concrete representation of your language. It can be defined either as a visual or textual representation. In the case of textual languages, you define its representation using a grammar.



```
Expression : PlusOrMinus ;

PlusOrMinus returns Expression:
    MulOrDiv (
        ({Plus.left=current} '+' | {Minus.left=current} '-')
        right=MulOrDiv
    )*;

MulOrDiv returns Expression:
    NumberLiteral (
        ({MulOrDiv.left=current} op=('*' | '/'))
        right=NumberLiteral
    )*;

NumberLiteral returns Expression:
    {NumberLiteral} value=INT
;
```

Figure 2: Grammar of the same calculator language

A grammar is defined using two types of rules: terminals and non-terminals. Terminals define language keywords or writing rules of symbols, while non-terminals define the structure of different rules. Non-terminals rules often define the representation of the abstract syntax concepts. In this example, we define operators and `NumberLiteral` using non-terminals rules while the value inside `NumberLiteral` is represented by the `INT` terminal rule. We use the **`returns Expression`** to explicits the parent-child relation defined in the meta-model.

Modern tools like ANTLR or xText can derive a parser from this grammar. The parser is the direct link between concrete and abstract syntax. Actually, the parser read the code written following the concrete syntax rules and create an Abstract Syntax Tree (AST). The AST is a structure that represents what was concretely written in the concrete syntax using abstract syntax concepts.

The semantics is the part that provides a meaning to our concepts. There are different ways to express the semantics of a language, through a compiler or an interpreter.

A compiler allows you to define a semantics by generating a program in another language (most of the time with a lower level of abstraction like C, LLVM or Assembly). With this method, you do not compute the program in your source language but create another program that will perform the same computations, this property is called translational semantics.

On the other hand, an interpreter will actually perform the computation by traversing the AST and report the result from the leaves of the tree to the root in the form of an operational semantics. There are multiple implementation patterns to perform such computations, those patterns will be detailed in the next section.

2.2 Existing implementation patterns of interpreters

As we have seen before, interpreters traverse the abstract syntax tree parsed from the code to compute it. To do so, multiple implementation patterns exist and differ from one to another for reason like modularity or reusability.

The first example is the Interpreter pattern [3]. This pattern consist on simply add the semantics inside the class through a method(See Figure 3). This often lead to the recursive call of the children node functions.

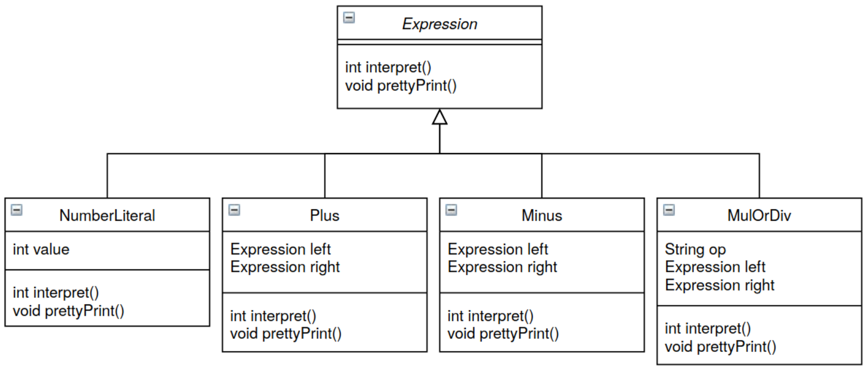


Figure 3: Interpreter pattern for the calculator language

The second pattern is the Visitor [3]. This pattern export the semantics in a `Visitor` class.

Each node implement an `accept` function that call the function containing it's semantics in the visitor passing itself as parameter(See Figure 4).

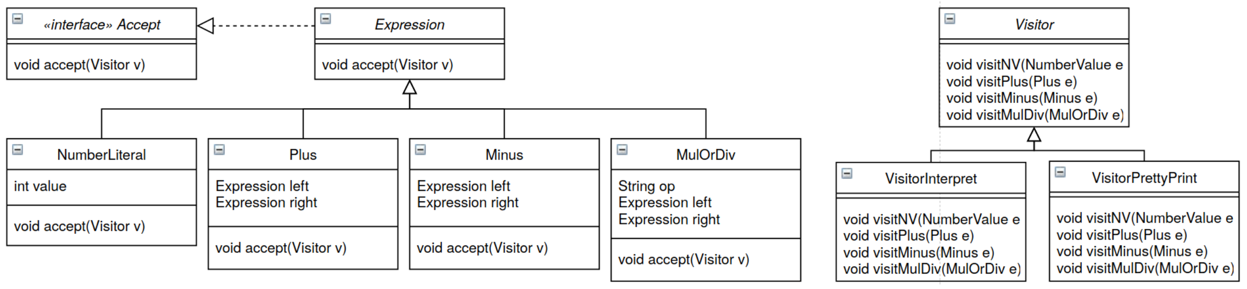


Figure 4: Visitor pattern for the calculator language

The EMF's Switch pattern is very similar to the Visitor pattern. The difference is that there is no `Accept` interface anymore. The concept is to do the dispatch in a generic function of the visitor resolving types and calling the appropriate method at runtime.

