Polyhedral Optimization For JavaScript: The Challenges

Manuel Selva
INRIA Camus, ICube lab., CNRS,
University of Strasbourg, France
manuel.selva@inria.fr

Julien Pagès
INRIA Camus, ICube lab., CNRS,
University of Strasbourg, France
julien.pages@umontreal.ca

Philippe Clauss
INRIA Camus, ICube lab., CNRS,
University of Strasbourg, France
philippe.clauss@inria.fr

Abstract

The JavaScript language was originally designed to help writing small scripts adding dynamism in web pages. It is now widely used, both on the server and client sides, also for programs requiring intensive computations. Some examples are video game engines and image processing applications. This work focuses on improving performance for this kind of programs. Because JavaScript is a dynamic language, a JavaScript program cannot be compiled efficiently to native code. For achieving good performance on such dynamic programs, the common implementation strategy is to have several layers handling the JavaScript code, starting from interpretation, up to aggressive just-in-time compilation. Nevertheless, all existing implementations execute JavaScript functions using a single thread. In this work we propose to use the polyhedral model in the just-in-time compilation layer to parallelize compute-intensive programs that include loop nests. We highlight what are the scientific challenges, resulting from the dynamism of the language, for integrating automatic polyhedral optimization. We then show how to solve these challenges in the JavaScriptCore implementation of Apple.

Keywords JavaScript, Engine, Automatic parallelization, Polyhedral Optimization, Just-in-time compilation.

1 Introduction

JavaScript is a high level, prototype-based, object-oriented, dynamic language. Strictly speaking, JavaScript is not the specification of the language itself, but the initial language and its implementation developed by Netscape. The standard name for the language is ECMAScript whose first version was released in June 1997 and the last one in June 2017. For simplicity purposes and to use the widespread "wrong" term as anywhere else, we will refer in the following to JavaScript instead of ECMAScript to mention the language itself.

IMPACT 2018 (http://impact.gforge.inria.fr/impact2018)
Eighth International Workshop on Polyhedral Compilation Techniques,
In conjunction with HiPEAC 2018.
January 23, 2018, Manchester, United Kingdom

Because JavaScript is a dynamic language, a JavaScript source program cannot be compiled efficiently to native code. Instead, JavaScript programs are handled by a JavaScript implementation referred to as JavaScript engine in the following. This engine is in charge of executing the JavaScript source program given as input.

JavaScript was used historically on the client web-browser side to enable dynamic web pages. A large number of applications are now using JavaScript also for running compute-intensive tasks such as image processing routines or video games engines. JavaScript is also now widely used as server side language. Because of its widespread usage, all the major internet companies and open source communities have their own JavaScript engine. Google has its V8 Engine [6], Apple has JavaScriptCore [2], Mozilla has SpiderMonkey [16] and Microsoft has Chakra [15]. For many years, these companies and open source communities have optimized their engines in the context of the so-called "browser war".

To efficiently execute JavaScript programs, all these engines use a layered approach often starting from interpretation of JavaScript source code and ending in aggressive just-in-time compilation to native code. Surprisingly, even if the engines themselves are parallel applications, none of them are able to execute JavaScript code in parallel. The sequential nature of the language itself is probably one explanation. In other words, because the language does not allow to express parallelism and because of the dynamism it provides, it is very challenging to identify and exploit parallelism in JavaScript applications.

Even if none of the widespread JavaScript engines mentioned above are able to execute JavaScript code in parallel, recent researches have started to study this question [9, 13, 14, 17]. Relying either on speculation, language extensions or on automatic loop parallelization, these proposals have shown that parallelism can be exploited in JavaScript benchmarks and real applications.

Independently of JavaScript, the polyhedral model [5] has proven to be very useful to optimize and parallelize compute intensive application kernels written in non dynamic languages such as C. More recently, polyhedral optimization has also been applied by just-in-time compilers [11, 12, 19]. In the latter case, the optimization

and parallelization are performed dynamically during the execution of the application.

Motivated by these first results regarding JavaScript parallelization and by the growing usage of polyhedral tools in just-in-time compilers, we study in this work the possibility of using the polyhedral model for automatic optimization and parallelization of JavaScript. As we show in the paper, the main challenges are related to the management of the dynamism allowed by the language. The growing usage of JavaScript for compute-intensive applications is a motivating indicator for the application of polyhedral optimization.

In this work, we make the following contributions:

- Identification of scientific challenges to integrate polyhedral optimization in JavaScript;
- Proposition of solutions for these challenges in the context of a state-of-the-art JavaScript engine;
- Demonstration of the benefits of doing polyhedral optimization on a matrix multiplication JavaScript kernel
- Identification of perspectives allowing to handle more JavaScript programs with the polyhedral model.

The rest of the paper is organized as follows. Section 2 describes the architecture of a layered JavaScript engine along with some basics regarding the polyhedral model. Section 3 presents the scientific challenges that must be handled to enable polyhedral optimization in a layered JavaScript engine. Section 4 proposes solutions for these challenges. Section 6 presents preliminary results on a matrix multiplication kernel. Finally, Sections 7 and 8 present related work and conclude this preliminary work.

2 Background And Objective

JavaScript is a very dynamic language. This dynamism has a strong impact on the way JavaScript programs are executed. The language allows to dynamically load piece of code during execution. This feature itself is hardly compatible with static compilers. Static compilation is also not an optimal solution because of the lack of information in the source code, e.g., no type information. For these reasons, JavaScript programs are executed by a JavaScript engine. This engine handles dynamism and can offer good performance by observing the execution of the program and then by optimizing it based on its observation.

Using the polyhedral model in the context of JavaScript implies to take into account this dynamism. This section gives an overview of what kind of dynamism is allowed by JavaScript, before describing how state-of-the-art JavaScript engines handle it. Finally, a brief introduction of the polyhedral model is given.

