Co-designed Solutions for Overhead Removal in Dynamically Typed Languages

Gem Dot Artigas
Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors:

Alejandro Martínez
ARM
Antonio González
Universitat Politècnica de Catalunya

A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

ACKNOWLEDGMENTS

Llegados a este punto toca poner la vista atrás y poder agradecer y dedicar este trabajo a todas las personas que han estado a tu lado a lo largo de este largo camino y también a las que se han unido en el transcurso de éste. Para ello agradezco primero a mi madre, por haber estado ahí en los momentos más duros; a mi padre, por haberme enseñado como se afrontan los momentos complicados de esta vida; a mis abuelas, por haber hecho de abuelas; a mis abuelos; a la Carme; a mis dos hermanos Nil y Blai y a mi hermana Ruth; y a mis primos y tíos. Quisiera agradecer mucho a mis directores de esta tesis, Alejandro y Antonio, por haber sido mis mentores en este largo camino y haberme dado la oportunidad de poder aprender de ellos. Muy muy importante para mí es agradecer también a mis amigos-hermanos del vallés accidental, especialmente a Ion, Pacheco, Anto, Chino, Mora, Chicho, Amanda, Pelli, Killo, Pau, Taba, Pol, Dani, David, Vakero, Ivan, Mella, Neus, Lluís, Artús, Roura, Roger, Kike así como también el resto de amigos del vallés. Agradezco mucho a la gente de mi segundo pueblo llamado Colera, especialmente a Antonio, por todo lo que me has enseñado, y a la familia y amigos que durante estos últimos 10 años han pasado por el restaurante La Gambina: Joelle, Xavi, Julie, Jose Luis, Jose, Ferran, Helena, Jordi, Victor, Roxana, Jadilla, Maite, Oriol, Oriol y Carlitos. También agradecer a los del restaurante Mont-Mercé. Agradezco mucho también a mis amigos de la carrera, los ingenieros anónimos, los cuáles estoy muy contento de seguir conservando amistades tan grandes. Agradezco muchísimo mis compañeros y excompañeros del grupo de investigación ARCO los cuales son grandes personas e investigadores. Agradezco a todas las grandes personas y amigos de mis equipos de futbol de estos últimos años (a pesar que ya hace más de un año que no juegue): Los Voldamistas, Vilerpool y Vengalas, especialmente a Javi y a Manu. Que grandes partidos, donde darlo todo en el campo y luchar por cada balón como si nos fuera la vida, llueva o nieva, era lo más importante, como un reflejo de la vida misma. También agradecer a Mika, por haberme ayudado a encontrar mi piso actual; a Eric, por enseñarme a cocinar sushi; también quiero agradecer a Asier, por haber dirigido mi trabajo de fin de carrera; agradezco a Josep Josa, por haberme asesorado cuando empecé el máster; y a todas las demás personas que tenga algo también que agradecer y me haya olvidado de mencionar. Por último, dedico también esta tesis a todas las personas que luchan y nunca se rinden, a pesar de caerse innumerables veces.

This Thesis 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.

ABSTRACT

Dynamically typed languages are ubiquitous in today's applications. These languages ease the task of programmers but introduce significant runtime overheads since variables are neither declared nor bound to a particular type. For efficiency reasons, the code generated at runtime is specialized for certain data types, so the types of variables require to be constantly validated. However, these specialization techniques still carry important overheads, which can adopt different forms depending on the kind of applications. This thesis proposes three hybrid HW/SW mechanisms that reduce these different forms of overhead.

The first two mechanisms target the overhead produced during the execution of the specialized code, which is characterized by the frequent execution of checking operations that are used to verify some assumptions about the object types. The first technique improves the performance by reducing the number of instructions used to perform these checks. The second technique is based on a novel dynamic type-profiling scheme that removes most of these checks.

The third technique targets the overhead due to the execution of the non-optimized code, which performs an important amount of profiling for future optimizations. We present a hybrid HW/SW mechanism that reduces the cost of computing the addresses of object properties in a very efficient manner. This is an innovative approach that significantly improves the speculative strategy currently adopted by state-of-the-art dynamic compilers.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Scripting languages have become very popular in the recent years [56]. These languages are often dynamically typed languages, which provide a higher flexibility and allow a faster application development compared to other traditional statically typed languages, such as C, C++ or Java. JavaScript [18] is the most popular one; Python, PHP, Ruby, Smalltalk and Self are other commonly used dynamically typed languages. Initially, these languages were designed for connecting different system components, which were written in traditional languages [35][57]. The reason for this is the higher flexibility that the dynamic typing provides for gluing tasks, which would require a more complex and longer-term task in statically typed languages. However, in the last years, scripting languages have gained popularity and have also been used to construct entire applications from scratch [35]. This is due to different factors:

- The increasing demand of web applications, where these languages require different components to work together.
- The importance of graphical user interfaces, which are used to connect graphical controls and the internal program functionality.
- Modern scripting languages are executed in complex virtual machines, which make use of Just-In-Time (JIT) compilation techniques to improve the performance of the code.
- Scripting languages are easier to learn for non-professional programmers than traditional static typed languages, due to their lower complexity. Non-professional programmers represent an important percentage in today's programmer community.

On the other hand, applications written in dynamically typed languages are less efficient than applications written in statically typed languages. The reason for this is because the types of variables in these applications are not known at compile time and therefore, these types need to be checked at run-time. This thesis focuses on proposing new dynamic compilation techniques for dynamically typed languages based on hybrid HW/SW support.

1.1 Dynamically Typed Languages

In dynamically typed programming languages, types are checked at run-time since variables are neither declared nor bound to a particular type, and their types change during the execution. In addition, the most popular dynamically typed programming languages are also object-oriented languages. In these cases, objects can change their class dynamically and therefore, the lookup of their methods and attributes (i.e. the properties of an object) are performed at run-time. This is also known as late binding.

Traditionally, dynamically typed languages used to be interpreted because a static compilation cannot benefit from the runtime information, which is necessary to perform the type checks and the late binding of the object methods. However, interpretation introduces a high overhead to their execution. These factors penalized the applications written in dynamically typed languages, in comparison with the execution of the same applications written in statically typed languages. In order to reduce this performance gap, modern virtual machines for dynamically typed languages combine both interpretation and Just-In-Time compilation techniques, with the support of some kind of dynamic profiling to produce specialized code.