The previous patterns resolve the problem of semantics definition. However, a problem remains in the evolution of our languages. Patterns like Visitor or EMF's Switch allow an easy evolution of the AST processing but the modification of the abstract syntax requires the modification of all the visitors. On the other hand, the interpreter pattern is resilient to abstract syntax evolution but necessitate to modify all the classes of the abstract syntax. This problem is called in the litterature: The language extension problem [4]. To resolve this problem new patterns have emerged, one of them is called Object Algebras [5]. This pattern allows the designer to create algebras using interfaces that define the abstract syntax and by implementing those interfaces to provide a semantic. With this alone, we resolve the problem for the semantics extension, but this pattern also provides a way to combine algebras and their semantics to solve the problem of syntax extension.

More recently, other patterns appeared like the Revisitor pattern [6]. This pattern aims to keep the extension capacity of the Object Algebras but without changing the form of the abstract syntax. With this pattern, you can keep the compatibility with code generated from the meta-model using EMF's tools.

3 Performances and energy management in DSL interpreters

In this section, we explore existing approaches for performance and energy consumption optimizations. We decide to focus on those extra-functional properties because performance is an essential property and energy consumption in the IT world is nowadays a major environmental concern.

3.1 Performance optimizations of DSLs interpreters

We isolate three main axes in performance optimizations applicable at the language level. The first is the approximate computing, which reduces the quantity of computation by approximating it. The second is the multi-stage programming and how to apply it at runtime. Finally, the Just-In-Time compilation that is a widely used method to optimize interpreted languages.

3.1.1 Approximate computing

Approximate computing is a technic that relies on the fact that little modification on wisely selected parts of the program will create gains in efficiency while not affecting too much the accuracy of the result [7]. This technic can be used at different levels like code, computer architecture or hardware component. We only focus on code approximate computing because we want a generic solution without restricting to a precise machine.

Approximate computing is a relatively young paradigm that emerged from the need to process always more data with computation time and energy consumption in mind. There is multiple software-based approximate computing technics [8, 7], we only detail two of them that may be interesting in our case. The other are difficult to use because we lack of control over JVM memory management or because it's system dependant.

The first is the memoization, the principle is to memorize the output of a function for a given input and return the stored result during the next call of the function with the same input. For example, the first call to `fib(12)` will fail to retrieve the result in the saved results, calculate its value, store it and return the result. But, future calls to `fib(12)` will just result in accessing its value in the saved outputs. Memoization has the advantage to be easy to put in place because transforming a function to it's memoized version can be done automatically. Yet, this can't be done for function with side effects and come with an overhad cost, that's why a static or dynamic analysis will be mandatory to use it correctly. A tool named *MemoizeIt* was created with this precise goal in mind and will be further studied during the internship [9].

The second is loop perforation, this consists of skipping some iteration of a loop. Such a method can be implemented inside a loop unrolling technic to replace the potentially costly body of the loop by an interpolation of the neighbors' results [10]. By doing this we can achieve good results for for-loops that generate data in an array for example.

Approximate computing can be a good way to optimize simultaneously performances and energy consumption, but approximations of the execution can not always be fully appreciated by a dynamic system and need information from the designer to specify which part of the code can be approximate. A viable use of approximate computing in a DSL interpreter would be to use memoization when calling a function allowing a language level memoization rather than at code level.

3.1.2 Multi-stage programming

Multi-stage programming is a type of meta-programming with multiple compilation phases. This approach can be seen as a generative approach with multiple intermediate states and can be compared to a partial evaluation but at compilation time. However, a recent study proved that Just-In-Time compilers can be derived from an interpreter using multi-stage programming [11]. Moreover, this JIT compiler uses a multi-stage method to enhance the performance of compiled code. It's in this context we decide to study such technic.

Multi-stage programming has been used multiple times to design internal DSL, for instance, Lightweight Modular Staging(LMS) is a Scala Framework multi-staging approach [12]. This approach differs from previous by using types rather than a quasi-quotation system. By introducing a `Rep[T]` type you can explicitly specify in which stage your data will be used, This way you can define a DSL that will generate Scala code when changing stage. Moreover, you can use such approach to simply optimize your code leveraging on static data, as an illustration, when iterating over a matrix of boolean you can perform loop unrolling for the inner loop when the line is sparse and not when it's dense. Furthermore, multiple performance-oriented DSL compilers were created from LMS and exploit staging to abstract performance concerns from the DSL designer [13, 14].