2.1 JavaScript Dynamism

To illustrate several forms of dynamism of JavaScript, let us consider the simple function shown in Figure 1.

```
f(img, width, height) {
  for (var i = 0; i < width; i++) {
    for (var j = 0; j < height; j++) {
      var v = img[i*width + j];
      v = v + 41;
      v = v * 7;
      img[i*width + j] = v;
    }
  }
}
```

Figure 1. Simple JavaScript function iterating over an image.

First of all, the language is dynamically typed. In our example, it means that neither the type of the function parameter nor the type of the property of this parameter, accessed through a numerical index, are known and they can change over time. As a consequence, to safely execute this function, the JavaScript engine must first look at each iteration where is located the property `i*width + j` of the `img` parameter. Then the engine must look what is the meaning of the `+` and `*` operators for the `v` variable, according to its type.

Another important concern for engines, revealed indirectly by this example, is JavaScript numbers. From the programmer point of view, the specification tells that numbers are all double precision floating point numbers. This has a strong impact on the performance of the engine that must implement such semantics. Nevertheless, JavaScript engines use cheap 32 bits integer instructions when programs manipulate small integer values. But because JavaScript numbers must behave as double precision floating point numbers, using 32 bits integer instructions is semantically correct only if the numbers fit in 32 bits. As a consequence in our example, considering that `img` contains only integers, the engine has to check that v fits in 32 bits when using the processor 32 bits integer instructions to perform the `+` and `*` operations.

The language is also very permissive regarding arrays. This is not shown in our example, but in JavaScript, it is possible to write outside the bounds of an array. For example, a function scaling up an image could have statements writing outside the image. In this case, the semantics of the language is to extend the array up to the index that has been written. Slots in the array between the previous last element and the one just written are then considered as holes.

2.2 Layered JavaScript Engine

For efficiently handling all forms of dynamism present in the language, all JavaScript engines rely on a layered architecture [6, 15, 16]. Figure 2 depicts the architecture of JavaScriptCore [2], the engine that we use in this work. JavaScriptCore is a-state-of-the-art JavaScript engine developed by Apple and used in the WebKit project. Webkit being itself used, among other, by the Safari web browser. Depending on the engine itself, some of the layers depicted on this figure may not be present, but the global architecture is the same for all the JavaScript engines in use. These engines work on a function basis. Each time a function is called, it is executed by a given layer depending on the context. The main idea is to execute the time consuming functions in the most efficient layers, in which the engine can afford to spend time for optimizing.

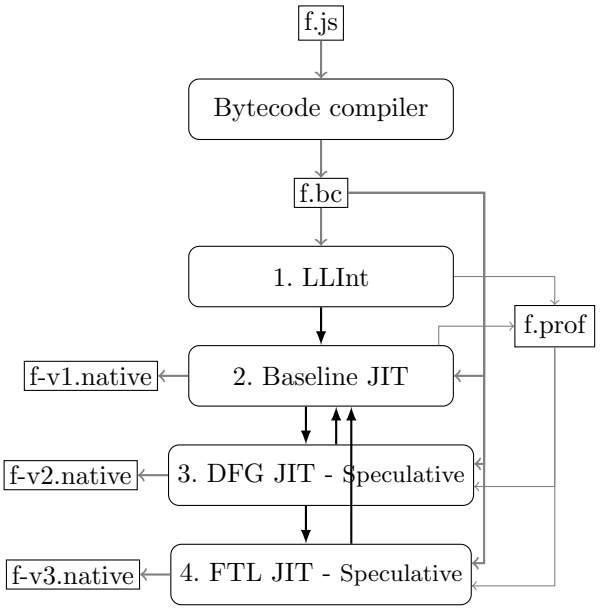

Figure 2. The layered architecture of JavaScriptCore.

We now describe this layered architecture by showing how our example function of Figure 1 is handled, focusing on the + and ∗ operators. The first time the engine must execute a given function, it compiles it to its own bytecode representation. Then the function is handled by the first layer, an interpreter, whose source code can be summarized in a simplified way by the code depicted in Figure 3. As already mentioned, the interpreter must take care of the types of the operands and dispatch to the appropriate implementation.

Then, on some next invocation or inside the current invocation of the function, the engine may decide to switch to the next layer, labeled Baseline JIT in Figure 2. This layer compiles the function bytecode to native code. This

```
while(i = next_instruction()) {
  switch(i->opcode) {
    case add:
      switch (type_pair(i->operand1->type(),
                        i->operand2->type())):
        case integer_integer:
          i->dest = i->operand1->as_int() +
                    i->operand2->as_int();
        case object_integer:
          ...
    case mul:
      switch (type_pair(i->operand1->type(),
                        i->operand2->type())):
        ...
    case ...
  }
}
```

Figure 3. Extract of the source code of a typical JavaScript interpreter.

is done in a very naive way by replacing each bytecode instruction with the corresponding assembly sequence of the interpreter. Figure 4 shows the generated binary code for the two consecutive instructions that compute the new value of v in our example function. This first compilation step removes the overhead of dispatching instructions. In the generated code, compared to the interpreter, there is no more any switch on the type for the current instruction.

```
...
...
switch (type_pair(v, 41)):
  case integer_integer:
    v = v->as_int() + 41;        } op_add v 41 v;
  case object_integer:
    ...
switch (type_pair(v, 7)):
  case integer_integer:
    v = v->as_int() * 7;         } op_mul v 7 v;
  case object_integer:
    v = v->obj_to_int() * 7;
...
...
```

Figure 4. Native code generated naively for the + and ∗ instructions of the example function.