1.2 Overheads in Dynamically Typed Languages

The performance of the applications developed in these languages depends on two kinds of overheads: the overheads associated to the virtual machines, and the overheads due to type checks and late binding in the generated code.

The former overhead is caused by the time spent in dynamic compilation, interpretation, garbage collection and other housekeeping tasks. Complex virtual machines adopt different strategies that combine JIT compilation and interpretation techniques, in order to focus the efforts in the most executed code regions.

The latter overhead (i.e. the overheads belonging to the type checks and late binding) has been an important focus of the research community. Most proposals consist of collecting dynamic information about types, in order to produce specialized code for these types. Although the specialized code is more efficient than a more general version, it still incorporates an important amount of overhead, which is mainly due to the execution of checking operations that verify the assumptions about the types. Often, these checking operations follow the same pattern of instructions, which basically are composed of an arithmetic and a branch instruction. Moreover, there are other frequently executed patterns of instructions, which are composed of more than one kind of checking operations. Other proposals focus on type inference techniques, which are based on the deduction of types at compile time.

The execution of non-optimized code is rather slow, which has an important performance impact for short applications common in some web sites. This penalty mainly comes from object property lookup operations and profiling activity that is necessary for the compilation of the optimized code.

1.3 Contributions

In this thesis we propose different HW/SW techniques that target the overheads due to type checks and late binding. We use JavaScript [18] as the experimental platform to demonstrate the benefits of the techniques, more concretely, the JavaScript engine from Google, known as V8 [28]. As a first step, we perform a detailed analysis of common JavaScript applications. Below we described in more detail these contributions.

1.3.1 Analysis of Overhead

We perform a detailed analysis that characterizes the contribution to the execution time of the different components of V8 [22][23]. We consider the execution of both the first phases of JavaScript applications and the steady state of these applications. We quantify the overhead produced by the dynamic type profiling and code specialization techniques. This analysis has served as a guide for the techniques proposed later in this dissertation.

1.3.2 Fusion of Common Instruction Patterns

The checking operations used to preserve some assumptions about types in the optimized code often use the same pattern of instructions. When these assumptions are not fulfilled, the code

branches to a deoptimization procedure. However, these assumptions are rarely not met. Taking account this consideration, we optimize the pattern of checking operations by proposing a novel exception mechanism that removes the branch instructions used to perform these type checks [22][23]. Moreover, two new optimizations are presented, which reduce the dynamic instruction count of other frequently executed instruction patterns.

1.3.3 The Class Cache

When the checking operations target *monomorphic* object variables (i.e., object properties or elements from an array that only have one single type during the execution), their execution is not necessary. In this regard, we have proposed a technique that completely removes some of these checking operations, which improves the execution of optimized code. This consists of a HW/SW mechanism based on a novel dynamic type-profiling scheme that identifies *monomorphic* object variables. Then, the application code is recompiled in a way that the type checks that target these monomorphic variables are completely removed, and an exception mechanism is triggered when this assumption is not met [24].

1.3.4 The Property Cache

This technique reduces the overhead related to the late binding of object properties. Moreover, this technique targets both non-optimized and optimized codes. For non-optimized code, all the operations related to the lookup and profiling of object property accesses are substantially optimized. On the other hand, most of the type checks that verify type assumptions before accessing object properties in the optimized code are also removed. This technique is based on a hybrid HW/SW mechanism that provides the information required to identify the addresses of object properties in a very efficient manner [25].

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 describes the most relevant related work about on the techniques that deal with the overheads described earlier. Chapter 3 provides some background to help understand the techniques that we will present later. Chapter 4 describes the simulation tools used to evaluate the proposed mechanisms. Chapter 5 presents the analysis of overhead of dynamic typed languages, which is used as the motivation for the proposed techniques. Chapter 6 explains our proposal of the fusion of pattern instructions in

the optimized code. Chapter 7 presents our proposal called The Class Cache mechanism. Chapter 8 presents the Property Cache mechanism. Finally, Chapter 9 concludes this dissertation.

Chapter 2

Related Work

The reduction of the overheads of dynamically typed languages has been an important topic for the research community, due to the booming of web scripting applications in recent years, including proposals based on parallelization techniques. In this chapter, we review the state-of-the-art techniques to improve the performance of dynamically typed languages.

2.1 Techniques to Reduce the Overhead Produced by Dynamic Typing

These techniques are divided in two different families: type inferring techniques and type feedback techniques. The latter are normally more effective due to two main reasons. On the one hand, type inference requires a significant amount of computation to deduce all the application types, which is an important drawback for these languages that are dynamically compiled and thus, the compilation time becomes critical. On the other hand, most of the types cannot be deduced at compile time, due to the dynamic typing nature of these languages.

Although modern virtual machines [9][28][45][63] combine both kind of techniques, the type feedback approach represents the main component of the strategy followed by these engines to reduce the overhead. In this regard, type feedback is applied at the beginning of the execution of the application, in order to collect the information necessary to specialize the hottest regions of code [41]. Once the code is specialized, a type inference pass is performed to eliminate unnecessary type checks and to specialize even more the code. Therefore, type feedback techniques introduce less initial overhead and collect more type information, whereas type inferring efforts mainly focus on hot specialized code, which is more deductible.

2.1.1 Type Feedback Proposals

A significant number of works target type feedback techniques for dynamically typed languages, due to their important role in modern virtual machines. In this section the most important approaches of type feedback techniques are presented.

2.1.1.1 Inline Caching

The state-of-the-art technique used by current JavaScript virtual machines [9][28][45][63] to address the overhead of object property accesses (i.e. accesses to an attribute or method of an object) due to the runtime binding problem is known as Inline Caching [40][15]. It consists of generating type specialized code for the accesses to properties and other program variables that have been previously seen. Next time a given property of a particular object type is accessed, this code is used to access the property in a more efficient manner.