Multi-staging relies on multiple code generation phases, such a tower of generators can create great variation between top and bottom semantics making optimization difficult or unrelated to DSL(top-level) semantic. N.Amin and T.Rompf present in their paper a way to collapse this tower [15]. Using this you can get one generator from top to the bottom of your tower. Moreover, Rompf et al. also proved that we can derive an optimizing JIT compiler from an interpreter definition [11]. By combining those methods to an optimized DSL tool like Delite or StagedSAC [13, 14] we can get a parallelizing and optimized JIT compiler.

In conclusion, multi-staging can be used to derive an optimizing JIT compiler and a possible area of research is how to add dynamic information like previous execution traces to this JIT compiler to enhance its optimization capability and introduce energy consumption concerns in this JIT compiler.

3.1.3 Just-In-Time Compilers

Following the observation of the slowness of language interpreters, Just-In-Time compilation technics have appeared to address this problem. Various compilation methods were used to find a trade-off between the cost of compilation and the benefits of the compiled code associated. One of the most known JIT compilers in the scientific literature is the one of the SELF language that is always the reference in this domain.

But, what is a JIT compiler and which concepts are embedded in it? Basically, a JIT compiler is a fast compiler called at runtime to compile parts of the code that are frequently used to get better performances. Arnold et al. define three types of program representations: High-Level Representation(HLR), Directly Interpretable Representation(DIR) and Directly Executable Representation(DER). A Java or C++ source code is an HLR, while Java bytecode and binaries are respectively DIR and DER. A JIT compiler is a fast optimizing compiler from DIR to DER [16].

One of the first attempts to implement a JIT compiler was done for Fortran [17]. The idea was to count the number of execution of a given block of code and when it reaches a threshold apply optimizations. Later, an implementation using compilation to pseudo-instructions was developed but was not outstanding in terms of performances comparing to the previous version [17].

As stated before SELF language JIT compilers represent a breakthrough in this field. Their JIT compilers can be clustered in three generation [17]. The first generation was based on the customization of methods. SELF being a dynamically typed language, much more information was available at runtime, the keystone of this generation was to compile to native code over-customized versions of the function for a particular context. Using context information, the JIT compiler was able to compile only the useful subset of the semantics to make it performant. The second generation added a better computation of types for loops and a probabilistic approach by deleting code for cases that are unlikely to occur(Arithmetic overflow, ...) providing a recovery solution if such a case appears. This probabilistic approach is a key concept of modern JIT compilers that over-simplify the compiled function keeping guards to rollback to the full and interpreted version of the function. Finally, the third generation try to solve the problem of JIT compiler slowness introduced by the new approaches and come with an interesting comment:

"...in the course of our experiments we discovered that the trigger mechanism ("when") is much less important for good recompilation results than the selection mechanism ("what")." — Hölzle

That implies that the block whose counter triggered the compilation is not always the one to compile, sometimes it's better to inline a function at the call site and compile a "caller" function.

Modern JIT compilers like Java's one are mainly based on the results of the SELF JIT compiler, for this reason, a translation from Java bytecode to SELF has been tried to leverage on SELF's optimisations [17]. Other approaches like annotating bytecode with information dedicated to the JIT system has also been tried.

We will now present two approaches to design or tune JIT compilers in the use case of language interpreters. Truffle and PyPy are two projects that aim to ease the development of new efficient programming languages, respectively in Python and Java. First, we will present the approach of the Truffle Framework[18] based on AST Typing and partial evaluation, then we will present the approach of PyPy relying on AST typing and meta-tracing.

AST specialization and partial evaluation with Truffle

Truffle is a framework developed by Oracle to define self-optimizing AST interpreter over GraalVM [19]. The main goal when Oracle created GraalVM was to amortize the development costs of an efficient VM by providing an easy way to implement new languages. To do this they rely on AST interpreters defined in the VM main language: Java. By factorizing common parts in VM development like JIT compilers, automatic memory management or well-defined memory model, they let language designers with only guest languages specific concerns to implement.

Truffle comes to boost guest language performances by specializing the AST using node rewriting [18]. Truffle allows the language designers to detail the behavior of a node considering child nodes type. The interpretation is done through traversal of the tree in post-order, this way child nodes are computed before their parent and can be considered as parameters of the parent. Considering that we can consider the specialization of nodes as a method specialization over a parameter inheritance hierarchy. This method allows the JIT compiler to compile only a sub-set of the node operational semantic and execute faster.

