Removing Checks in Dynamically Typed Languages through Efficient Profiling

Gem Dot
Universitat Politècnica de Catalunya
Barcelona, Spain
gdot@ac.upc.edu

Alejandro Martínez
ARM
Cambridge, United Kingdom
almavi1980@gmail.com

Antonio González
Universitat Politècnica de Catalunya
Barcelona, Spain
antonio@ac.upc.edu

Abstract

Dynamically typed languages increase programmer's productivity at the expense of some runtime overheads to manage the types of variables, since they are not declared at compile time and can change at runtime. One of the most important overheads is due to very frequent checks that are introduced in the specialized code to identify the type of the variables.

In this paper, we present a HW/SW hybrid mechanism that allows the removal of checks executed in the optimized code by performing a HW profiling of the types of object variables. To demonstrate the benefits of the proposed technique, we implement it in a JavaScript engine and show that it produces 7.1% speedup on average for optimized JavaScript code (up to 34% for some applications) and 6.5% energy reduction.

1. Introduction

The popularity of dynamically typed programming languages has increased significantly in the last years [33]. The most common languages are JavaScript, Python, PHP, Ruby, Smalltalk and Self. Nowadays, JavaScript is the most popular one [8].

The dynamic typing feature of these languages require that their applications are Just-in-Time (JIT) compiled. A characteristic of these languages is that variables are neither declared nor bound to a particular type in the source code, and their types may change during the execution. Compilers usually make some assumptions about the types of the variables, in order to generate specialized code, which is significantly more efficient than a generic one. These assumptions are based on some dynamic profiling information collected previously by the runtime, during the execution of the application.

Therefore, we can identify two main types of overheads in the execution of these languages: the overheads associated to the dynamic compilation, profiling, garbage collector, and other housekeeping tasks and the overheads related to the verifications of the assumptions that have been introduced in the specialized code. In modern JITs, these verifications are referred to as checks.

In this paper, we take an innovative approach to eliminate some of these checks. A detailed characterization of representative applications shows that most of the time variables stay with the same type throughout the whole execution of a program. We have also observed that programs normally use a limited number of classes (as in object-oriented programming) and these classes tend to remain constant. Based on these observations, we propose a hardware mechanism to track classes and more specifically, which object properties and arrays of these classes contain objects that always have the same type (i.e., they are *monomorphic*). Once we have identified these properties, the information is passed to the compiler, which can use it to remove some checks and perform new optimizations assuming that the type of these variables will never change. The last key element of our scheme is an efficient way to verify the compiler assumptions. For this purpose, when a store that writes an object property is executed, the Memory Unit sends a request to a special hardware structure called the Class Cache, which tracks the properties that so far are *monomorphic*. If this is the case for the property to be written, then an exception is triggered when trying to write them, as long as this property has been used to remove any check. This exception is captured by the runtime, which can choose to execute a non-specialized version of the code or recompile the offending function.

In this work we use JavaScript as a vehicle to demonstrate the benefits of the proposed technique. First of all, we develop an instrumented version of V8, the JavaScript engine used by Google, which allows us to perform a detailed characterization for a representative set of benchmarks. In particular, this analysis quantifies the overhead due to checks.

The proposed scheme attacks this overhead and results in significant improvements in performance (7.1% speedup on average for optimized code and up to 34% for some applications) and energy consumption (6.5% reduction on average).

The remainder of this paper is organized as follows: In section 2, the related work is presented. Section 3 provides an overview of the most important aspects of the V8 JavaScript engine. In Section 4, we describe the proposed hardware mechanism and some optimizations that rely on it. The results are presented in section 5 and finally, section 6 concludes this paper.

2. Related Work

The state-of-the-art technique that modern JITs use to deal with the dynamic typing feature is known as *Inline Caching*

[21, 22]. It is based on specializing every code section that accesses a variable or an object property, according to the types that have been previously seen during profiling.

Inline Caching technique was first introduced in the Smalltalk compilers [21] and it was later used in Self compilers [1][5][12][13][17][18]. One important contribution [17] evolves this technique to *Polymorphic Inline Caching.* There are some recent contributions that improve *Inline Caching* performance for JavaScript compilers. One of them [29] is based on applying classic optimizations to specialized code based on profiled runtime values. Ahn et al. [14] proposed a technique to increase the hit rate of Inline Cache accesses and improvements to the *Polymorphic Inline Caching* scheme.

Various proposals to reduce the overhead of checks can be found in the literature [2][3][4][7][10][20]. Checked Load [20] introduces automatic checking of types. Basically, checks are not removed, but partially performed implicitly by a specialized hardware instead. However, this work deals only with those checks that are necessary just before an object property is loaded.

3. The V8 JavaScript Engine

In this paper we evaluate the proposed technique and demonstrate its benefits for a JavaScript [24] environment, which is the most popular dynamically typed language nowadays. In particular, we use the JavaScript engine from Google, which is known as V8 [9][11[15][16]. Other vendors have alternative engines for JavaScript. Apple uses Nitro [26] (previously known as SquirrelFish Extreme), Mozilla uses SpiderMonkey [19] and Microsoft uses Chakra [34]. The proposed scheme can be applied to other dynamically typed languages and engines.

V8 is open source and widely used. V8 compiles JavaScript to native machine code (IA-32, x86-64, ARM, or MIPS) before executing it, instead of using other traditional techniques such as interpreting it and compiling only those sections that are frequently used. Below we outline its internal operation because it will help to understand how our scheme works.

V8 was specifically designed for fast execution of large JavaScript applications. Depending on the nature of a specific program, its performance is normally better if it runs the same functions repeatedly, instead of running many different functions very few times each. This is because V8 optimizer focuses on hot functions (i.e. those functions that execute more often). V8 includes two compilers: The first one is called Full Codegen, which has light overhead but produces generic code; The second one is called Crankshaft, and is slower but produces more optimized code, including code specialization [6][35][36].

3.1 Hidden Classes

JavaScript is an object-oriented programming language in which classes are not explicitly specified by the programmer. However, hidden classes are immutable entities introduced by V8 that represent object types. In other words, objects that share the same hidden class have the same type. In this regard, each hidden class represents an ordered set of object named variables and methods (i.e., object properties). When at runtime an object is created for the first time, its hidden class is also created. Moreover, every time that a new property, *x*, is added to an object, the object changes its hidden class to another one, which contains all properties of the old hidden class plus the property *x*. If this latter hidden class does not exist yet (i.e. it is the first time that *x* is added to the old hidden class), then it is created. Note that the first field of each object contains the address of the hidden class descriptor, which is also used as *hidden class identifier*.