The first work [40] to introduce the Inline Caching technique targeted Smalltalk compilers. Other works [59][60][61] have improved this technique for Self [17][20] compilers. Hölzle et al. [59] extends this technique to polymorphic Inline Caching, which extends Inline Caches to more than one object type. Hölzle and Ungar [60] propose a dynamic recompilation of hot functions that uses the type information previously collected by the Inline Caches to produce more efficient specialized code for the whole function.

Recently, some other techniques to improve the performance of Inline Caching have been proposed [25][51][54][62]. Ahn et al. [62] presents a new scheme that reduces the miss ratio of Inline Caching and optimizes polymorphic Inline Caching for real-web applications. Li et al. [51] propose a new mechanism similar to Inline Caching, in order to access the object properties without incurring the overhead produced by the code generation. It is based on a software structure that keeps the information corresponding to property accesses produced in every location of the source code. When an object property is accessed in line *i* of the source code, the *i-th* position of this structure is accessed and the necessary information to perform the access (i.e. the address of the property) is obtained.

2.1.1.2 Trace Based

In a trace based approach [1][7][42], cyclic regions of code called traces are dynamically recorded in initial runs of the application. At the same time, the specific types used in these

traces are also profiled. Then, when a trace becomes hot, its corresponding code is recompiled and speculatively specialized with the profiled types and predicted branches, which results in a more efficient code. These assumptions about types and taken branches are verified with checks and when these conditions are not fulfilled, the execution exits the trace. When a trace is exited, a new trace may be recorded and recompiled with the alternative path or type, forming a trace tree.

2.1.1.3 Customized Compilation

Customized compilation techniques [8][12] are based on dynamically compiling functions according to the types of their arguments when these functions are called for the first time. In this regard, multiple versions of the same function can be compiled, which takes more memory space and compilation time. However, the advantage of these techniques is the generation of more type specialized code for the compiled functions, which results in a better performance.

2.1.1.4 Other Complementary Works

Other works propose improvements that can be complementary to the approaches described in the above sections. Driesen [39] proposes a space-efficient technique for object method lookups in dynamically typed languages. Other works focus on reducing the overhead produced by type checks [22][23][24][47][49]. Anderson et al. [47] introduces automatic checking of types, which is performed implicitly by a dedicated hardware.

2.1.2 Type Inferring Proposals

Type inferring techniques for dynamically typed languages [10][11][37][43][48][50][52] are based on statically analyzing the applications, in order to ensure type safety. These techniques allow for the early detection of type errors, such as accesses to non-existing members of objects or incorrect type conversions. In this regard, the applications that contain errors are rejected before executing them. Basically, these works provide a type system that defines some constraints to represent the relationships between types. Then, an algorithm uses these constraints to infer the types of the application. When these constraints are violated, runtime errors are signaled.

2.1.3 Value Specialization Proposals

Costa et al. [32] proposes a technique to dynamically specialize the values of function parameters. When a function is called for the first time, the values of the parameters are collected. If these arguments remain unchanged between calls, then the function is recompiled and its arguments are replaced by the collected values. This allows to apply classic optimizations for the recompiled functions, such as constant propagation, dead-code elimination, array bounds check elimination and function inlining, which further improve performance.

2.1.4 Hybrid Proposals

Other works combine both type inference and type feedback approaches, in order to take benefit from the synergy between them [6][41]. Hacket and Guo [6] propose a hybrid type inference algorithm that performs an analysis of the application before its execution, in order to make assumptions about types. These assumptions are guided by some rules, which are based on the effect that operations have on their produced values. However, these assumptions are not guaranteed to be correct during the execution of the application and therefore, runtime checks are required. This hybrid mechanism is faster and more precise than pure static inference algorithms, which cannot perform assumptions about types.

On the other hand, Kedlaya et al. [41] propose an initial type inference step before the execution of the application, in order to reduce the profiling overhead produced by the type feedback step. As a result, some type profiling activity becomes unnecessary because the type inference pass has already deduced the type.

2.2 Parallelization Techniques

Although most of the research efforts in dynamically typed languages focus on reducing the overhead of type checks and late binding, some other recent works propose different techniques to introduce some kind of parallelization. Traditionally, virtual machines for these languages do not exploit parallelism, despite the fact that these applications are often executed in parallel hardware platforms. In addition, some studies [19] show that current applications written in these languages are well-suited for parallelization.

These works are basically divided into implicit and explicit parallelization support. The former is based on dynamically identifying code regions that are parallelizable and speculatively execute them on different threads. There are proposals that parallelize loops [44][66], whereas others parallelize function calls [33][34]. Moreover, some modern virtual machines for dynamically typed languages execute application code and compilation tasks in parallel, as it is the case of V8 JavaScript engine [28]. On the other hand, works based on explicit parallelism extend the APIs of these languages to provide parallel semantics to the programmer [13][30][31][38].

Chapter 3

Background

In this chapter, we present the main characteristics of JavaScript and the V8 JavaScript engine from Google [28], which is an open source and widely used dynamic compiler for JavaScript. In this thesis, V8 (64 bits) has been used as part of the experimental platform. Nevertheless, the basic techniques used by this dynamic compiler are also adopted by other modern virtual machines, such as Nitro from Apple (previously known as SquirrelFish Extreme [63]), SpiderMonkey from Mozilla [45] and Chakra from Microsoft [9]. Therefore, although the techniques presented in this dissertation are evaluated for a V8 JavaScript environment, they can be extended to other engines for dynamically typed languages, as long as they use similar approaches to deal with object typing.

3.1 JavaScript

HTML5 has improved not only the design of attractive websites (CSS style sheets, SVG images, and video), but also has succeeded in creating web applications with performance comparable to desktop applications. To achieve this, it uses JavaScript [18], which is a dynamically typed programming language embedded into web pages that allows the creation of sophisticated solutions in the client-side web.

JavaScript provides a small set of data types (e.g. Boolean, String, Object, etc.), some built-in objects and functions and an inheritance mechanism based on prototype objects. In addition, JavaScript has access to its host environment through the Document Object Model (DOM), which is an API that allows JavaScript to interact dynamically with web pages.