Additionally, Truffle provides annotation dedicated to partial evaluation [20]. This part of the framework let the designer add information about what is unlikely to occur by creating `Assumption` objects or considering a field static for partial evaluation(@PEFinal annotation) even if it's not. This part is powerful because it allows you to specify the corner cases in your semantic and consider it as dead-code unless a guard fails. The difference between `Assumption` and PEFinal is that PEFinal fields are "considered" final in the interpreter but are evicted from the JIT-compiled code by constant folding, meaning invalidation is performed inside the compiled code using a call to transfer to the interpreter while `Assumption` can be invalidated from another method and the compiled code will not be called next time but directly the interpreter.

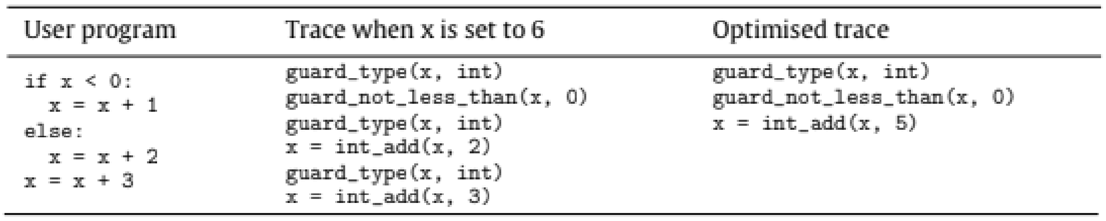
However, Truffle is a powerful framework but needs a lot of expertise from the language designer to achieve a performant interpreter. An attempt to generate automatically Truffle annotations from a language definition demontrate better performance than the classic interpreter, but only a few of those annotations can be generated now [21]. This is due to the complex to generalize properties for all programs of the resulting language.

Meta-tracing with PyPy

PyPy includes a meta-tracing technics to produce optimized JIT-compiled code [22]. The approach consists in tracing the interpreter(written in a statically typed version of Python called

RPython [23]) and optimize it rather than the program(or it's AST). The interpreter is instrumented to be able to provide trace for code that we might need to compile, for instance, a loop body. At the execution, the trace is created and if the code needs to be compiled we will not look at the code to generate the binary but the trace. The main goal when we use the trace is to be able to make assumptions on, generally speaking, the useless portion of code. The advantage of the trace is the fact that it contains only already used path in the control flow, paths that never occur will be eliminated by default. Of course, like other JIT assumption, the compiler will put guards for the corner-case omitted.

PyPy also uses technics from partial evaluation, the promotion [24]. The goal of the promotion is to look at precedent executions and if an input is often given separate the path in two, one with this parameter as a constant allowing constant folding and other classical optimizations, and the other with the full semantic to keep a fallback solution.



User program	Trace when x is set to 6	Optimised trace
`if x < 0:`	`guard_type(x, int)`	`guard_type(x, int)`
`  x = x + 1`	`guard_not_less_than(x, 0)`	`guard_not_less_than(x, 0)`
`else:`	`guard_type(x, int)`	`x = int_add(x, 5)`
`  x = x + 2`	`x = int_add(x, 2)`	
`x = x + 3`	`guard_type(x, int)`	
	`x = int_add(x, 3)`	

Figure 5: Example of program and traces from [25]

When compared to other languages [25], languages based on PyPy perform rather well. LuaJIT is a VM-based language with a JIT also, but perform slightly better. It's probably due to the fine-tuning of the JIT compiler and the smaller size of the language.

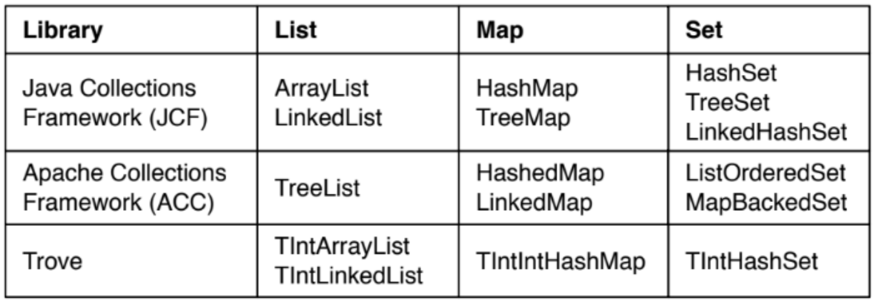
3.2 Energy consumption

Nowadays, energy consumption in IT represents a major problem because it grows exponentially. In this part, we will review some comparative studies of energy consumption to see if performance and energy consumption are opposed. Moreover, we will search for sources of energy over-consumption to see if we can manage those at a language level.