Furthermore, objects contain two reserved special properties, which are used to manage their numbered variables (i.e., variables that are indexed by a number): The *elements array pointer* and the *elements length*, which are located in the third and fourth 8-byte words of the object, respectively. The former contains a pointer that targets an internal array called the *elements array*, which contains all the variables of the object that are indexed by a number. Note that the addition of a numbered variable to an object does not change the hidden class of that object. The latter contains the length of the *elements array*, which can change during the execution. However, in some other cases, the *elements length* is directly located inside the *elements array*, instead of the object itself.

3.2 Inline Caching

Inline Caching technique has a twofold purpose: recording information concerning the types of objects and improving the performance of the system lookup routine used to disambiguate the type of objects when they are accessed. Each of the two compilers, Full Codegen and Crankshaft, applies this technique in a different manner.

During the execution of the generic code produced by Full Codegen, for each object property access, a *call* instruction is executed, which is constantly patched by the runtime. The first time that the access is produced, the *call* instruction targets a lookup routine that performs a sequence of steps that determine the type of the object and find the offset for that property. Then, the access is performed by this routine. Since this process is quite costly, a special software structure called Inline Cache (IC) is created, which contains specialized code (i.e., the code to perform that access) for that particular object type and the offset found. Then, the *call* instruction is patched to point to this Inline Cache. Therefore, subsequent accesses are substantially faster if the type keeps being the same. In this regard, a checking operation needs to be inserted before the generated code to verify that the type is the expected one.

The information recorded during the above process is also used by Crankshaft (the optimizing compiler) to perform more aggressive optimizations for hot code. In this regard, Crankshaft generates specialized code that performs directly the property accesses for those hidden classes previously encountered by the Inline Caches, instead of executing a call instruction for each of them. Also, checking operations are introduced in this specialized code, in order to verify that the encountered type is the expected one; otherwise (i.e. when a checking operation fails), the optimized code falls back to non-optimized code through a deoptimization bailout. Note that the specialized code produced by Crankshaft is much more efficient than the non-optimized code produced by Full Codegen, due to the fact that the *call* instructions are not present, which also allows that other standard compiler optimizations can be performed over this specialized code. Therefore, this specialized code is key to efficiently implement JavaScript characteristics, such as the dynamic typing feature.

3.3 Checking Operations

Next, we list the most common checking operations used in V8:

- **Check Map**: The first slot in each V8 object points to its class identifier (corresponding to its hidden class, which is its type). In this case, the type of an object is checked to be the same as that of another type, which has been seen before.
- **Check SMI**: A register containing a boxed object can be of two types: either a SMI (small integer), which has its last bit cleared or a pointer, which has its last bit set. In this case, the last bit of a register is checked to know whether it is a SMI.
- **Check Non-SMI**: The opposite of Check SMI.

In Figure 1, we show the breakdown of dynamic instructions for *Checks*, *math assumptions* and *Tags/Untags* categories. *Checks* category refers uniquely to the checking operations listed above. However, both *math assumptions* and *Tags/Untags* categories can also contain some of these checking operations. As we can see, 19.5% of the dynamic instructions correspond to these categories for representative benchmarks (Octane [30], Kraken [31] and SunSpider [27] suites) on their steady state, which is a significant amount of overhead. In V8, the most common kind of checking operations is referred to as *Check Maps*. However, other kinds of checks are also common during the execution of optimized code, such as *Check Non-SMI*.

Tags/Untags are operations used to box and unbox number values. When a number value is boxed, the register that supposedly contains that number does not contain the value directly. Instead, it contains the object (i.e. the address of the object, but its last bit is set to 1) where that value is stored. As an exception, if the boxed number is a SMI (i.e., a small integer), the value is located in the 32 most significant bits of the register and the last bit is set to 0. Note that some of the untagging operations also perform *Check Maps*, *Check Non-SMI* and *Check SMI* operations before the value

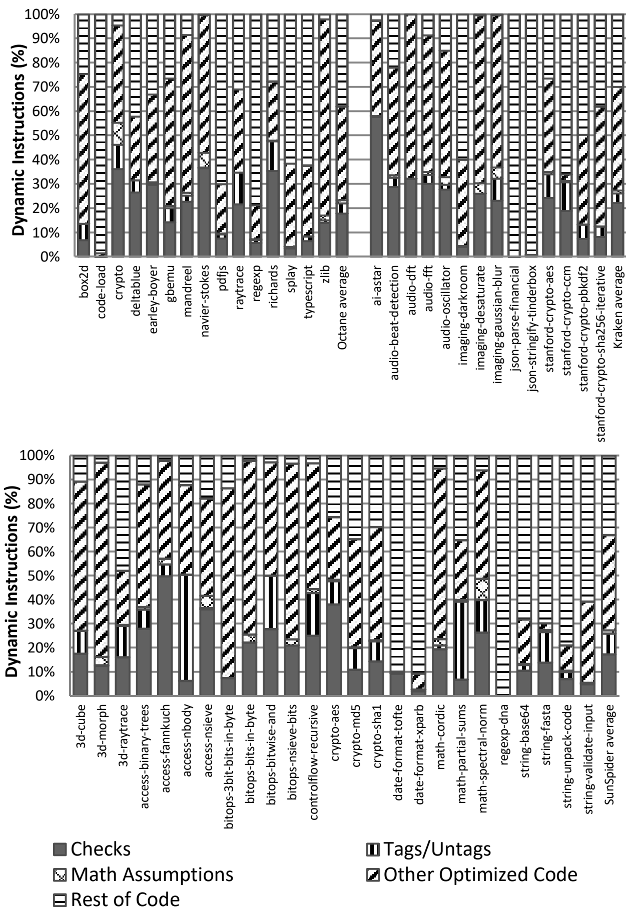


Figure 1. Breakdown of dynamic instructions.

is untagged, in order to verify that the number to be untagged has the expected type (i.e., either SMI or Non-SMI). We have included these additional checking operations in the *Tags/Untags* category.

On the other hand, the *math assumptions* category corresponds to some math operations that require some runtime value verifications on their source operands or the produced result. The most common scenarios are overflows of SMIs and division by 0.

4. Dynamic Type Profiling and Optimization

In this section we present our proposed technique and some of the optimizations that it allows, which reduce some of the most important overheads due to dynamic typing. First, we explain the reasons that have motivated us to devise this new technique. Next, we present the design and functionality of the technique and lastly, we describe some optimizations that make use of it.

4.1 Motivation

We have observed that for many benchmarks, the main source of the overhead quantified in Figure 1 comes from checking operations of objects obtained from properties or *elements arrays*. In Figure 2 we quantify this overhead for both the whole application and optimized code. Note that we also include part of the overhead of untagging operations, which corresponds to the checking operations needed before unboxing a value.