In JavaScript, the structure of an object is defined by its ordered set of named variables (i.e. variables that are referenced by name) and methods. Objects that have the same structure are considered of the same type. For the rest of this dissertation, the term property of an object refers to any of its named variables or methods. Moreover, JavaScript uses an inheritance

mechanism to share properties among different objects. This mechanism is implemented through what is called prototype objects. Each object *x* has an associated prototype, which is another object that contains a set of additional properties that the object *x* can access. In JavaScript all objects have a property that point to its prototype (which can be null in some cases), thereby forming a prototype chain (see next paragraph for more detail). Therefore, when accessing a property of an object, the entire prototype chain needs to be searched. Furthermore, when a constructor object creates an object, the new object inherits the prototype of the constructor. Some object variables are accesses by a number like conventional arrays elements. However, these numbered variables do not affect the object structure because they are considered as conventional elements of a special array that belongs to the object.

In Figure 3.1, objects *o* and *k* have been created by the object constructor *c*. Therefore *o* and *k* inherit the prototype of the constructor *c*, which is *p*. In this way, properties *f* and *g* from the prototype *p* can also be accessed from *o* and *k* in addition to *c*. In other words, these are properties shared by these three objects.

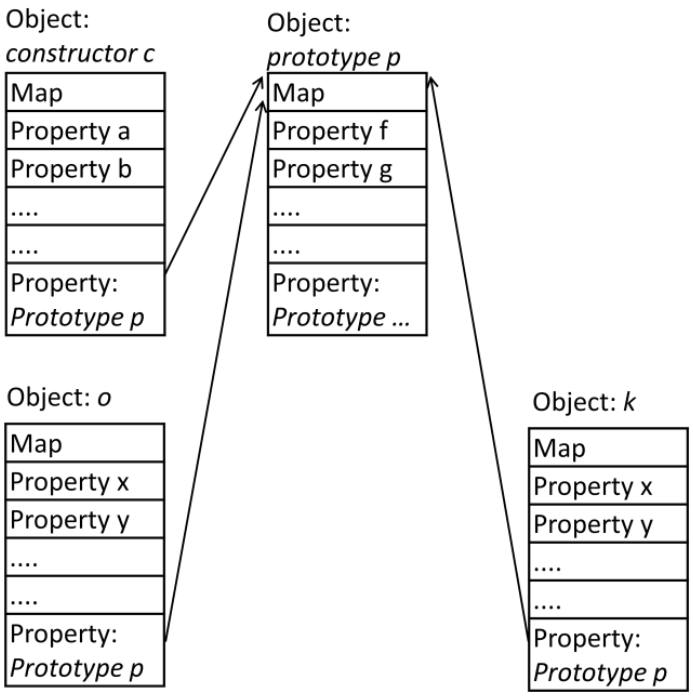

Figure 3.1: Prototype chain scheme.

Dynamically-typed languages like JavaScript increase programmer productivity but present important inefficiencies compared to statically-typed languages. This is mainly due to the fact that compilers cannot determine the type of the objects that will be accessed at runtime. One of the major overheads is when the value of an object property has to be loaded. In this

scenario, we need to first know the address in memory of this property (i.e. the offset of this property with respect to the address of the object), which entails a costly sequence of steps: first of all, the type of the accessed object has to be obtained, then the property has to be found in the type descriptor, which contains the offset for all its properties, and finally this offset is used to obtain the memory address where the value is stored.

3.2 The V8 JavaScript Engine

V8 was specifically designed for fast execution of large JavaScript applications. 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 focuses on optimizing hot functions (i.e. those functions that execute very often). V8 integrates two compilers, one that has light overhead and produces generic code (Full Codegen); and another that is heavier but generates more optimized code (Crankshaft) [3][4][5]. When a new function is encountered, it is first compiled by Full Codegen just before its execution, instead of being interpreted. After a while, if the function becomes hot, then it is compiled by Crankshaft.

Inline Caching [15][40] is applied by both compilers, despite the fact that the dynamic profiling of the code is only performed during the execution of the non-optimized code (i.e., the generic code produced by Full Codegen).

3.2.1 Hidden Classes

JavaScript is an object-oriented programming language without explicitly declared classes. However, V8 uses Hidden Classes to represent object types (i.e. an ordered collection of properties). All objects built by the same function constructor share the same Hidden Class. In other words, objects that share the same Hidden Class have the same type. When a function constructor at runtime creates an object 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 and the property *x*. If this second 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.

In Figure 3.2, there is an example: object *v* belongs to Hidden Class *Vector*. It also contains a property, *x*, which belongs to Hidden Class *Double*. Note that the first field of each

object (called *Map*) contains the address of the Hidden Class descriptor. In V8, this address is also used as identifier for the Hidden Class, which is called the *Hidden Class identifier*. For the rest of this thesis, we use the terms Hidden Class and type of an object indistinctly.

Note also that the prototype property of an object is kept in its Hidden Class descriptor, instead of the object itself, as we can see in Figure 3.2. Therefore, when the prototype property of an object is overwritten, the Hidden Class of that object also changes, as Hidden Classes are immutable data structures.

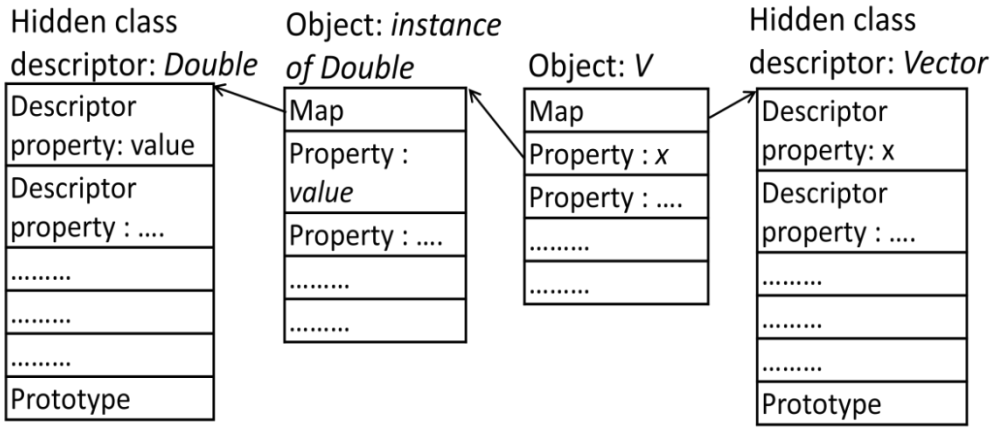

Figure 3.2: Example of two objects and their corresponding Hidden Classes (Double and Vector).