Then, again on some next invocation of the function or inside the current invocation of the function, the engine may decide to switch to the next layer called DFG

(DataFlow Graph) JIT. Compared to the interpreter and the Baseline JIT layers, the execution enters the speculative part of the JavaScript engine. In this layer, JavaScriptCore relies on assumptions made by looking at profiling information gathered by the previous layers. This profiling information contains for example the effective types that have been encountered up to now. Considering that our function has been only called with arrays of integers fitting on 32 bits, JavaScriptCore creates a new representation of the program taking into account this information, as shown in Figure 5. In the case of speculation failing, JavaScriptCore must go back to the last non speculative layer in order to correctly execute the function according to the JavaScript semantics. Starting from this new representation, that does no more contain any instruction devoted to handling the dynamism, typical compiler optimization may be performed, before generating new native code far more efficient than the one generated by the naive Baseline JIT layer.

```
f(img, width, height) {
  if (img is array of 32 bits integers) {
    for (var i = 0; i < width; i++) {
      for (var j = 0; j < height; j++) {
        var v = img[i*width + j];
        v = v->as_int() + 41;
        v = v->as_int() * 7;
        img[i*width + j] = v;
      }
    }
  }
  else {
    return to Baseline JIT;
  }
}
```

Figure 5. Internal representation of the DFG JIT speculative layer for the example function.

Finally, the FTL JIT layer which is also speculative, consist in applying more aggressive and thus more time consuming transformations. To that end, JavaScriptCore compiles the dataflow graph intermediate representation to LLVM-IR and then to native code. Using LLVM allows to leverage most of its transformations and its backend.

The general idea of this design is to remove as much dynamism as possible by profiling the code behavior. Final layers can then apply aggressive optimization since they do not need to handle dynamism. In case of bad predictions, the execution rollbacks to the first layers.

Existing JavaScript engines are not able to execute JavaScript code in parallel. Said differently, the interpreter code and the different versions of native code generated dynamically are all sequential. Nevertheless, the just-in-time compilation process itself is often done in parallel of the execution in the previous layer. Other tasks of the JavaScript engine, e.g., garbage collection, are also done in parallel in existing engines.

2.3 The Polyhedral Model

The polyhedral model [5] is a mathematical model devoted to the analysis and transformation of loop nests. In order to be optimized by the polyhedral model, a loop nest must be compliant with what is called a Static Control Part (SCoP). A SCoP is a loop nest where loop bounds, memory accesses and branches conditions are all affine functions of parameters constant in the nest and of enclosing loop iterators. Based on this model, classical loop transformations such as skewing, interchange and others can be expressed in a common simple formalism.

Historically, polyhedral tools were implemented as source-to-source compilers. More recently, polyhedral optimization have been implemented at the level of compilers intermediate representations. It has been successfully deployed in production compilers such as GCC and LLVM respectively by the GRAPHITE [20] and the Polly [7] frameworks. The main challenges for performing optimization on intermediate representation are the identification of SCoPs and the granularity choice of what would be considered as a schedule unit by the polyhedral optimizer, i.e., a statement in the polyhedral terminology. In this work, we do not address these challenges but rely on existing proposals and tools. The addressed challenges are related to the objective of having the JavaScript engine last optimization layers generating code which can be handled by polyhedral tools.

2.4 Objective

The final objective of this work is to integrate polyhedral optimizations in JavaScript engines. This can only be done in the last speculative layer of the engine, when the dynamism has been entirely removed, such that the code is in the closest shape to what can be handled by polyhedral tools. To reach this goal, we present in this paper what are the challenges to merge both the world of advanced static optimization and the world of just-in-time compilation for a dynamic language such as JavaScript.

3 Challenges For Polyhedral Optimization Of JavaScript Programs

This section brings to light the challenges for integrating polyhedral optimization inside a JavaScript engine to generate native code that is more efficient than the one generated by state-of-the-art engines. Additional

performance comes from parallelization and data locality optimization provided by the polyhedral model. In Section 4, we propose solutions to these challenges in the context of the JavaScriptCore engine developed by Apple.

3.1 Issue 1: SCoPs Detection

As stated in Section 2.3, the first challenge of polyhedral tools working on intermediate representations is to identify SCoPs. In the context of intermediate representation generated by a JavaScript engine, this detection is made even more difficult because of the shape of the code. This code is quite different from the one generated by front-ends for static languages such as C or C++.

3.1.1 Single Entry Single Exit Regions

Section 2.3 introduced the three constraints a loop nest must satisfied to be a SCoP. The constraint on the conditionals leads to the fact that the control flow of the loop must be statically known. In practice, polyhedral tools operating on intermediate representations add the constraint that the loop nest must form a Single Entry Single Exit (SESE) region. This means that all the basic blocks of the loop nest must be in a region of code with only one entry point and one exit point.

In the context of code generated by a JavaScript engine, this requirement is not met, because of the handling of speculation. For each speculation check, the engine inserts a jump back to the previous layer as shown previously in Figure 5. This jump is implemented in intermediate code representations by blocks jumping to a particular address of the runtime, which is responsible to follow up the execution in the previous layer.

Figure 6 shows the control flow graph generated by the JavaScriptCore engine for the innermost loop of Figure 1. This diagram clearly reveals a non-SESE nature. The four blocks on the left are jumps to the previous layer of the engine, which make this loop nest non conform to a SESE region. As stated in Section 2.2, these checks are required to ensure that the speculations made by the engine on past profiling results are still valid. In this particular example, the checks ensure that the code does not perform an access outside the bounds of the array, that it accesses an element of the array that has been already allocated, i.e., not a hole, and that the result of the integer operations fit on 32 bits. This 32 bits size has been chosen by the engine when generating native code, because all the profiled values were fitting on 32 bits.