One fact that comes in mind when we talk about energy consumption and performance is that they are opposed, meaning a program made to have low energy consumption will be slower. A comparative study of programming languages energy consumption, memory usage and performance show quite the opposit [26]. This study concludes that energy consumption is more related to computing time than memory usage. For example, in their benchmarks, Java is 5^{th} over 27 programming languages in energy consumption and time while being 22^{nd} in terms of memory usage. In contrast, Python is 12^{th} in memory usage but 26^{th} in terms of performance and energy consumption. Those results need to be taken carefully, only 13 programs were used to do this study and no program have heavy I/O on disk and less computation, such program will raise the energy consumption due to memory usage. In the meantime, another study tries to show the impact of compression ratio on the energy consumption [27]. This study shows exactly the missing case of the benchmark suite, the idea is to see if a better compression will reduce the energy consumption of I/O. The result is that it is not clear, a better compression ratio needs more computation but produces less I/O, but for some compression algorithms, the computational cost grows so much that it erases the gain in I/O while others are non-monotonic and get optimization of overall consumption between two ratios. For instance, gzip with the CFQ I/O scheduler has better performance with a compression ratio set to 8 than a ratio set to 6,7 or 9. These two conclusions show the complexity to define a power model at a language level.

As we have seen there are two major sources of energy consumption: computation and memory access. Reducing power consumption from computation comes back to minimize the computation and we do that when optimizing performances. Therefore, we will focus now on the optimization of memory usage for the sake of energy consumption.

In this context, we read up on Java collections energy consumption [28]. In this article, the authors compare different implementations of java collections included in the Java Collections Framework(JCF), Apache Commons Collections(ACC) and Trove. The choice of Trove is motivated by the reduced space needed in comparison to classical java implementation(33% for large data structure). The panel of collection studied is the following :



Library	**List**	**Map**	**Set**
Java Collections Framework (JCF)	ArrayList LinkedList	HashMap TreeMap	HashSet TreeSet LinkedHashSet
Apache Collections Framework (ACC)	TreeList	HashedMap LinkedMap	ListOrderedSet MapBackedSet
Trove	TIntArrayList TIntLinkedList	TIntIntHashMap	TIntHashSet

Figure 6: Table of the studied collections

Concerning List implementation, LinkedList and ArrayList perform the same as for performance, .i.e inserting elements at the beginning and end of the list consume less energy if you use LinkedList implementations while ArrayLists perform better at inserting elements in the middle and accessing at a random position. For the difference between JCF and Trove, they perform similarly when used in a good way but Trove implementations consume more when not well used(inserting in the middle of a LinkedList). The TreeList is proved quite bad at insertion but performs as well as ArrayList in random position access. Finally, iterate over the structure cause the same consumption for all the List implementations. Those results are comforting because they correspond to the good practices for performance.

For the Map implementations, iterate over them or compute random queries result in the same power models. However, insertion is costly for JCF TreeMap when others perform quite the same. So there is no "best" implementation for Map. Similarly Set has no "best" implementation and like for Map, JCF TreeSet performs badly at insertion in comparison to other implementation. JCF LinkedSet is oddly also bad at inserting new elements when the set size is inferior to 1500 elements.

In conclusion, following good practices for performance is not harmfull to energy consumption. However, a surprising result is the equivalence of JCF and Trove implementations knowing the difference in the memory space used. We may assume that Trove must do more computation to be able to keep a small use of memory space. In our context, one idea would be to abstract the implementation of used collection to use the most fit for the task considering energy consumption. Unfortunatelly we can't really conclude of a good impact of a compact representation of collection due to equivalent results of Trove and JCF.

4 Architecture for dynamically adaptable systems

In this section, we will review dynamic adaptive systems as a base for our new dynamic pattern of interpreters. We decided to not study feedforward control because it doesn't adapt to disturbance(for instance invalidation of JIT-compiled code) and only offers as benefits the non-deterioration of system classical stability over feedback control, which is not relevant in our case because the system that we will monitor is unstable. We will also study self-healing software technics as a starting point of a self-healing Just-In-Time compiler.

4.1 Feedback Control loops

Feedback control is a common way to create Dynamic Adaptive System(DAS), it works using a control loop that continuously compares the output of the system's monitoring to the reference and applies changes to the application to make this output converge to the reference [29]. The objectives of such systems can be either a regulatory control, disturbance rejection or optimization, in our case we are interested in the last. In feedback control systems, multiple properties can be considered, but those are keystones for proper use.

First, there is the *stability* of the system. This property can be defined differently but as an example, we will take the BIBO(Bounded Input Bounded Output) stability that is used in [29]. If we state that the input and output are bounded and that our control module doesn't create periodic patterns in the output, the system will certainly converge to a steady-state. This property is fundamental for the design of efficient optimizations because an unstable system cannot be optimized correctly.

A second property is the *accuracy*, meaning, in this case, the convergence capacity of the system. If the system is stable, then it will reach a steady-state. Usually, it is not *accuracy* that is measured but we rather quantify imprecision using the difference between the system's steady-state monitoring output and the reference. Achieve good accuracy will be our main goal because it will directly influence the quality of the optimization performed.