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, as explained in section 3.1. 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.

Occasionally, when the number of properties (i.e., named variables or methods) of an object exceeds a particular threshold (128 properties), their properties are stored in a separate dictionary-style structure called the *property dictionary collection*. When this happens, the subsequent additions of new properties for the object will not change its Hidden Class, because its structure keeps being the same. In addition, the object will contain another special property called the *property pointer*, which keeps the address to its *property dictionary collection*. Note that the *property pointer* is always located in the second position of the object. This is an

optimization performed by V8, in order to avoid an explosive number of Hidden Class creations for objects that are used as dictionaries.

Note that all objects in V8 are represented by their address. When they are stored in a register, its least-significant bit is set to *1*. Therefore, before a particular object is accessed, this bit has to be cleared, in order to obtain the address. As exception, small integers (SMIs) that do not need more than 32 bits for its representation are directly stored in registers, in the 32 most significant bits, and the least-significant bit is set to *0*, to indicate that the register contains a SMI, instead of an object address.

3.2.2 Inline Caching in V8

Inline Caching 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. Full Codegen and Crankshaft apply this technique in a different manner, as described below for loads or stores to object properties, which is the most common scenario for this technique. Inline Caching is also applied for loads and stores to object array elements (i.e. object variables that are referenced by number), method invocations, arithmetic operations, boolean operations, and other binary operations, in a similar 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 type check needs to be inserted before the generated code to verify that the type is the expected one. Figure 3.3 illustrates the use of Inline Caching for loads. In Figure 3.3a, we show the scenario for the first time an object property load is performed. In this case, it is executed by the lookup routine, which is rather costly. Subsequent accesses are executed by the optimized Inline Caches, as we can see in Figure 3.3b.

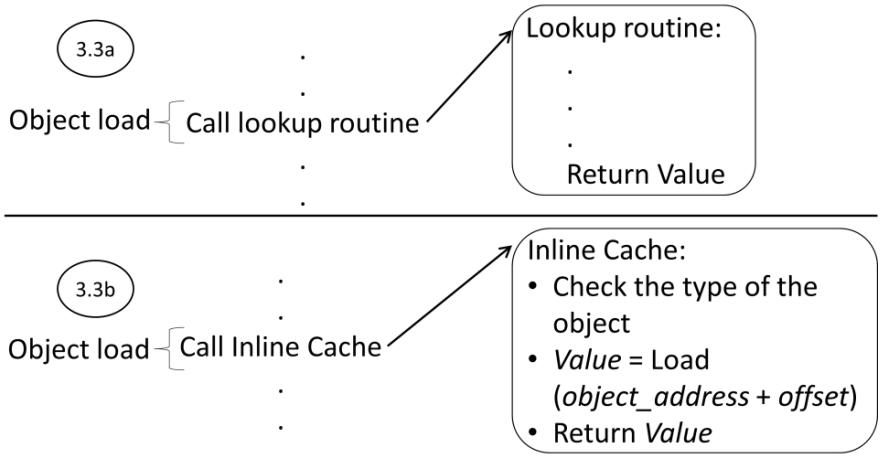

Figure 3.3: Basic Inline Caching process.

The information recorded during this 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, type checks are introduced in this specialized code, in order to verify that the encountered type is the expected one; otherwise (i.e., when a type check 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.

3.2.3 Full Codegen Compiler

This compiler takes as input the abstract syntax tree (AST) of a function, walks over the nodes in the AST, and emits calls to a macroassembler. The code produced is generic native code, for which only the inline caching optimization is performed. Figure 3.4 outlines the flow of data at the compilation process.

Furthermore, Full Codegen does not store local variables in registers; instead these are stored either on the stack or on the heap. All variables stored on the heap belong to objects contexts, each one associated to a different function. When the value of a local variable is needed, the compiler emits a load to pull the value into a register [5].

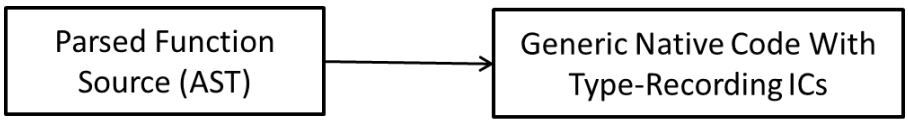

Figure 3.4: Full codegen compilation process.

3.2.4 Crankshaft Compiler

Once V8 has identified that a function is hot (by profiling) and has collected some type information through the Inline Caches, compiles that function through the optimizing compiler (Crankshaft). As we can see in Figure 3.5, the process of Crankshaft is more sophisticated than Full Codegen [4].

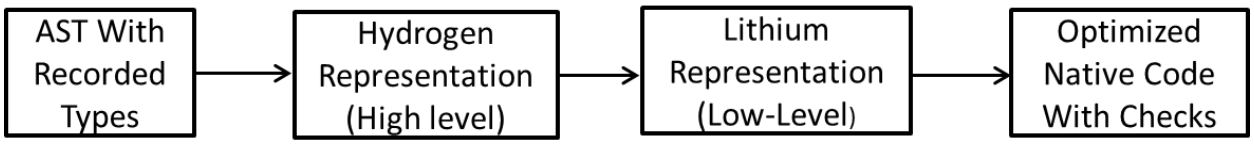

Figure 3.5: Crankshaft compilation process.

Crankshaft first translates the JavaScript AST to a high-level static single-assignment (SSA) Intermediate Representation, which is called Hydrogen. In this part of the process, the compiler specializes the code according to the information collected (i.e. Inline Caching technique) and tries to apply other high-level optimizations. Then, the Hydrogen code is translated to a machine-specific low-level Intermediate Representation, which is called Lithium. This representation facilitates other machine-specific optimizations. Finally, Register Allocation and Code Generation are performed.