3.1.2 Detection Of Affine Accesses To Arrays

As described in the previous sections, JavaScript arrays are complex objects. They can be extended and are not typed. One array can store various types of data

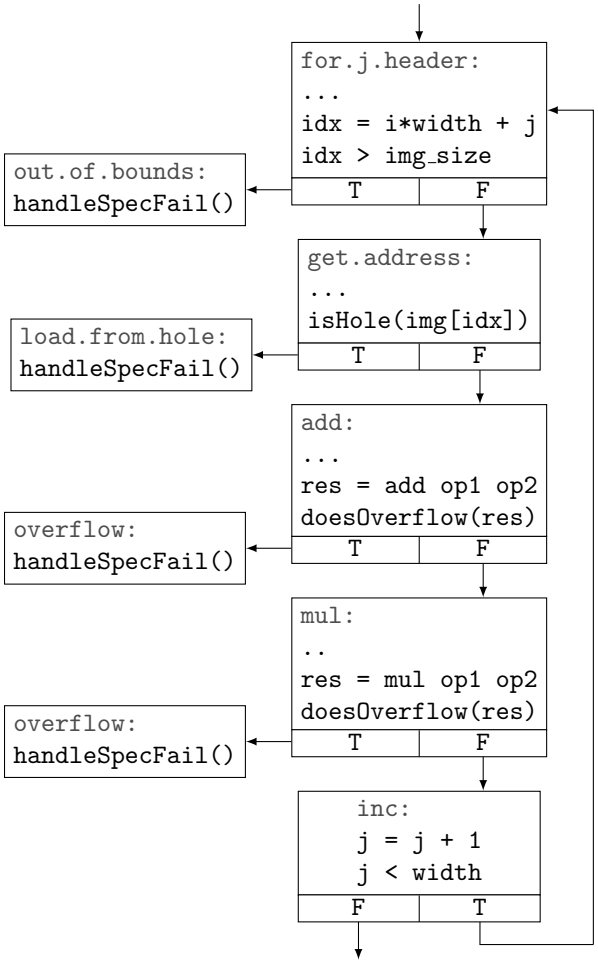

Figure 6. Engines speculation leads to non-SESE regions in the control flow graph of the intermediate representation.

simultaneously in its cells. Nevertheless, for an array of primitive types without holes, JavaScript engines will use contiguous memory.

The successful detection of affine accesses to these contiguous arrays of primitive types strongly depends on the structure of the code that is generated by the JavaScript engine. This structure must comply with a code shape that can be successfully parsed by polyhedral tools. Section 4.1.2 will show why code generated by JavaScriptCore does not enable detection of affine accesses.

3.1.3 Two Dimensional Arrays And Arrays Of Objects

Two-dimensional arrays do not exist in JavaScript. To declare a two-dimensional array, the programmer has to create a first one-dimensional array, and then a second one in each cell of the first array. Because of that lack of

two dimensional arrays in the language, JavaScript engines cannot store two-dimensional arrays in contiguous memory. Also, because of the nature of JavaScript arrays that can contain different types of elements, arrays of objects are actually implemented as arrays of pointers. As a consequence, the following expressions both imply two memory accesses:

- `t_ints[i][j]=17;`
- `t_objs[i].foo=17;`

In both cases, the first memory access is a load from an array. In the first case, the second access is also a load from an array while in the second case it is a load from a property by its name `foo`.

These two loads in generated code, prevent polyhedral tools to represent JavaScript loop nests iterating over such arrays. Indeed, by analyzing the code, it is impossible to know whether the successive loads of the `i` integer property, or of the `foo` named property, are affine functions of loop iterators and constant parameters. The target locations for these memory accesses depend on the location of the objects refereed to by arrays `t_ints` and `t_objs`.

3.1.4 Alias Analysis

As it is for many compiler transformations, polyhedral optimization requires precise information about pointer aliasing. For a given transformation to be safe, the optimizer must ensure that arrays accessed by different names are actually different arrays. In our example function, there is no such alias issue. Such problems typically occur, as it is the case for compilers of C programs, for example in a matrix multiplication function defined as shown in Figure 7.

```
matmul(left, right, res, left_nblines,
       left_nbcols, right_nbcols) {
  for (var i=0; i<left_nblines; i++) {
    for (var j=0; j<left_nbcols; j++) {
      var idx_left = i * left_nbcols + j;
      for (var k=0; k<right_nbcols; k++) {
        var idx_res = i*right_nbcols + k;
        var idx_right = j*right_nbcols + k;
        res[idx_res] = res[idx_res] +
                       left[idx_left] *
                       right[idx_right];
      }
    }
  }
}
```

Figure 7. Matrix multiplication function leads to alias analysis issues. The compiler cannot know whether or not the res matrix will alias with the left or with the right one.

3.2 Issue 2: Parallel Speculation Failure

After an automatic optimization and parallelization of the JavaScript code by a polyhedral optimizer has been performed, several threads run in parallel a single loop nest. Because the engine applies polyhedral optimization in its last layers, optimization is done on speculative code. As a consequence, the parallel code generated includes jumps back to the previous layer in case of speculation failure.

At runtime, if such a jump is triggered in one of the threads executing the loop nest, the current state of the system may be wrong. In a sequential execution, the speculation check is always performed before executing the code relying on it. Nothing can go wrong and the dynamism that appears again in the code is handled by the previous layer. In case of parallel execution, several threads may already have performed some wrong computations when a particular thread encounters a speculation failure. All threads must then be stopped and the execution is potentially incorrect. The jump must be handled properly and the loop has to be restarted from the beginning in sequential mode where speculation failure will be properly handled.