We can see that about half of the total benchmarks present a zero overhead. One of the major reasons for this is that some of them do not exploit the object-oriented paradigm of JavaScript and therefore, they do not perform many dynamic object accesses. Another important reason is that although some of these benchmarks perform a significant number of object accesses, they do not require any checking operation after these accesses because they use built-in JavaScript objects for their computations. Note that most of the properties from built-in objects are either read-only or type specific (e.g., *Float64Array* objects) and therefore, they do not require any type check after they have been obtained. Finally, there are a few number of benchmarks that still are spending a significant fraction of the time in non-optimized code (e.g., *string-base64* benchmark, from SunSpider suite), which does not suffer from the overheads targeted in this section. For the rest of this paper, in order to evaluate the impact of these particular checking operations, we have selected the benchmarks with more than 1% overhead, which represent 27 out of 54 benchmarks. In this regard, we have averaged the benchmarks suites of Figure 2 only for these selected benchmarks. We can see that these overheads represent 10.7% of the total dynamic instructions for the whole application in steady state. Furthermore, if we take into account only the optimized code, these overheads represent 15.9% of the total dynamic instructions, which is quite significant.

On the other hand, we have observed that most of the type checks quantified in Figure 2 are performed over *monomorphic properties* or *monomorphic elements arrays* (i.e., those that contain objects with the same type throughout the whole execution of the program). We have quantified that 66% of the object load accesses target either *monomorphic properties* or *monomorphic elements arrays* as showed in Figure 3. Lastly, we have also observed that many checking operations target these object load accesses. Therefore, the key idea behind our technique is that these checking operations can be removed as long as the monomorphism of the variables is preserved during the execution of the program.

Finally, we have also observed that programs normally use a limited number of hidden classes and these classes tend to remain constant. Our analysis of representative workloads reveals that the number of hidden classes is relatively small in almost all benchmarks: they all use up to 32 hidden classes excepting *box2d* and *raytrace*, from Octane. Therefore, the hardware structure that we use to keep the hidden class information about *monomorphic properties* or *monomorphic elements arrays* (i.e., the Class Cache) does not have important storage requirements.

4.2 The Proposed Mechanism

The proposed mechanism is based on a small, special new HW/SW structure called the Class Cache that keeps information about *monomorphic properties* and *monomorphic elements arrays* at hidden class level. In other words, for each hidden class, it stores which properties and *elements arrays* contain objects with the same type (i.e. a particular hidden class or SMI) during the whole execution of a pro-

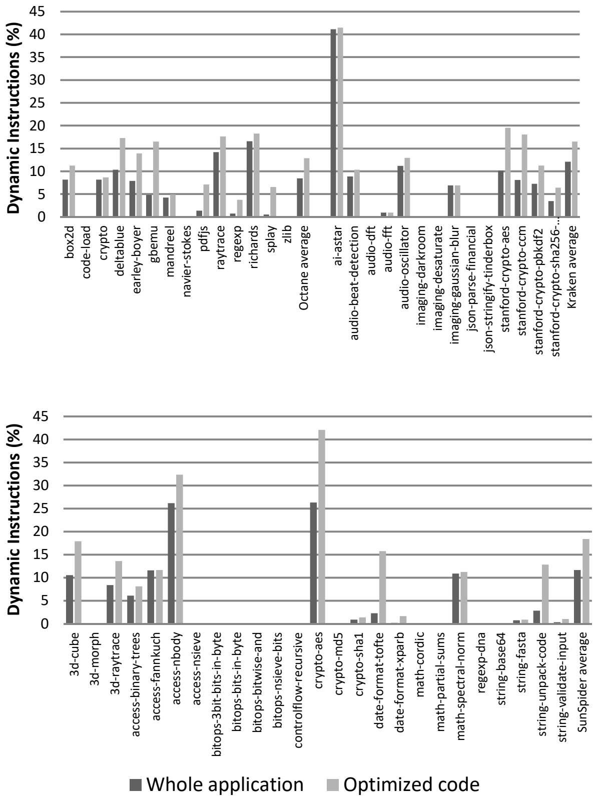


Figure 2. Overhead produced by checking and untagging operations after performing object load accesses.

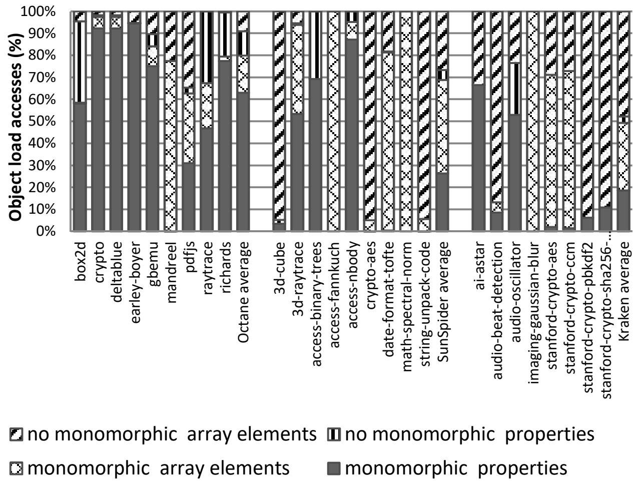


Figure 3. Object load accesses to *monomorphic properties* and *monomorphic elements arrays*.

gram. This structure collects information during the execution of the code produced by the Full Codegen compiler (i.e., non-optimized code). This information is used to perform the optimizations in the code produced by Crankshaft compiler (i.e. optimized code). Then, this structure is accessed to verify the assumptions about *monomorphic properties* and *monomorphic elements arrays*. In this regard, the class properties' information is read on demand when a store that targets an object property is executed. Similarly, the class *elements array*' information is read on demand when a store that targets an *elements array* is executed.

On the other hand, a new entry is stored in this structure every time that a property of a new hidden class is written for the first time. Below we explain in detail the new structures used for this mechanism and how these two phases, profiling and optimization, work

4.2.1 The New Structures

In this section, we present the software and hardware components used for the proposed mechanism.

4.2.1.1 The Class List

The runtime maintains a software structure that we call the Class List, which stores the object types of the *monomorphic properties* and *monomorphic elements arrays* for each hidden class of the JavaScript application. As we outlined in Section 3.1, the V8 engine creates these hidden classes dynamically as objects are constructed. For each hidden class, the Class List contains as many entries as cache lines the objects belonging to this class occupy. Note that for each 64-byte cache line, there are up to seven 8-byte properties, because the first 8-byte word contains the identifier for the hidden class along with the corresponding relative cache line position. For each entry, it contains the following information.