3.2.4.1 High Level Optimizations

Other high level optimizations mentioned above are:

- **Mark Dead Subgraphs:** Regions of code without Inline Cached types means that some paths in the original function may have never been executed. This optimization avoids optimizing these blocks and wasting compilation time.
- **Redundant/Dead Phi Elimination:** The AST to SSA Hydrogen translation process handles Phi instruction insertions [48]. This optimization eliminates Phi instructions that are not needed, either because they are redundant (consequence of a dead subgraph) or because they do not have real uses.

- **Representation Inference:** Numbers need also a mechanism to indicate that they are numbers (i.e. integers or doubles), before manipulating them. In this regard, numbers are represented in a similar manner than the rest of object types, which is called boxed representation. When a boxed number is needed to execute any operation, it is necessary a previous unboxing process to obtain the value of the number. Therefore, the manipulation of objects in a boxed representation is costlier than the direct manipulation of the value. The goal of this optimization is to represent temporary variables as integers or doubles, whenever possible, instead of a boxed representation. Inference techniques are used to deduce the type of these temporaries.

- **Range Analysis:** This optimization tries to determine the range of some values. It allows asserting various properties that influence code generation, such as the lack of overflow or the lack of negative zero values.

- **Type Inference and Canonicalization:** Type Inference can help to eliminate runtime checks. After this, each instruction is canonicalized (i.e. elimination of unnecessary operations) to eliminate other useless checks.

- **Stack Check Elimination:** Loops need to be interruptible. As a solution, a Stack checking operation is inserted at the beginning of every iteration. If the runtime wants to interrupt a loop, it resets the stack limit of the process to wait for the next stack check. This optimization eliminates some of these checks in case that a call dominates its backward branches, since all calls have a fixed stack check in the callee's prologue.

- **Global Value Numbering:** Other typical high level optimizations are implemented, such as Common Subexpressions Elimination and Loop Invariant Code Motion.

3.2.4.2 Additional Operations Performed in the Optimized Code

In this section, we present some operations performed during the execution of the optimized code that are not part of the JavaScript application. Instead, these additional instructions can be considered as overhead produced by Crankshaft, as a consequence of the dynamic

characteristics of the language.

Checking Operations

As we outlined above, type checks are introduced in the execution of optimized code to preserve assumptions about types. Moreover, other kinds of checks are also inserted at this optimization level, which are used for similar purposes. When a check fails, the optimized code falls back to non-optimized code through a deoptimization bailout, with the exception of *Check Stack*. In this latter case, the program is interrupted and another routine is executed, which handles an external exception. These checking operations are detailed below:

- **Check Maps**: These are the most commonly used type checks. The first slot in each V8 object points to its *Hidden Class identifier* (i.e. the object type). In this operation, the type of an object is checked to be the same as that of another recorded type, which has been seen before.

- **Check SMI**: A register containing a boxed object can be of two types: either a small integer (SMI), which has its last bit cleared or an object address, 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.

- **Check Instance Type**: It checks whether the kind of a particular instance is the expected one.

- **Check Function**: This is used to check whether an inlined function is the expected one.

- **Check Prototype Maps**: There are accessible object properties that reside in its prototype chain. Therefore, when a function call belonging to a prototype object is inlined, it is necessary to introduce a type check of that prototype.

- **Check Map Value**: It is like a *Check Maps*, but for enumerable objects in a for-in statement.

- **Check Bounds**: After obtaining the total length of the array, it is checked that the

accessed position of the array is not out of bounds.

- **Check Stack**: Inside a loop, the stack pointer is checked to see if it has been reset, in order to know whether an external exception has been produced.

Tagging/Untagging Operations

Other overhead instructions are tagging and untagging instructions, which are 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, the value is located in the 32 most significant bits of the register and the last bit is set to *0*. The specific tagging/untagging operations are detailed next:

- **Number Tag** (Non-SMI): This process consists of allocating in the heap an object structure that contains a number. The type (i.e. Hidden Class) of this object depends on the type of the number, such as Integer, Unsigned Integer or Double. Therefore, the resulting register contains the address of the allocated object.

- **SMI Tag**: This process consists in introducing the value in the 32 most significant bits of the resulting register. In addition, the last bit of the register is set to *0*.

- **Number Untag** (Non-SMI): The input register contains the address of an object that contains a number. Therefore, this operation obtains this number from this object, which is in heap memory.

- **SMI Untag**: Reverse process of tagging a SMI. Therefore, the resulting register contains the SMI value from the 32 most significant bits of the tagged input register.

Math Assumptions

There are 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. Note that the former is necessary because when the produced value is not a SMI (i.e., it needs more than 32 bits for its representation), this has to be boxed as a non-SMI. Note also that these situations rarely occur during the execution. Therefore, V8 assume

optimistically that they will never occur, in order to produce more efficient code (i.e., the generated code does not cover these alternative paths). However, these rare situations have to be detected when they occur. In this regard, V8 inserts additional instructions that verify that these math assumptions are correct. When any of these validations fails, the code is deoptimized.

3.2.5 V8 Dynamic Components

V8 is a dynamic compiler and therefore, program execution and code generation/optimization have to be efficiently synchronized in order not to affect responsiveness. Moreover, JavaScript is a managed memory language, which needs a mechanism to reclaim memory that will no longer be used. Taking into account these characteristics, the execution of an application in V8 can be broken down into the next components:

- **Non-optimizing compiler**: V8 runtime executing Full Codegen compiler.
- **Execution of non-optimized code**: Execution of application code produced by Full Codegen compiler.
- **V8 runtime**: The execution of various management routines.
- **Optimizing compiler**: V8 runtime executing Crankshaft compiler.
- **Execution of optimized code**: Execution of application code produced by Crankshaft compiler.
- **Garbage collector**: V8 runtime mechanism that reclaims memory used by objects that are no longer required.
- **Shared libraries**: Execution of auxiliary libraries.
- **Helpers**: Execution of helper code to carry out some aspects of the JavaScript engine, such as built-ins corresponding to JavaScript construct calls. Note that most of these helpers are executed during the execution of the non-optimized code, in order to perform tasks related to the management of the Inline Caching mechanism.

This classification will be used in the following chapters and we will refer it frequently.

Chapter 4

Experimental Framework

The V8 JavaScript engine described in the previous chapter is part of the experimental framework that we have used to demonstrate the ideas proposed in this dissertation. In this chapter, we describe the rest of tools and benchmark suites that have been used.