3.3 Issue 3: Gain Versus Overhead

Considering that we are able to perform polyhedral optimization of JavaScript programs inside a JavaScript engine by solving issues 1 and 2 described above, the last challenge is to ensure that it is worth to do so. As for any runtime optimization, the time spent in performing the optimization must be counterbalanced by the reduction of execution time provided by the optimized version of the code.

In the context of polyhedral optimization, this gain versus time overhead dilemma is directly related to the performance of tools implementing the optimization. Even if these tools have an exponential complexity in the number of statements of the target loop nest, it has also been shown [11, 19] that they can still be used successfully at runtime. So the question to answer is whether the time required by polyhedral optimization is acceptable in the context of JavaScript engine. This question leads to the question of the configuration of the polyhedral tools used inside a JavaScript engine that may strongly impact optimization time.

4 Solutions Proposals

We now propose solutions to the challenges described in the previous section. Our proposal is to integrate polyhedral optimization in the last layer of JavaScriptCore, the FTL JIT. At this stage, all JavaScript dynamism

has been removed. Also, as described in Section 4.3, integrating polyhedral optimization in the last layer helps in answering issue 3 about gain versus overhead. Moreover, because this step relies on LLVM[1], our engine can leverage its mature polyhedral framework called Polly [7]. Polly first builds a polyhedral representation of the LLVM-IR. Then some polyhedral transformations are performed on the polyhedral representation. Finally a new version of the LLVM-IR is generated back from the optimized polyhedral representation.

4.1 Solution To Issue 1 : SCoPs Detection

As stated in Section 3.1, the LLVM code of a loop nest generated by JavaScriptCore must be in a SESE region and must form a SCoP to be optimized with polyhedral tools such as Polly.

4.1.1 SESE Regions

The solution to this problem is related to the solution of parallel speculation failure described in Section 4.2. The main idea is to remove all terminal blocks that jump back in the previous layer leading to non SESE regions. We can do this because if such a block is executed during the parallel execution, we need to re-execute again the whole loop. As a consequence, the generated parallel code is semantically correct only if no such terminal block is executed.

More precisely, our solution proposal is as follows:

1. Remove all terminal blocks that jump back in the previous layer;
2. Insert metadata information for each instruction that can trigger a jump back in the previous layer, along with information about the condition of the jump. These instructions are the branching instructions at the bottom of each basic block shown in Figure 6;
3. Apply Polly transformations on this simplified SESE version of the code;
4. Using the metadata information, insert back after Polly transformations the blocks jumping back to the previous layer. Section 4.2 presents the detailed content of these blocks.

This solution allows to apply Polly optimization while still properly detecting engines speculation failures.

4.1.2 Detection Of Affine Accesses To Arrays

In JavaScriptCore, for an access such as `t[index]=17;` where `t` is an array of numbers being all 32 bits integers, JavaScriptCore originally generated the code in Figure 8. First, the `offset` into the array is computed. The index is multiplied by the size of one element in the array. This size is always 64 bits even for an array of 32 bits integers. This is due to the way the engine internally represents objects and primitive types through a technique called NaN boxing[2]. The second step is to add the integer value of the pointer on the array's base and this offset. Since LLVM is a typed IR, the conversion from integer to pointer is done by the `inttoptr` instruction. Finally the store is performed. All these steps are complex from a compiler point of view and are hard to track for a tool like Polly.

```
%offset = shl i64 %index, 3
%cell_as_int = add i64 %base_as_int, %offset
%cell_ptr = inttoptr i64 %cell_as_int to i64*
store i64 %boxed_17, i64* %cell_ptr
```

Figure 8. Original LLVM-IR generated by JavaScriptCore for a write in an array of integers.

To expose array accesses in a way handled by Polly, our engine replaces these instructions performing pointers arithmetic by the `getelementptr` instruction that takes a variable number of parameters. The first one is the type of the array that allows LLVM to know the size of each element with optionally the number of elements. The second one is the accessed array, i.e., a pointer whose type must be conformed to the type described by the first parameter. The following parameters, whose number depends on the number of dimensions of the array, indicate which element is targeted.

There is no conceptual difference between the `getelementptr` instruction and manual computation of an address with pointer arithmetic. Nevertheless, the first method allows to have all information in only one instruction and simplifies the work for polyhedral tools.

Figure 9 presents the modified LLVM-IR that is equivalent to the first form but that allows Polly to perform its analysis. The first step is to convert the integer value of the pointer to a real LLVM pointer. The variable `base_ptr` is now a pointer to the array's base. The following `getelementptr` instruction internally performs pointer arithmetic to get the cell pointer[3].

[1]Starting from version 2.12, JavaScriptCore is no more using LLVM but a custom low level intermediate representation along with a custom backend called B3. The solutions proposed in this section are nevertheless all applicable to this custom representation excepted the one for the issue of detecting affine accesses to arrays.

[2]http://www.redditmirror.cc/cache/websites/blog.mozilla.com_cwn0q/blog.mozilla.com/rob-sayre/2010/08/02/mozillas-new-javascript-value-representation/index.html

[3]See the LLVM documentation to understand why there is an additional 0 parameter before the index one https://llvm.org/docs/GetElementPtr.html#what-is-the-first-index-of-the-gep-instruction.

```
%base_ptr = inttoptr i64 %base_as_int
                to [1000 x i64]*
%cell_ptr = getelementptr [1000 x i64],
              [1000 x i64]* %base_ptr,
              i32 0,
              i32 %index
store i64 %value, i64* %cell_ptr
```

Figure 9. Enhanced LLVM-IR for a write in an array of integers allowing Polly to compute affine functions.