- **ClassID**, **Line**: The identifier of the hidden class together with the relative cache line that this entry represents. As commented above, each entry represents up to seven properties of the object. Note that these identifiers are not the same that the ones used by V8 (described in section 3.1), which need 48 bits for their representation because they are memory addresses of the hidden class descriptors. Instead, the identifiers for the hidden class that we use (i.e., *ClassID*) are consecutive numbers, which allow us to represent them with only 8 bits. On the other hand, the *Line* attribute is represented with 8 bits. Note that the Class List occupies only 2^16 entries, which are located together in the same memory region. As special case, the SMI (i.e., small integer) type is encoded as *11111111*.
- **InitMap**: An 8-bit map that indicates for each property of the entry whether it has been initialized in any object. This bitmap is initialized to zeros, indicating that no property has been initialized so far. Note that each bit represents a different property, so only the 7 least-significant bits are used in practice.
- **ValidMap**: An 8-bit map that indicates for each property of the entry whether this is *monomorphic* so far. As with *InitMap* field, each bit represents a property of the object. This bitmap is initialized to *11111111*, indicating that all properties are *monomorphic*. Note that the first time that a type is profiled for a particular property, the corresponding bit of the *InitMap* field is set to *1*. Then, if the type of that property differs from the profiled one, the corresponding bit of the *ValidMap* field is set to *0* and this will never be set to *1* again.
- **SpeculateMAP**: A bit map that indicates for each property whether a speculative optimization that depends on this property has been applied. This field is initialized to zeros.
- **Prop1 … Prop7**: Seven 1-byte fields that contains the *ClassIDs* that are profiled for each property of the entry. As special case, the *Prop2* field of the first line of each object contains the *ClassID* that has been profiled for the objects contained in the *elements array*, as long as all the objects contained in this array have been profiled with one single *ClassID*.
- **FunctionList**: For each property, the list of functions that have been speculatively optimized based on this property.

In Table 1 we show an example of a Class List, which contains two hidden classes: *NodeList* and *GraphNode*. *GraphNode* occupies two cache lines because it has 9 prop-

ClassID, Line	**InitMap**	**ValidMap**	**Speculate Map**	**Prop1**	**Prop2**	**…**	**Prop6**	**…**	**FunctionList (property: functions)**
GraphNode, 1	01111111	11111111	00000010	….	….	…	*classPosition*	…	6th *property*: *findGraphNode*
GraphNode, 2	01100000	11111111	00000000	….	….	…	….	…	---
NodeList, 1	01111000	11111111	00100000	….	*GraphNode*	…	….	…	2nd property: *findGraphNode*
…	…		…	….	….	…	….	…	…

Table 1. Class List Structure.

erties. In the first cache line, the *InitMap* field indicates that all the properties have been initialized for that line and therefore, *Prop1* to *Prop7* fields contain the profiled ClassID for each property. Note also that the *ValidMap* field indicates that all the *ClassIDs* profiled for each property are valid (i.e., *monomorphic*), which means that they can be used for our optimizations. Moreover, *findGraphNode* function has been speculatively optimized assuming that the sixth (*position*) property is *monomorphic*, and its type is *classPosition* hidden class according to the profiling data. The two properties contained in the second cache line have not been used to optimize any function, despite the fact that both properties are valid and initialized.

NodeList objects occupy only one cache line because they contain four properties. In Table 1, all the properties of this hidden class have been initialized and are considered valid. Note also that the second property of this hidden class has been used to speculatively optimize *findGraphNode* function. As commented above, this is a special property that contains the *elements array pointer* of the object. Therefore, the hidden class profiled for this property (i.e., *GraphNode*) corresponds to the type of the objects contained in the *elements array* of *NodeList*.

Besides, there is a special register that has a pointer to this Class List in memory, in a similar way that there is a special register that points to the memory translation tables. Note that the Class List entries are together in the same 64 KB memory region and therefore, all the entries are indexed by adding to this special register the resulting value of concatenating the *ClassID* and the *Line* number attributes.

4.2.1.2 New Machine Instructions

The compiler (both Full Codegen and Crankshaft) identifies which stores can affect objects and they are encoded with a new different opcode through two new instruction called *movStoreClassCache* and *movStoreClassCacheArray*. The former is used for stores that target properties and the latter is used for stores to the *elements array* of an object. These instructions are similar to a *mov* x86-64 instruction, but in addition to the L1 data cache write, they perform a request to the Class Cache in parallel.

Besides these instructions, two more new instructions are required by our mechanism, which are called *movClassID* and *movClassIDArray*. The former loads the *ClassID* of an object to a special 8-byte register called *regObjectClassId*. If the object is a SMI (i.e., the least-significant bit of the register that represents the object is *0*), the corresponding *ClassID* value for SMI's (i.e., *11111111*) is directly loaded to the *regObjectClassId*. Otherwise, since the register that represents the object contains the memory address where the object resides, the *ClassID* is obtained from the first 8-byte word of this location. Note that this register will be used by both *movStoreClassCache* and *movStoreClassCacheArray* instructions. The latter works similar to the former, but instead of loading the *ClassID* to the *regObjectClassId* special register, it is loaded to a specified register among an additional set of four special 8-byte registers called *regArrayObjectClassId0-3.* Note that these registers will be consumed only by *movStoreClassCacheArray* instructions.

4.2.1.3 The Class Cache

The Class Cache is a cache of the Class List, in a similar way as the TLB is a cache of the Page Table. When a special store that writes to an object property or an *elements array* is executed, the Memory Unit sends a request to the Class Cache that includes the *ClassID* of the hidden class that contains that property or array, the relative cache line (0 in case of a *movStoreClassCacheArray* instruction*),* the position of the property that is written (3 in case of a *movStoreClassCacheArray* instruction*)* and the *ClassID* of the object to be stored.