The third property called *settling time* correspond to the speed of convergence, a short settling time implies a quick convergence. This property will also be studied keeping in mind what was stated before, what to optimize is more important than when.

The last property is the not-overshooting property. This property assures that if the conditions stated in the control module are valid the system will not do anything more. This property seems not relevant in our case since we want to reach optimality. To be able to optimize our interpreter depending on different extra-functional properties, generic architectural patterns like MAPE-K [30] bring necessary modularity to implement optimization concerns in separated modules keeping our future pattern agnostic of the optimization the language designer want to perform.

4.2 Self-healing software

Jointly, we search on the subject of self-healing software to apply it on extra-functional properties concerns like others before us [31].

There are two main currents [32], the first is the "behavioral repair" which changes the code to add a patch. The second is based on the modification of runtime state, in this current, we can find, for instance, self-healing data structures. Self-healing methods are generally implemented in two modules, one to monitor the program and check for failure and the second to perform the recovery.

This very similar to what we can find in MAPE-K(monitor module monitor and analyze and recovery module plan and execute) [30]. Such a model was further developed by E.Albassam et al. defining a generic model of self-healing using a MAPE-K loop for the adaptive part making calls to the recovery module to keep a separation of concerns between adaptation and recovery [33]. A more complete model of the component of a self-healing software detail a fault model, recovery model and define also other information necessary to well-perform the recovery like system completeness and design context [34]. We will now focus on "behavioral repair" because our use case is the convergence of JIT-compiled code behavior, consequently runtime state repair seems not relevant for this particular task.

One of the challenges when designing a self-healing system is to determine which recovery solution is acceptable and which is not. To do that, we need to provide oracles to assess the correctness of the proposed solution [35]. Those oracles can take multiple forms: tests, pre/post conditions, etc. Another challenge is the generalization of the solution. In fact, self-healing solutions can exploit specificities and weaknesses and fail to generalize the solution. This is a problem that we especially don't want since the major problem in JIT compiling is the invalidation of compiled code due to a too specialized solution. This overfit problem is will be one of the main concerns when developing the dynamically adaptive part of the interpreter.

5 Performance and enrgy consumption benchmarking

In this section, we will highlight the difficulty of reliable and reproducible benchmarking of extra-functional properties such as performance and energy consumption. After that, we will discuss two ways to manage the non-reproducibility of this task, upstream with tools to mitigate the non-deterministic behavior and downstream with a rigorous statistical approach.

5.1 The challenge of benchmarking

When we talk about benchmarking we generally talk about the measurement of execution time. However, this task and any measurement of extra-functional properties are very difficult to perform in a rigorous and reproducible way. In fact, various sources of non-determinism hamper the precise evaluation of such properties. External factors like CPU frequency or thread scheduling cause variations in the program execution. But those factors are not the only ones that matter, for instance, the initial state of the benchmark [36]. Kalibera et al. proved in their study that the variations in the initial state of the benchmark create an even bigger variation in the results than variations due to the environment itself. Moreover, other factors like UNIX environment size or linking order in program compilation can also create variations in the results [37].

In addition to the non-reproducibility due to the condition of the experiment, time counters can also be a source of non-reproducibility. For instance, the Java instruction `nanotime()` is a source of imprecision [38]. Indeed, the call to a time counter takes time depending on the computer workload and is not updated every nano-second, that make the nano-benchmarking a difficult task.

5.2 Mitigate the non-deterministic behavior

The first axis studied to create reproducible benchmarks is a rigorous control over the sources of results variations. In this state of the art, we focused on three tools developed to manage and measure extra-functional properties.

The first tool is named Krun [39]. This tool is interesting because it tries to reduce the impact of hardware and operating system non-deterministic behaviors. This tool comes in two parts, a Linux kernel and an application. The kernel provided is a custom Debian kernel that allows more control over the hardware and operating system, for instance, it will deactivate processor optimistic optimization, set the CPU frequency to a constant and deactivate all process not needed to do benchmarking. On the other hand, the application manages the initial state of all your experiments by restarting the computer between each benchmark, verify that the CPU temperature is the same and allow you to disable remaining services that are not useful for your benchmark.

Since this work will mainly focus on JVM-based interpreters, we studied other tools created to evaluate performances(JMH) and energy consumption(E-Surgeon) over JVM. Those tools help in the management of the JVM non-deterministic behavior like garbage collection or Just-In-Time compilation and in the measurements concerning fine-grained parts of java code.

The framework Java Microbenchmark Harness(JMH), was developed to perform precise microbenchmark of java code [40]. This framework provides java annotations to define the way of benchmarking the method and tooling to avoid dead-code elimination and constant folding in your benchmark. In addition to this framework, other tools were developed on top of JMH like AutoJMH [41] that abstract the good practices for dead-code elimination and constant folding by automatically generating a proper JMH benchmark for your functions.