4.1.3 Two Dimensional Arrays And Arrays Of Objects

We currently do not support optimization of loop nests including accesses to two dimensional arrays and arrays of objects. We focus on single dimension arrays of primitive types. Regarding two dimensional arrays of primitive types, this is not a strong concern because JavaScript programmers are used to avoid them and polyhedral tools are capable of recovering dimensions [8, 11]. JavaScript programmers already linearize arrays because JavaScript engines are far more efficient with single dimension arrays leading to a single memory load compared to multi dimensional ones as described in Section 3.1.3. Our example function in Figure 1 and the `matmul` function in Figure 7 show examples of two dimensional arrays that have been linearized by the programmer.

A possible solution to handle two dimensional arrays and arrays of objects would be to modify the memory allocator to force them to be contiguous. This would require either a new construct in the language, or new analyses during profiling ensuring that the engine can reallocate the array in a contiguous way. That would also imply to modify the garbage collector to maintain this property even if the arrays are copied in another location of the heap.

An other possible solution consists in studying how static analyses developed for other languages [21] without native multi dimensional arrays could be applied to JavaScript.

4.1.4 Alias Analysis

In the context of JavaScriptCore, the alias problem is already mitigated by custom analyses. The main idea already implemented in the engine is type-based analysis relying on the JavaScript objects type hierarchy. In the LLVM-IR code generated by the engine, the object oriented nature of JavaScript has been removed. This is a requirement, since LLVM-IR has no such high level concepts. Nevertheless, because LLVM is not only used to compile low level languages such as C but also object oriented languages, mainly C++, the LLVM-IR provides mechanisms to allow specifying high level typing information. This is done through a particular type of metadata information. This metadata specifies both a type hierarchy in the form of a tree and the type being accessed by each store and each load instruction. Based on this information, LLVM provides alias analyses that can ensure that two memory operations will not access the same address if they access different branches of the type tree.

The alias analysis problem is also mitigated by runtime solutions proposed recently [1] and implemented in Polly. The idea is to compute at compile time, required conditions ensuring that pointer based accesses will not alias. Two versions of the code guarded by a check implementing these conditions are then generated.

4.2 Solution To Issue 2 : Parallel Speculation Failure

To bypass this problem we are currently investigating two solutions. The first one proposed recently in [17] leverages so called idempotent regions, and the second one uses rollbacks and checkpoints.

An idempotent code region [3] is a region that can be interrupted in the middle of its execution and then re-executed from the beginning while still providing the same result. In other words, the region does not modify its inputs. A matrix multiplication producing a resulting array is an example of an idempotent region. Exploiting this property, a JavaScript polyhedral optimizer would only handle idempotent loops to be able to re-execute them with the correct sequential non speculative layer. This solution is simple and has only a small cost consisting in detecting idempotent regions. On the other hand, not all loops can be parallelized with this solution.

The rollback solution implies to make a checkpoint of the memory state before starting parallel execution of a loop. When a speculation failure is triggered, the memory state is first restored from the checkpoint. Then, as in the previous solution, the execution starts again with the correct sequential non speculative layer. In our polyhedral context, the cost of saving the memory state to create the checkpoint could be greatly reduced. Relying on proposals of speculative just-in-time polyhedral optimizers [10], we could exploit the affine memory accesses functions to only save the part of the memory which is known to be updated by the loop nest. The main difference with existing work is that the speculation does not concern the polyhedral nature of the loop, but the removal of JavaScript dynamism. Nevertheless, the solution of only saving what would be modified is the same.

For both solutions, when a speculation failure is triggered in one of the parallel thread, our engine must jump to a custom handler. This handler stops all threads and

re-executes the loop from the beginning using the non speculative sequential layer. For the rollback based solution, the custom handler must also restore the memory state.

4.3 Solution To Issue 3 : Gain Versus Overhead

Our proposal first relies on the layered architecture of the JavaScript engine to ensure that spending time in polyhedral optimization is acceptable. JavaScriptCore already has a configurable cost model, which is based on the number of bytecode instructions executed by a function and on the number of invocations. This model ensures that functions optimized by the FTL JIT layer are functions where the program spends most of its execution time.

Secondly, relying on Polly, our proposal also leverage another cost model. Polyhedral optimization, even in a static context, must not be applied blindly because in some cases it may hurt performance instead of improving it. As a consequence, Polly includes its own cost model which checks some conditions on the loop nest before optimizing it. Thus, Polly will not even try to optimize the LLVM-IR code generated by our JavaScript engine if it believes that it will not benefit from polyhedral optimization.

Last, integrating a state-of-the-art JavaScript engine such as JavaScriptCore allows us to benefit from the parallel compilation mechanisms that are already at work. In JavaScriptCore, the generation of the LLVM-IR and the compilation of this code representation to native code is already achieved by a parallel thread. As a consequence, the additional time spent by Polly to perform polyhedral optimization only delays the time when the optimized native code is ready for execution.

5 Prototype Implementation

Our implementation is a modified version of JavaScriptCore including the solutions proposed in Section 4 and including the run of Polly passes in the FTL JIT layer. The implementation regarding non SESE region and parallel speculation failure as described in Section 4.1.1 and in Section 4.2 is not yet completed. Nevertheless, as described in the next section, this baseline prototype allows us to assess the validity of our approach, at least on carefully chosen examples.

We also modified Polly for different purposes. First, we added the support of the `inttoptr` instruction that caused our `matmul` function to be rejected by Polly SCoP detection pass. This instruction is widely used by the engine to create LLVM-IR pointers from constant locations known by the FTL JIT compiler.

We also added the support of `sext` and `trunc` instructions in Polly arrays delinearization algorithm. These instructions, previously not handled by the algorithm, are widely used by JavaScriptCore. This is related to the NaN boxing trick used by the runtime to internally store JavaScript objects and primitive types. Both of them are represented by a 64 bits value where the first bits indicate the type. For 32 bits integers, a `trunc` instruction is required to remove these first bits to get the effective 32 bits value.