In Figure 4 we depict a Class Cache request for a

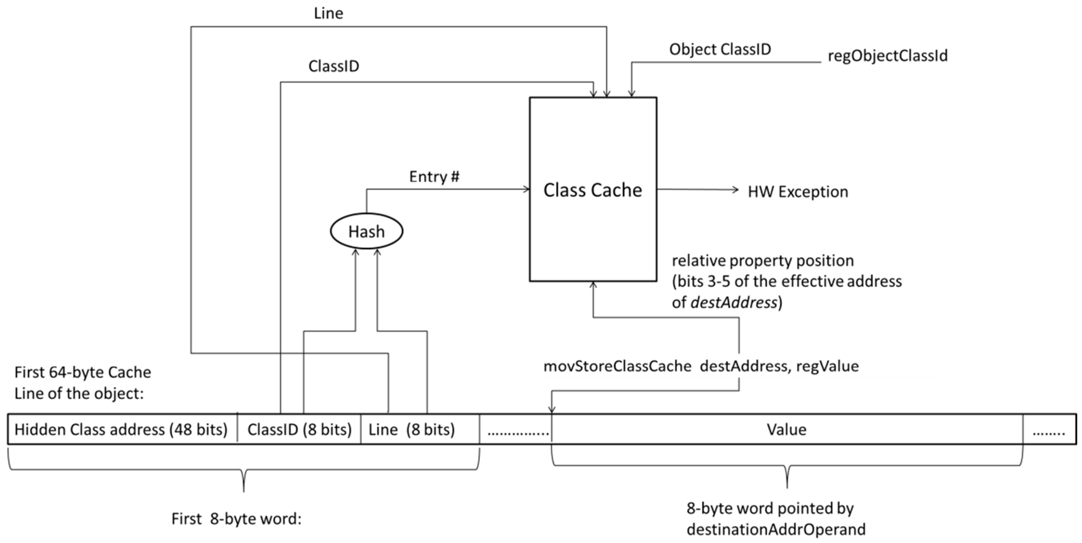


Figure 4. Block diagram of a Class Cache access a for a *movStoreClassCache* instruction.

movStoreClassCache instruction. Note that in V8, the first 8-byte word of the first cache line of an object contains its *hidden class identifier*, which occupies the 48 least-significant bits. Therefore, we store the *ClassID* and *Line* parameters in the two most significant bytes of the first 8-byte word. Furthermore, for objects larger than one cache line, the rest of lines also contain the *ClassID* and *Line* parameters in the same position (and the rest of the bytes in the first 8-byte word are not used). Consequently, the proposed mechanism requires that objects are created aligned to cache lines. Note that this restriction is not costly [20] and both Nitro [26] and Mozilla JavaScript engines [19] already apply it. Moreover, a Class Cache request needs to specify the relative position that the property occupies inside the cache line. Since objects are cache line aligned, this information is contained in the bits 3-5 of the store address. Finally, each execution of a *movStoreClassCache* instruction requires the previous execution of a *movClassID* instruction, which loads the *ClassID* of the object that is written in the selected property to the *regObjectClassId* register.

In Figure 5 we illustrate a Class Cache request for the *movStoreClassCacheArray* instruction. This scenario is very similar to the previous one, with two main differences. The first one is that the *relative property position* and the *Line* parameters of the Class Cache are fixed to *3* and *0*, respectively. This is because the field inside the Class Cache that is reserved for the second property of each hidden class (i.e., the special property that contains the pointer to the *elements array*) is used to keep the *ClassID* that has been profiled for the objects contained in the *elements array.* Note that this special property will never be used by a *movStoreClassCache* instruction. The second difference is that the *ClassID* parameter of the Class Cache (i.e., the *hidden class identifier* of the object that contains the array in which the store will write) comes from another special register (*regArrayObjectClassId0-3*), which is selected by the *movStoreClassCacheArray* instruction. In this regard, each execution of a *movStoreClassCacheArray* instruction requires also the previous execution of a *movClassIDArray* instruction, apart from the corresponding *movClassID* instruction. This *movClassIDArray* instruction loads the *ClassID* of the object that contains the *elements array* to one of the *regArrayObjectClassId0-3* registers.

Note that in the optimized code, both *movStoreClassCache and movStoreClassCacheArray* instructions are inserted only for those properties or *elements arrays* that still are considered *monomorphic*. Otherwise, a regular store is used. Furthermore, the *movClassIDArray* instructions can be moved out of the loop in many cases, as long as the variable that contains the object is not modified inside the loop and there are not function calls inside this loop. For this reason, we have four *regArrayObjectClassId0-3* registers, in order to move out of the loop up to four *movClassIDArray* instructions for different objects that are accessed inside the loop.

Each Class Cache entry contains the *ClassID*, the *Line*, the *InitMap,* the *ValidMap* and the *SpeculateMAP* attributes from the Class List, as we can see in Figure 6. The *ClassID* and *line* parameters are used to index the Class Cache. The Class Cache checks whether it has the corresponding entry stored, as we can see in the left upper part of Figure 6. If the class is not present, its information is obtained from the Class List in memory, in a similar way to a TLB miss, and one of the entries is replaced and copied back to the Class List. Once the requested entry is in the cache, the corresponding bits of *InitMap*, *ValidMap* and *SpeculateMap* are selected by the relative property position input parameter.

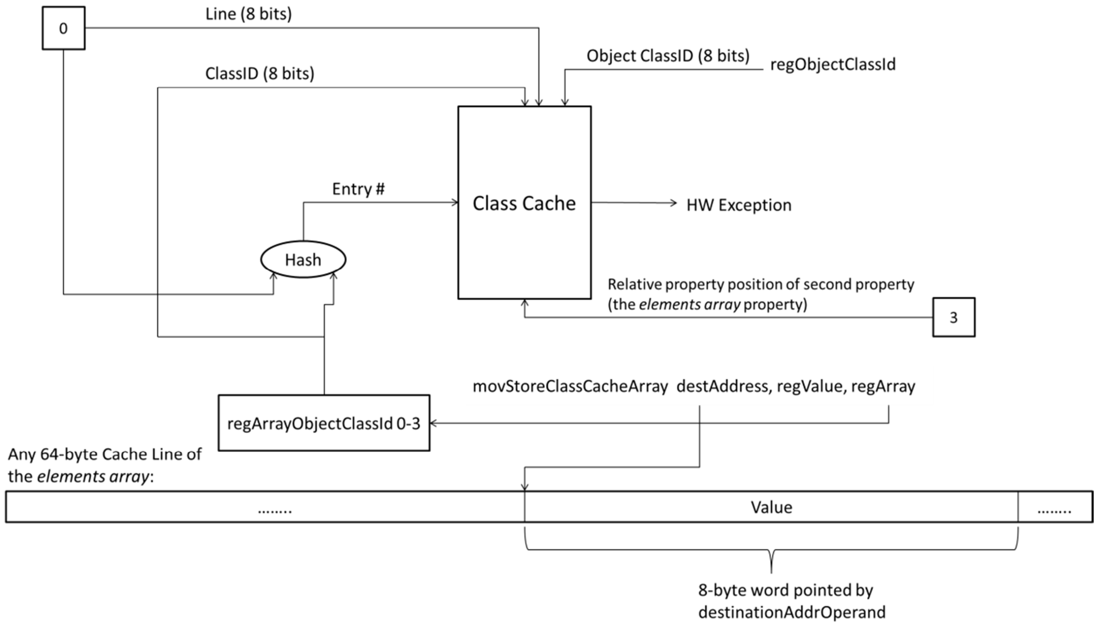


Figure 5. Block diagram of a Class Cache access for a *movStoreClassCacheArray* instruction.