The last tool is E-Surgeon, this tool is composed of two parts, a system library called PowerAPI and a java monitoring component called Jalen [42]. E-Surgeon was implemented to provide fine-grained energy consumption measurements. The system library, PowerAPI, can provide per PID component(CPU, Network card, ...) power consumption using pre-defined power models. Jalen, for his part, instrument the java code to extract methods information like start/end time and running state to process the energy consumption of each method using PowerAPI and dedicated powers models. To reduce the overhead of the monitoring, it offloads all the computations.

In addition to those tools, we could use other methods like replay compilation [43] to get reproducible garbage collector and Just-In-Time compiler calls or use input shaking to detect or suppress measurement bias [44].

5.3 Addressing bias with statistics

To do a rigorous analysis of benchmark results, we need to correctly design the experiment to be able to use a proper statistic methodology. But as we have seen benchmarking over JVM is challenging and numerous experience designs exist in the community, a comparative study [45] show what is done and tries to give proper methodologies. For instance, benchmarking an application start-up or steady-state is done differently. The first will be processed using multiple VM invocation but running only one iteration of the benchmark while the second will use fewer VM invocations but will iterate multiple times during a VM invocation to let the program reach a steady state. To evaluate an execution time of a program the method described uses multiple VM invocations to take into account the initial state of the benchmark [36] and either one or multiple iterations depending on what you want to evaluate. After that, the statistic method proposed is to compute the mean with confidence intervals assuming a normal distribution or a Student's t-distribution depending on the number of measures. Besides, they propose a method based on an ANalysis Of VAriance(ANOVA) to compare the results of two benchmark.

More recent papers [46, 47] propose new methods that provide better precision of the results using effect size. With this method, they propose a way to design an experiment to get good results in a reasonable time by reducing the number of measurements to the minimum needed. By complementing this method with technics like replay compilation [43], we can drastically reduce the number of measurements needed to get trustful results.

6 Conclusion

In conclusion, to dynamically optimize the performance and energy consumption of programming language interpreters, many solutions can be considered from this corpus. We plan to develop a dynamic interpreter implementation pattern based on a MAPE-K architecture. In the meantime, we will develop concrete performances and energy consumption enhancement technics based on the self-healing model. To achieve that we will use Truffle and trace the execution to enhance performances and energy consumption by fine-tuning the JVM JIT compiler.

The improvements offered by this pattern will be evaluated and quantified using a proper environment and rigorous statistical methods. We plan to use Krun[39] as a foundation and then instrument our code with JMH[40] for the performance and E-Surgeon[42] for energy consumption.

By achieving that we aim to answer the following question: Can we design a DSL interpreter as a dynamic adaptative system that leverages on domain-specific information and apply optimization at runtime for the sake of performance and energy consumption?

Further work would be to use the work of T.Rompf et al. to derive an optimizing JIT compiler[11] that leverage on Truffle information and meta-trace of the previous executions. By designing the JIT compiler in the form of the previous solution and using GraalVM capacity to create native-images to compile our interpreter and JIT compiler, we can leverage on Java code generated by Truffle at compile-time and keep the adaptive part of the JIT compiler. This solution is interesting because a compiled program runs faster and consumes less energy[26].

References

[1] B. Combemale, R. France, J. Jézéquel, B. Rumpe, J. Steel, and D. Vojtisek, *Engineering Modeling Languages: Turning Domain Knowledge into Tools*, ser. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series. CRC Press, 2016.

[2] Eclipse Foundation, “Eclipse Modeling Project | The Eclipse Foundation,” Jan 2020, [Online; accessed 21. Jan. 2020]. [Online]. Available: https://www.eclipse.org/modeling/emf

[3] E. Gamma, *Design patterns: elements of reusable object-oriented software.* Pearson Educ. India, 1995.

[4] M. Leduc, T. Degueule, E. Van Wyk, and B. Combemale, “The Software Language Extension Problem,” *Softw. Syst. Model.*, pp. 1–5, Dec 2019.

[5] B. C. d. S. Oliveira and W. R. Cook, “Extensibility for the masses,” in *European Conference on Object-Oriented Programming.* Springer, 2012, pp. 2–27.

[6] M. Leduc, T. Degueule, B. Combemale, T. Van Der Storm, and O. Barais, “Revisiting visitors for modular extension of executable dsmls,” in *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS).* IEEE, 2017, pp. 112–122.

[7] S. Mittal, “A survey of techniques for approximate computing,” *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2893356

[8] Q. Xu, T. Mytkowicz, and N. S. Kim, “Approximate computing: A survey,” *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb 2016.

[9] L. Della Toffola, M. Pradel, and T. R. Gross, "Performance problems you can fix: A dynamic analysis of memoization opportunities," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 607–622, 2015.