6 Experimental Results

To assess the validity of the proposed solutions, we now review preliminary results obtained with the matrix multiplication function shown in Figure 7. This function satisfies the requirement of our implementation handling only single dimension arrays of primitive types as stated in Section 4.1.3. The matrices provided as parameters are single dimension arrays which size is the number of elements in the matrix.

We ran the following experiments on a desktop machine with an Intel Xeon W3520 processor with four physical cores and hyperthreading disabled. The machine is running Linux 4.4.0 and we used LLVM and Polly version 4.0.0.

Thanks to these modifications both on the LLVM-IR generated by JavaScriptCore and on Polly, Polly is able to handle the loop of the matrix multiplication function. We run Polly with only the `--parallel` option. Figure 10 shows that after optimization, as reported in a pseudo code fashion by Polly, the matrix multiplication loop has been made parallel and tiled.

Regarding aliasing issues, Figure 10 shows that JavaScript type based alias analysis has not been effective because Polly generates a runtime test guarding the execution of the parallel version of the code. Because the three matrices provided as parameters to the `matmul` have the same type, i.e., arrays, it is impossible, looking at their type only, to ensure that they will not alias.

The content of the alias checks also reveals that Polly was able to recover the two dimensional nature of the memory accesses in the three matrices, internally called `Mem5`, `Mem6` and `Mem7` by Polly.

Because our prototype implementation is not yet completed regarding the handling of speculation failure as stated in Section 5, we must ensure that no such failure can happen in our evaluation. To that end, we use input matrices guarantying that the `matmul` function always write inside the bounds of the result matrix, that the result matrix does not contain any hole and that integer operations never overflow. To fairly evaluate our version of JavaScriptCore including polyhedral optimization where blocks handling speculation failure have been

```
if(
   (&Mem6[p0-1][p1] <= &Mem5[0][0]
    ||
    &Mem5[max(0,p2-1)][p1] <= &Mem6[0][0])
   ) &&
   (&Mem7[max(0,p2-1)][p0] <= &Mem5[0][0]
    ||
    &Mem5[max(0, p2-1)][p1] <= &Mem7[0][0])
){

 #pragma omp parallel for
 for(c0=0; c0<=floord(p2-1,32); c0+=1)
  for(c1=0; c1<=floord(p1-1,32); c1+=1)
   for(c2=0; c2<=floord(p0-1,32); c2+=1) {
    for(c3=0; c3<=min(31,p2-32*c0-1); c3+=1)
     for(c4=0; c4<=min(31,p1-32*c1-1); c4+=1)
      for(c5=0; c5<=min(31,p0-32*c2-1); c5+=1)
       Stmt_68(32*c0+c3, 32*c2+c5, 32*c1+c4);
}

else {
  original code version
}
```

Figure 10. Optimized code generated by Polly for the matrix multiply function in Figure 7. Tiling and parallelization have been performed and a runtime check is required for aliasing issues.

removed but not re-introduced, we created a modified version of the classic JavaScriptCore where these blocks are also removed.

Size of left matrix	Execution time without Polly (s)	Execution time with Polly (s)	Speedup
50x3000	0.08	0.06	1.33
500x3000	0.85	0.22	3.86
2000x3000	3.3	0.87	3.79

Figure 11. Speedups resulting from polyhedral optimization for the `matmul` function with different matrices size.

Our benchmark calls the `matmul` function twice. The first call is handled by the LLINT, the Baseline JIT and the DFG JIT layers while the second one is handled by the FTL JIT layer. Our implementation currently only support polyhedral optimization for the next invocation of a function even if JavaScriptCore supports layer switching inside a function call. The benchmark is executed three times with different sizes for the left matrix. In the three cases, the right matrix size is 3000x300. Figure 11 reports the speedup obtained on the second execution of the `matmul` function using four threads, i.e, one thread on each physical core of the machine.

From these results, it is clear that the benefit of performing polyhedral optimization depends on the amount of computation performed by the code. Nevertheless, as soon as matrices are large enough, the polyhedral optimization leads to a speedup which is almost the number of threads. In this benchmark, because our engine performs polyhedral optimization in parallel of the first execution of the function, the additional compile time required by Polly is hidden. Said differently, it is far longer to execute the function with the LLINT, the Baseline JIT and the DFG JIT layers than compiling the function with the FTL JIT.

7 Related work

We now review existing research works that recently addressed the parallel execution of JavaScript programs.

River Trail [9] is an extension of the language allowing the programmer to indicate to the JavaScript engine where parallelism can be exploited. It mainly provides an application programming interface allowing to express data parallelism. This is done through the addition of a new special JavaScript type called *parallel array* on which the programmer can apply parallel operations. River trail can be seen has a kind of map reduce framework for JavaScript. Compared to the goal pursued by the ongoing work presented in this paper, programs source code must be changed to be used with River Trail, and parallelization is not automatic.

More recently, additions towards concurrency in JavaScript have been introduced in the language specifications [4]. The main novelty is the capability to perform concurrent accesses to the new `SharedArrayBuffer` type. The role of threads is played by Web Workers in the browser environment. This feature is already supported by JavaScriptengines and the authors of JavaScriptCore started recently to investigate how the thread concept could be added to JavaScript [18]. As River Trail and compared to our proposal, these extensions propose new constructs in the language and do not address automatic parallelization.

Recently, Thread Level Speculation (TLS) systems have been proposed in the context of JavaScript implementations. Martinsen et al. [13] proposed to parallelize the execution of different JavaScript functions by integrating standard TLS mechanisms within the just-in-time compilation layer of Google's V8. Compared to our proposal, they cannot exploit loop level parallelism because they only execute different functions in parallel. TLS at loop level has also been proposed [14]. Compared to us, this work adds a non negligible speculation overhead. In addition to the JavaScript speculation that

must be handled by any JavaScript engine, such systems must also dynamically check that the memory accesses performed in parallel by the different threads do not break the sequential semantics of the program.