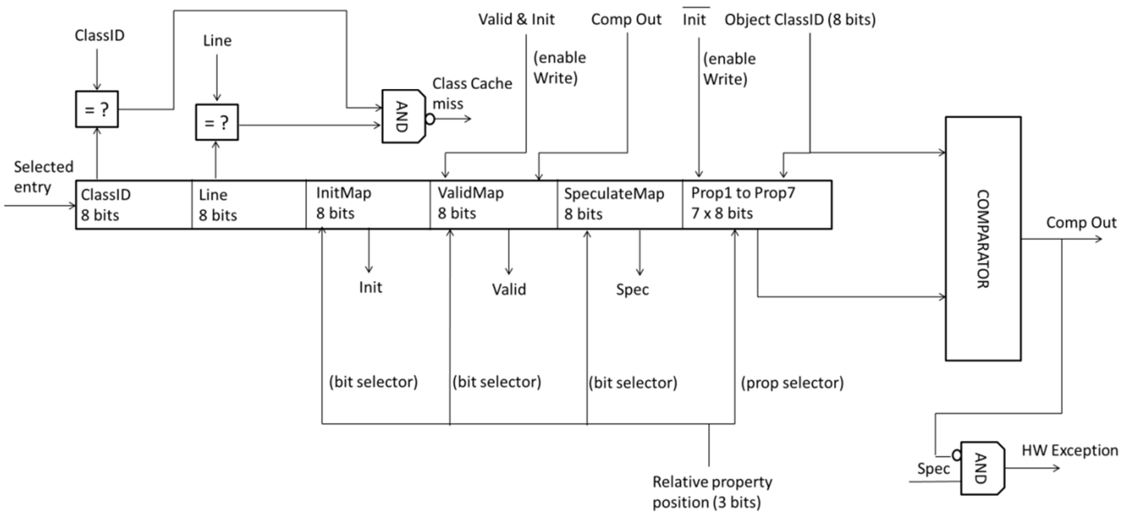


Figure 6. Scheme of a Class Cache entry.

Moreover the corresponding field with the profiled *ClassID* (*Prop1-Prop7*) is selected by this input parameter.

The first time that a particular property is selected, the corresponding InitMap bit contains a 0 value, indicating that no ClassID have been profiled yet for that property. Therefore, the Object ClassID input parameter is stored in the corresponding prop1-prop7 field and the InitMap bit is set to 1. For the following accesses to that property, the Object ClassID input parameter is compared to the corresponding prop1-prop7 field. When this comparison is not equal, the corresponding ValidMap bit is set to 0 and it will never be set to 1 again. Moreover, the corresponding SpeculateMap bit is checked. If this bit is set to 1, then a HW exception is raised, because at least one function was optimized assuming that this property was monomorphic, but it is not anymore. The exception routine deoptimizes the offending functions and sets to 0 the corresponding SpeculateMap bit.

4.2.2 How the Mechanism Works

As explained in section 3, when a function is invoked by the first time, the code is compiled by Full Codegen and then it is executed. This execution may create new classes and their corresponding entries in the Class List and the Class Cache. In addition, it updates all the fields of the Class Cache accordingly. That is, when a property or *elements array* is written, the Class Cache is accessed, in order to perform the corresponding profile.

When a function has been executed often enough (hot function), the runtime compiles it with the more aggressive compiler (Crankshaft). Using the information collected by the Class List, the compiler can perform some speculative optimizations that we describe later (section 4.3), based on the assumption that *monomorphic properties* or *monomorphic elements arrays* will remain so for the rest of the execution. When any of these optimizations are applied, the relevant bit in the *SpeculateMAP* of the corresponding property or *elements array* is set to *1*. Figure 7 illustrates this optimization process.

For every store to an object property or *elements array*, the Class Cache is accessed, in order to perform the corresponding hidden class profiling and to check whether a misspeculation has occurred (i.e. a *monomorphic property* or *elements array* is not *monomorphic* anymore and it had previously been used to optimize at least one function). If so, then a hardware exception is triggered. In the exception routine, the V8 runtime is called, which invalidates and recompiles all the functions that have performed speculative optimizations assuming that the property or *elements array* was *monomorphic*. These functions are identified by the runtime through the *FunctionList* field of the Class List. Note that the application state is correct because up to this point in the execution all the assumptions were correct, so no recovery action is required.

There is a situation that deserves special attention, which is due to functions in the program stack (i.e. function f calls function g, and g causes an exception that requires f to be deoptimized). This case can be handled by performing on-stack-replacement, which is a technique that modern JavaScript engines already support.

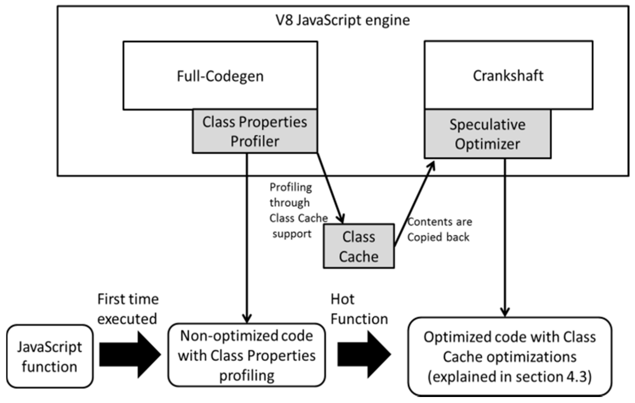

V8 JavaScript engine
Full-Codegen
Class Properties Profiler
Crankshaft
Speculative Optimizer
Profiling through Class Cache support
Class Cache
Contents are Copied back
First time executed
Hot Function
JavaScript function
Non-optimized code with Class Properties profiling
Optimized code with Class Cache optimizations (explained in section 4.3)

Figure 7. Optimizations process.

Although this technique introduces some overheads (extra *movClassID* and *movClassIDArray* instructions, larger objects, Class Cache misses), it allows for new compiler optimizations, and the net benefit is a significant reduction in execution time and energy consumption, as we will see in the next sections.

4.3 New Speculative Optimizations

Functions compiled with the non-optimizing compiler do not contain any speculation and are executed as usual. When functions become hot and are optimized by the Crankshaft compiler, the information contained in the Class Cache and the Class List is used to optimize the generated code. Below we describe several new optimizations that we have developed based on this scheme. Note that these optimizations also includes checking operations that are necessary for the *Tags/Untags* quantified in Figure 1

4.3.1 Check Maps Elimination

We remove the Check Maps operations that verify *monomorphic properties* or *monomorphic elements arrays*.