4.1 Tools

Pin [16] and V8's built-in sampling profiler [28] have been used for the analysis of the V8 JavaScript engine. Marss [2] and Sniper [58] micro-architectural simulators have provided the results regarding the speedups achieved by the proposed ideas. McPAT [53] and CACTI [55] power modeling tools have provided the energy savings of these proposals. We briefly describe these tools below.

4.1.1 Pin

Pin is a dynamic instruction instrumentation tool for the x86 instruction set architecture. It is based on inserting profiling code to the application binary. This instrumentation code collects run-time information about the instructions, such as their type, the type of their operands, number of operands, etc. In our experiments, we use Pin mainly to count the total number of dynamic instructions.

4.1.2 V8 Sampling Profiler

This is an internal tool of the V8 JavaScript engine, based on a sampler profiler that records the execution time spent in the different execution components of V8, which are described in section 3.3.5.

4.1.3 Sniper

Sniper is a timing multi-core simulator that uses a novel simulation technique called interval simulation [14], which allows nearly as much accuracy as a cycle-level approach while providing faster simulation speed. Interval simulation is based on partitioning the execution time into intervals, which are determined by the miss events. These events are modeled by an event based simulator. Then, the timing for each interval is derived by an analytical model, which also includes the penalties associated to the corresponding miss event. These penalties are determined by the type of the miss: I-cache miss, branch misspredict ion, long-latency load miss and resource stalls.

4.1.4 Marss

Marss is a cycle-level, full system, multicore simulator for the x86-64 ISA. This simulator is the result of the combination of two existing frameworks, which are Qemu and PTLsim. The former is a full system emulator that supports multiple ISAs and the latter is a cycle-level simulator. Marss provides a modular framework for the simulation of different cpu cores and cache memory models. Moreover, this modular framework allows the integration of other simulation tools, such as DRAMSim2, which is a DRAM simulator. The full system simulation includes the activity of operating systems, standard libraries and kernel interrupt handlers. It allows the evaluation of parallel applications and heterogeneous architectures, and eases the implementation of HW/SW co-designed techniques.

4.1.5 McPAT

McPAT is a multicore modeling framework that integrates power, area and timing. The power model includes both leakage and dynamic power consumption. The input for McPAT is a file that describes the dynamic events of a previous performance simulation, such as the number of cycles, number of L1 misses, number of branch misspredictions, etc. In addition, this file defines the architectural, circuit and technological parameters of the system. In general, any micro-arquitectural performance simulator can feeds the input for McPAT, as long as its output has the required format for McPAT.

4.1.6 CACTI

CACTI is an analytical model that estimates latency, power and area of memory units (e.g.,

cache memories). For caches, this tool takes as input the cache size, cache block size, cache associativity, technology, number of ports, and number of banks. The output is the optimal cache configuration for the input parameters and its associated latency, power and area.

4.1.7 Microarchitectural Configuration

For the experiments presented in this dissertation, the core configuration used by the simulators described above is shown in Table 4.1, which closely resembles a Nehalem core [65]. In this thesis, we use applications compiled for the x86-64 ISA.

Issue width	4
Instruction Issue queue	36 entries
Window size	128
Outstanding load/stores	10
Itlb	128 entries, 4-way
Dtlb	64 entries, 4-way
IL1 cache	32 KB, 4-way
DL1 cache	32 KB, 8-way
L2 cache	256 KB, 8-way
L3 cache	2MB, 16-way

Table 4.1: Microarchitectural configuration.

4.2 Benchmarks

We have used four common JavaScript benchmarks suites for our experiments: Octane [26][27], SunSpider [64], Kraken [46] and JSBench [29][36].

4.2.1 Octane

Octane is commonly used for evaluating JavaScript since it is representative of current workloads and execution profiles of real web applications. Octane's goal is to be a proxy for JavaScript applications like browser games, highly-interactive web pages and online productivity tools.

4.2.2 SunSpider

SunSpider consists of more than a dozen tests, each concentrating on a different part of the JavaScript language. Since SunSpider focuses on computation (does not include HTML, CSS, networking, etc.), its applications are micro-benchmarks that are very computational intensive.

4.2.3 Kraken

Kraken centers on four key areas of browser performance:

- **Audio**: These applications perform various audio functions, including fast fourier transforms, discrete fourier transforms, audio oscillator and beat frequency detection.
- **Image filtering**: These application use Pixastic Processing Library to desaturate an image, perform a Gaussian blur and other image manipulations.
- **JSON**: These applications test how quickly a browser can transmit data between two or more objects, which are usually a web server and a client.
- **Cryptography**: These applications use the Stanford JavaScript Crypto Library to perform four common cryptographic functions.

4.2.4 JSBench

JSBench are representative of user interactions from five representative websites: Google, Amazon, Facebook, Yahoo and Twitter. However, we have discarded Facebook and Yahoo benchmarks for our evaluations, because these applications cannot run without a browser.

Chapter 5

Analysis of Overhead

The performance of applications written in dynamically typed languages such as JavaScript changes with time. In this regard, at the beginning, the execution time is dominated by non-optimized code, interpretation, and compilation tasks, which result in low performance. Later, as the code becomes increasingly optimized, performance improves until no further optimization is possible, reaching a steady state. Note that some overhead remains in the steady state, which is mainly due to the verification of some assumptions made to generate the code.

Short-running JavaScript web applications for event-driven scripts are dominated by the execution of the non-optimized code, helper routines and runtime tasks (i.e., compilation tasks), because there is not time for much code specialization. On the other hand, long-running, sophisticated applications such as online image editors and games tend to execute repeatedly the same regions of code, which become very soon specialized. Therefore, the execution time is dominated by this optimized code.

Since there is a no consensus in the scientific community about which kind of applications will predominate in the future [44], this dissertation considers both short and long-running applications. Two of the techniques we propose target the execution of optimized code, whereas another technique targets both the execution of the non-optimized code and some runtime tasks.

In this section, we analyze the contribution of these execution components in the V8 JavaScript engine, and characterize the most important overheads.