Finally, the closest work to our proposal, is also an extension of a JavaScript engine for automatic loop parallelization [17]. Our proposal is an extension of this work which only focused on so called DOALL loops. A DOALL loop does not exhibit any dependency between its iterations. Using the polyhedral model, thanks to its precise representation of loop dependencies, our goal is to parallelize more loops and perform complex optimizing loop transformations.

8 Conclusion

We have presented the challenges and solutions to make polyhedral optimization work on JavaScript programs. We integrated our proposal in the last layer of the JavaScriptCore engine which uses LLVM to generate efficient native code. We have modified the JavaScript engine to add more information about loop nests and we integrated Polly to produce optimized parallel code.

Our preliminary results demonstrate that polyhedral optimization could be beneficial in the context of JavaScript programs. We now need to complete the implementation of the mechanism handling parallel speculation failure. This development is a large amount of work in the context of a production engine such as JavaScriptCore. Its performance comes at the price of a very large code base containing a lot of corner cases. We also need to perform solid benchmarking experiments to confirm with numbers, at least on some standard benchmarks and applications, that polyhedral transformations are effective in a JavaScript engine. Looking at the benchmarks and applications from previous studies [14, 17], we are confident that polyhedral optimization will be beneficial.

The techniques presented in this paper are specific to the JavaScript language. Other dynamic languages like Python or Ruby are not direct targets of this work. Nevertheless, we believe that it could be very interesting to study polyhedral opportunities also for these languages.

Regarding the perspective opened by this work, we believe that it could be very interesting and challenging to merge polyhedral speculation as proposed in static languages [11] with JavaScript speculation. This polyhedral speculation concerns the memory accesses performed by loop nests that cannot be statically defined as SCoPs. At runtime, thanks to memory profiling, the loop nests conform to SCoPs can be optimized by polyhedral tools. In the context of JavaScript, we want to study how to merge memory accesses profiling in the first layer of the engine with the JavaScript profiling mechanisms already at work. From this memory profiling information, the JavaScript engine would construct memory accesses models. Finally, these models would be used in the last layers to perform polyhedral optimization on code that cannot be identified as SCoPs by tools like Polly or to perform other parallelization transformations.

References

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 589–606.

[2] Apple. 2017. JavaScriptCore. (Oct. 2017). Retrieved October 19, 2017 from https://trac.webkit.org/wiki/JavaScriptCore

[3] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 475–486.

[4] ECMAScript. 2017. ECMAScript Shared Memory and Atomics. (Jan. 2017). Retrieved October 19, 2017 from http://tc39.github.io/ecmascript_sharedmem/shmem.html

[5] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer US, Boston, MA, 1581–1592. https://doi.org/10.1007/978-0-387-09766-4_502

[6] Google. 2017. V8 JavaScript Engine. http://code.google.com/p/v8/. (2017).

[7] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Pptimizations on a Low-level Intermediate Representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[8] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. 2015. Optimistic Delinearization of Parametrically Sized Arrays. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15)*. ACM, New York, NY, USA, 351–360.

[9] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. 2013. River Trail: A Path to Parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 729–744. https://doi.org/10.1145/2509136.2509516

[10] Juan Manuel Martinez Caamaño. 2016. *Fast and Flexible Compilation Techniques for Effective Speculative Polyhedral Parallelization*. Ph.D. Dissertation. University of Strasbourg. https://hal.inria.fr/tel-01377758

[11] Juan Manuel Martinez Caamaño, Manuel Selva, Philippe Clauss, Artyom Baloian, and Willy Wolff. 2017. Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4192–n/a. https://doi.org/10.1002/cpe.4192 e4192 cpe.4192.

[12] Juan Manuel Martinez Caamaño, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, and Philippe Clauss. 2017. APOLLO: Automatic speculative POLyhedral Loop Optimizer. In *IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques*. Stockholm, Sweden, 8. https://hal.inria.fr/hal-01533692

[13] Jan Kasper Martinsen, Hkan Grahn, and Anders Isberg. 2017. Combining thread-level speculation and just-in-time compilation in Google's V8 JavaScript engine. *Concurrency and Computation: Practice and Experience* 29, 1 (2017), e3826–n/a. https://doi.org/10.1002/cpe.3826 e3826 cpe.3826.

[14] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. 2011. Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, Washington, DC, USA, 87–98. http://dl.acm.org/citation.cfm?id=2014698.2014898

[15] Microsoft. 2017. Chakra JavaScript Engine. https://github.com/Microsoft/ChakraCore/wiki/Architecture-Overview. (2017).

[16] Mozilla. 2017. SpiderMonkey JavaScript Engine. https://developer.mozilla.org/fr/docs/Mozilla/Projects/SpiderMonkey/Internals. (2017).

[17] Yeoul Na, Seon Wook Kim, and Youngsun Han. 2016. JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 64.

[18] Filip Pizlo. 2017. Concurrent JavaScript: It Can Work! (Aug. 2017). Retrieved October 19, 2017 from https://webkit.org/blog/7846/concurrent-javascript-it-can-work

[19] Sanjay Srivallabh Singapuram. 2017. Polly Support for Julia. (2017). http://pollylabs.org//2017/08/29/GSoC-final-reports.html

[20] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*. Pisa, Italy. https://hal.inria.fr/inria-00551516

[21] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. 2002. Instance-wise Points-to Analysis for Loop-based Dependence Testing. In *Proceedings of the 16th International Conference on Supercomputing (ICS '02)*. ACM, New York, NY, USA, 262–273. https://doi.org/10.1145/514191.514228