4.3.2 Check Non-SMI Elimination

We remove the *Check Non-SMI* operations that verify the *monomorphic properties* or *monomorphic elements arrays* that are profiled as non-SMI.

4.3.3 Check SMI Elimination

We remove the *Check SMI* operations that verify *monomorphic properties* or *monomorphic elements arrays* that are profiled as SMIs.

5. Performance Evaluation

In this section, the benefits of the proposed technique are evaluated in terms of execution time and energy consumption. The V8 JavaScript engine has been extended to include the proposed optimizations. The hardware has been modeled through Marss [25], which is a cycle-accurate, full-system simulator of the x86-64 architecture, with a micro-architectural configuration closely matching a Nehalem core. Energy consumption is measured through the McPat simulator [32] and Cacti [23]. We run three commonly used benchmark suites: Octane [30], Kraken [31] and SunSpider [27]. We focus on the steady state, to center on the execution of non-optimized code, which is achieved by executing the benchmark ten times and taking statistics from the tenth iteration.

5.1 Cycle Count Improvements

In this section we evaluate the performance benefits of our technique, as measured with the Marss cycle-level microarchitecture simulator. Table 2 shows the main microarchitectural features of the simulated core, which resemble those of a Nehalem core [28]. The Class Cache has 128 entries and 2-way set associativity. We have chosen this configuration because it achieves more than 99.9% of hit rate for all the benchmarks, with very low hardware cost.

Figure 8 shows the speedups for both the optimized code and the whole application. Regarding the former, our technique achieves an average speedup of 7.1%. We can see benchmarks with gains up to 34%. This confirms that our technique has an important impact on the execution of many JavaScript applications.

If we look at the whole application, including all the runtime, the average speedup is 5%. This is still an important benefit and, as discussed above, we expect it will improve as JavaScript applications become more compute intensive and the relative overhead of the housekeeping tasks decrease.

A remarkable case is *ai-astar* benchmark, from Kraken, which achieves a 34% of speedup. This benchmark is executing most of the time a loop with many object property accesses, which require an important number of checking operations that are removed by our optimizations, which more than half are *Check-Maps* operations. Note also that a *Check-Maps* operation performs a memory access, in order to obtain the *hidden class identifier* of the object. We have observed that after removing most of these memory access-

Issue width	4
Instruction Issue queue	36 entries
Window size	128
Outstanding load/stores	10
L1 load latency	2 cycles
Itlb	128 entries
Dtlb	256 entries
Il1 cache	32 KB, 4-way
Dl1 cache	32 KB, 8-way
L2 cache	256 KB, 8-way
Class Cache	128 entries, 2-way

Table 2. Simulated Micro-architecture configuration.

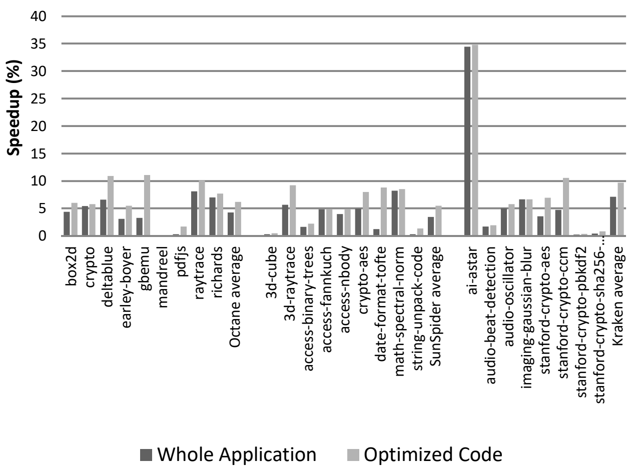


Figure 8. Improvement in number of cycles.

es, the DL1 hit rate, the L2 hit rate and the Dtlb hit rate have improved by 20%, 40% and 37% respectively, which indicates that memory accesses are an important bottleneck for this benchmark.

5.2 Energy Reduction

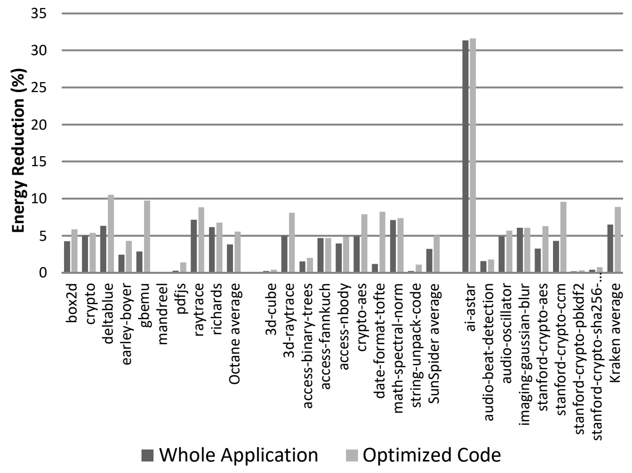
Figure 9 shows the energy savings of our technique for the three benchmark suites, which are measured through the McPAT simulator [32]. We used CACTI [23] to obtain the energy consumption of the Class Cache. Energy consumption is reduced by 4.5% on average for the whole application and 6.5% for optimized code. These savings come mainly from the reduction in number of executed instructions (which results in less dynamic energy) and execution time (which results in less leakage energy). Again, Kraken suite achieves the best energy savings with a 6.5% improvement. The consumed energy of this suite is also significantly reduced for optimized code, by 8.8% on average.

5.3 Incurred Overheads

In this section we present a detailed analysis of the overheads incurred by our technique.

5.3.1 Warm-Up Period

The amount of time constructing the Class List is proportional to the number of hidden classes that are dynamically



created as objects are constructed. On the other hand, the number of hidden classes is relatively small in almost all benchmarks, which use up to 32 hidden classes excepting two benchmarks. Therefore, this overhead can be considered as negligible.

5.3.2 Class Cache Hits

Every time that a special store instruction that targets an object is performed, the Class Cache has to be accessed at the same time as the data is written to L1 data cache. Therefore, as long as the access hits in the Class Cache, we do not incur any penalty for the *movStoreClassCache* and *movStoreClassCacheArray* instructions.

5.3.3 Class Cache Misses

When a miss in the Class Cache happens, the information has to be retrieved from the Class List, which resides in main memory, and is rather slow operation. However, the hit rate of a Class Cache of just 128 entries and 2-way associativity is higher than 99.9% for all benchmarks and thus the penalty of misses is negligible.