[10] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, "Approximate loop unrolling," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*. ACM, 2019.

[11] T. Rompf, A. K. Sujeeth, K. J. Brown, H. Lee, H. Chafi, and K. Olukotun, "Surgical precision jit compilers," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 41–52.

[12] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," *Communications of the ACM*, vol. 55, no. 6, 2012.

[13] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun, "Building-blocks for performance oriented dsls," *arXiv preprint arXiv:1109.0778*, 2011.

[14] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky, "Stagedsac: A case study in performance-oriented dsl development," in *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. ACM, 2012, pp. 73–82.

[15] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 52, 2017.

[16] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, 2005.

[17] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, 2003.

[18] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger, "A domain-specific language for building self-optimizing ast interpreters," *ACM SIGPLAN Notices*, vol. 50, no. 3, 2015.

[19] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 2013, pp. 187–204.

[20] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 662–676, 2017.

[21] M. Leduc, G. Jouneaux, G. Le Guernic, T. Degueule, B. Combemale, and O. Barais, "Automatic generation of truffle-based interpreters for domain-specific languages," 2020, Not published.

[22] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, "Tracing the meta-level: Pypy's tracing jit compiler," in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ACM, 2009, pp. 18–25.

[23] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, "Rpython: a step towards reconciling dynamically and statically typed oo languages," in *Proceedings of the 2007 symposium on Dynamic languages*, 2007.

[24] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo, "Runtime feedback in a meta-tracing jit for efficient dynamic languages," in *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ACM, 2011, p. 9.

[25] C. F. Bolz and L. Tratt, "The impact of meta-tracing on vm design and implementation," *Science of Computer Programming*, vol. 98, pp. 408–421, 2015.

[26] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: ACM, 2017, pp. 256–267.

[27] Z. Li, R. Grosu, P. Sehgal, S. A. Smolka, S. D. Stoller, and E. Zadok, “On the energy consumption and performance of systems software,” in *Proceedings of the 4th Annual International Conference on Systems and Storage.* ACM, 2011, p. 8.

[28] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, “Energy profiles of java collections classes,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[29] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback control of computing systems.* Wiley Online Library, 2004.

[30] E. Rutten, N. Marchand, and D. Simon, “Feedback control as mape-k loop in autonomic computing,” in *Software Engineering for Self-Adaptive Systems III. Assurances.* Springer, 2017, pp. 349–373.

[31] C. Le Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Commun. ACM*, 2019.

[32] M. Monperrus, “Software that learns from its own failures,” *arXiv preprint arXiv:1502.00821*, 2015.

[33] E. Albassam, H. Gomaa, and D. A. Menascé, “Model-based recovery connectors for self-adaptation and self-healing.” in *ICSOFT-EA*, 2016, pp. 79–90.

[34] P. Koopman, “Elements of the self-healing system problem space,” *ICSE WADS03*, 2003.

[35] M. Monperrus, “Automatic software repair: a bibliography,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[36] T. Kalibera, L. Bulej, and P. Tuma, “Benchmark precision and random initial state,” in *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, 2005, pp. 484–490.

[37] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Producing wrong data without doing anything obviously wrong!” *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1, 2009.

[38] A. Shipilev, “Nanotrusting the Nanotime,” 2014. [Online]. Available: https://shipilev.net/blog/2014/nanotrusting-nanotime

[39] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount, and L. Tratt, “Virtual machine warmup blows hot and cold,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 52, 2017.

[40] A. Shipilev, “Java microbenchmarks harness (the lesser of two evils),” 2013. [Online]. Available: https://www.youtube.com/watch?v=VaWgOCDBxYw

[41] M. Rodriguez-Cancio, B. Combemale, and B. Baudry, “Automatic microbenchmark generation to prevent dead code elimination and constant folding,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2016, pp. 132–143.

[42] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier, “Runtime monitoring of software energy hotspots,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012, pp. 160–169.

[43] A. Georges, L. Eeckhout, and D. Buytaert, “Java performance evaluation through rigorous replay compilation,” in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 367–384.

[44] D. Tsafrir, K. Ouaknine, and D. G. Feitelson, “Reducing performance evaluation sensitivity and variability by input shaking,” in *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems.* IEEE, 2007, pp. 231–237.

[45] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically rigorous java performance evaluation,” in *ACM SIGPLAN Notices*, vol. 42, no. 10. ACM, 2007, pp. 57–76.

[46] T. Kalibera and R. Jones, “Quantifying performance changes with effect size confidence intervals,” Citeseer, Tech. Rep., 2012.

[47] T. Kalibera and R. Jones, “Rigorous benchmarking in reasonable time,” *ACM SIGPLAN Notices*, vol. 48, no. 11, pp. 63–74, 2013.