5.3.4 Larger Objects

The objects whose size is higher than 64 bytes (one cache line) require an extra memory word for each extra. The fact that some objects are slightly larger (ranging from 7% to 11% of memory space increment for objects that occupy more than one cache line) may affect the L1 Data Cache hit rate. However, most of the object property accesses (79%) target the first cache line. Therefore, the L1 Data Cache miss rate hardly increases and this overhead is not relevant.

5.4 Hardware Cost

The Class Cache occupies less than 1.5KB, which represents less than 0.04% of the total area of the core, measured through McPAT [32] and CACTI [23]. Similarly, the energy consumption of this hardware structure has a negligible impact in total consumption of the core.

Note that a pure software implementation of the proposed technique would be possible but would result in significant penalties, which would more than offset its benefits.

6. Conclusions

Dynamically typed programming languages have become very popular and are widely used nowadays. In these languages, performance is significantly burdened by the fact that object types have to be constantly checked at run time.

In this paper, we have proposed a new mechanism, the Class Cache, which allows a number of optimizations based on code specialization for particular object types. The specialization is based on a run time profiling that is extremely accurate. Besides, the proposed scheme detects when the specialized code is no longer correct before executing it, so there is no need for providing a recovery mechanism. In those cases, an exception is triggered and the code is recompiled to a non-specialized version that is guaranteed to be correct.

We have shown that these optimizations achieve important improvements in terms of speedup (7.1% on average; up to 34% for some programs) and energy consumption (6.5% on average) for optimized JavaScript code.

Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness under grants TIN2010-18368 and TIN2013-44375-R and the Spanish Ministry of Education, Culture and Sport under grant FPU12/05670.

References

[1] C. Chambers and D. Ungar, 1989. Customization: optimizing compiler technology for Self, a dynamically-typed object-orientied programming language. In Proceedings of the SIGPLAN'89.

[2] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. PLDI, 2009.

[3] C. Anderson. Type Inference for JavaScript. PhD thesis, Department of Computing, Imperial College London, March 2006.

[4] C. Anderson and P. Giannini. Type checking for JavaScript. Electr. Notes Theor. Comput. Sci., 138(2), 2005.

[5] D. Ungar and R. B. Smith. Self: The power of simplicity. In Proceedings OOPSLA '87.

[6] Andy Wingo. inside full-codegen, v8's baseline compiler. http://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler.

[7] M. Chevalier-Boisvert and M. Feeley. Simple and effective type check removal through lazy basic block versioning. In Proceedings of the 2015 European Conference on Object-Oriented Programming (ECOOP). LIPIcs, 2015.

[8] ECMA 2011. ECMAScript Language Speci cation – Fifth Edition. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf.

[9] F. Schneider, High performance JavaScript with V8, 2012. http://cs.au.dk/~jmi/VM/IC-V8.pdf

[10] G. Dot, A. Martínez, A. González, "Analysis and optimization of engines for dynamically typed languages", Proc. Of the 27th Int. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE (ISSN 1550-6533), Florionopolis (Brasil), October 2015, pág. 41-48.

[11] Google. V8 JavaScript engine, 2009. http://code.google.com/p/V8/.

[12] U. Holzle, Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming, PhD dissertation, Stanford Univ., Stanford, Calif., 1994.

[13] U. Holzle and D. Ungar. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In Conference on Programming language Design and Implementation (PLDI), 1994.

[14] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas. Improving JavaScript performance by deconstructing the type system. In PLDI, 2014.

[15] M. S. Ager, V8 internals,2009. https://dl.google.com/io/2009/pres/W_1230_V8BuildingaHighPerformanceJavaScriptEngine.pdf

[16] M. S. Ager, The V8 JavasCript engine, 2011. http://websrv0a.sdu.dk/ups/SCM/slides/lecture_03_mads_ager.pdf

[17] U. Hölzle , C. Chambers , D. Ungar, Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, Proceedings of the European Conference on Object-Oriented Programming, p.21-38, July 15-19.

[18] E. Lee. Object Storage and Inheritance for SELF, a Prototype-Based Object-Oriented Programming Language. Engineer's thesis, Stanford University, 1988.

[19] Mozilla. SpiderMonkey javascript engine. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[20] O. Anderson, E. Fortuna, L. Ceze, S. Eggers. Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors. Computer Science and Engineering, University of Washington, 2011.

[21] L. P. Deutsch and A. Schiffman, Efficient Implementation of the Smalltalk-80 System. Proceedings of the 11th Symposium on the Principles of Programming Languages, Salt Lake City, UT. 1984.

[22] D.-M. Ungar. The Design and Evaluation of a High-Performance Smalltalk System. Ph.D. dissertation, the University of California at Berkeley, Feb., 1986. MIT Press, Cambridge, MA, 1987.

[23] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. HP Laboratories, April 2008.

[24] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In Conference on Programming language Design and Implementation (PLDI), 2010.

[25] A. Patel, F. Afram, S. Chen, K. Ghose, MARSS: a full system simulator for multicore x86 CPUs, Proceedings of the 48th Design Automation Conference, June 05-10, 2011, San Diego, California

[26] WebKit. Introducing SquirrelFish Extreme. http://webkit.org/blog/214/introducing-squirrelfish-xtreme/, 2008.

[27] WebKit. SunSpider JavaScript Benchmark. http://webkit.org/perf/SunSpider-0.9/SunSpider.html , 2008.

[28] White Paper: Intel® Next Generation Microarchitecture (Nehalem), 2008.

[29] I. R. de Assis Costa, H. N. Santos, P. R. Alves, F. M. Quintão Pereira. Just-in-Time value specialization.

Department of Computer Science, Federal University of Minas Gerais (UFMG), Brazil. In Proceedings CGO 2013.

[30] Google Inc. Octane. https://developers.google.com/octane, 2013.

[31] Mozilla. Kraken. https://krakenbenchmark.mozilla.org, 2013.

[32] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M.Tullsen, and N. P. Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. ACM Transactions on Architecture and Code Optimization (TACO), 10(1):5, Apr. 2013.

[33] Tiobe programming community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[34] Chakra’s Technical review. http://blogs.msdn.com/b/ie/archive/2014/10/09/announcing-key-advances-to-javascript-performance-in-windows-10-technical-preview.aspx

[35] Andy Wingo. v8: a tale of two compilers. http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers , 2011.

[36] Andy Wingo. a closer look at crankshaft,v8's optimizing compiler. http://wingolog.org/archives/2011/08/02/a-closer-look-at-crankshaft-v8s-optimizing-compiler,2011.