5.1 Analysis of V8 Dynamic Components

Figures 5.1 and 5.2 show the contribution of the V8 dynamic components described in section 3.2.5 to the total run time of Octane, SunSpider and Kraken benchmarks. We consider both

steady state and first iteration (i.e. the first execution of the application). Figure 5.3 shows the same statistics for JSBench benchmark suite, regarding the fourth iteration of each application (i.e., the application is executed four times and the statistics are taken from the fourth iteration) [62]. We are most interested in the execution of the JavaScript code rather than other preliminary compilation tasks, such as the source code scanning process, which represents more than 50% of the dynamic instructions for the first iteration of these benchmarks. Steady state for JSBench is not analyzed because these benchmarks are not meant for modeling long-running complex applications. In all the cases, results were obtained through the V8's built-in sampling profiler [28].

Figure 5.1 shows that in steady state, most of the time is consumed in optimized code and only a very small fraction of time is dedicated to non-optimized code and helpers. In addition, V8 runtime tasks represent around 24% of the total time for Octane and Kraken and 14% for SunSpider. In Octane, the garbage collector is a very important contributor for *splay*, *typescript* and *earley-boyer* benchmarks, which make an intensive use of heap memory. Nevertheless, in this dissertation, we do not focus in the garbage collector because this is a widely investigated topic elsewhere. Note also that many benchmarks (for instance, *code-load*, *json-parse*, and *json-stringify* benchmarks) have little optimized code since they have limited code reuse.

Figure 5.2 shows that during the first iteration of Octane, SunSpider and Kraken, the execution time is dominated by V8 runtime tasks. SunSpider and Kraken have more optimized code than non-optimized code and helpers, which means that these are benchmarks that are dominated by hot loops. In contrast, the contribution of non-optimized code and helpers in Octane is higher than the optimized code. This correlates with the profile of JSBench (Figure 5.3), which hardly executes any optimized code. Therefore, Octane in the first iteration presents a similar behavior to real web applications.

5.2 Overheads in the Steady State

As detailed above, in the steady state of Octane, SunSpider and Kraken, the most important component is the optimized code. This subsection quantifies the contributions of the different kinds of overheads in optimized code. We achieve steady state by executing each benchmark ten times and taking statistics from the tenth iteration. We run each benchmark twice. In the

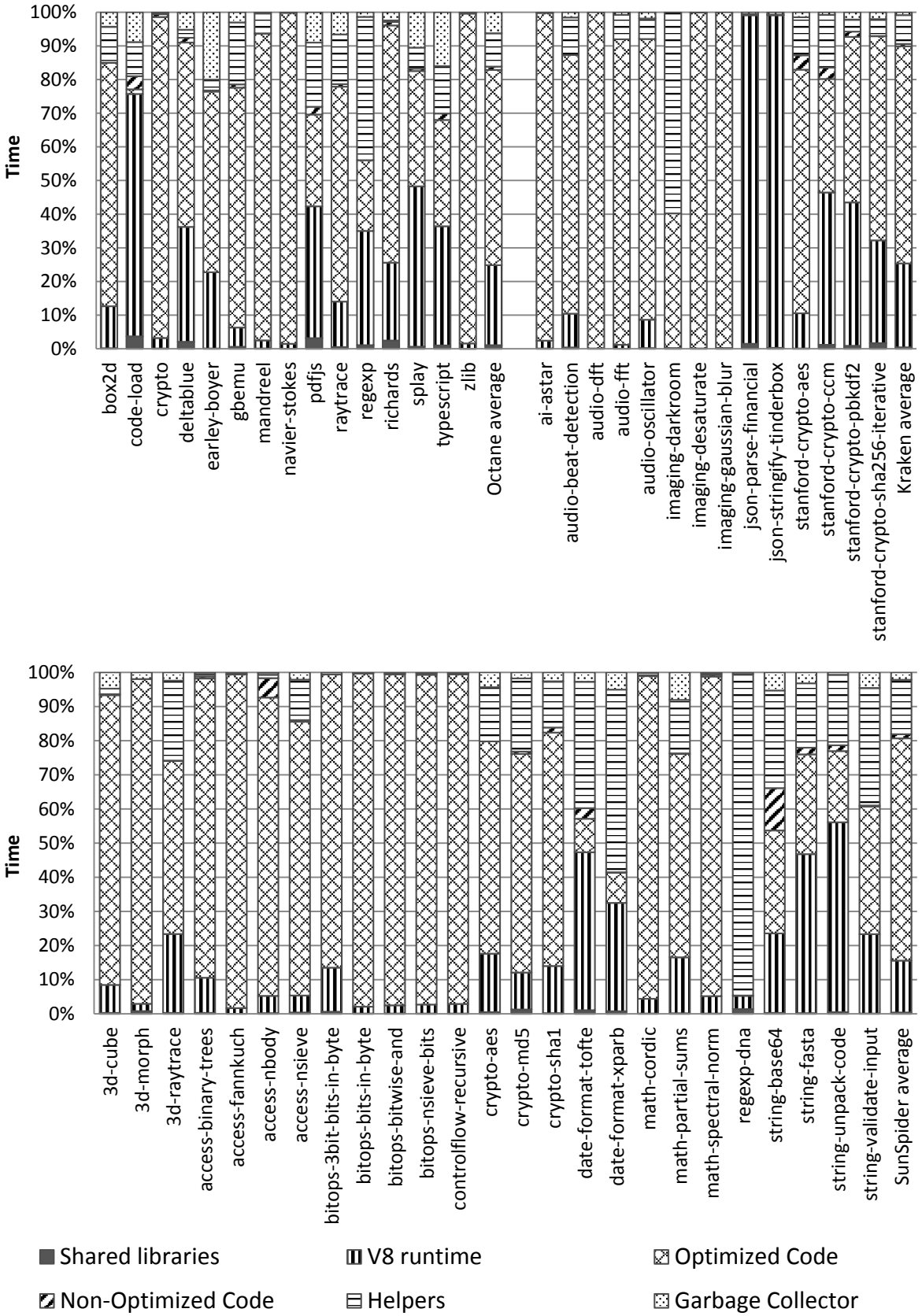

Figure 5.1: V8 engine components in steady state execution.

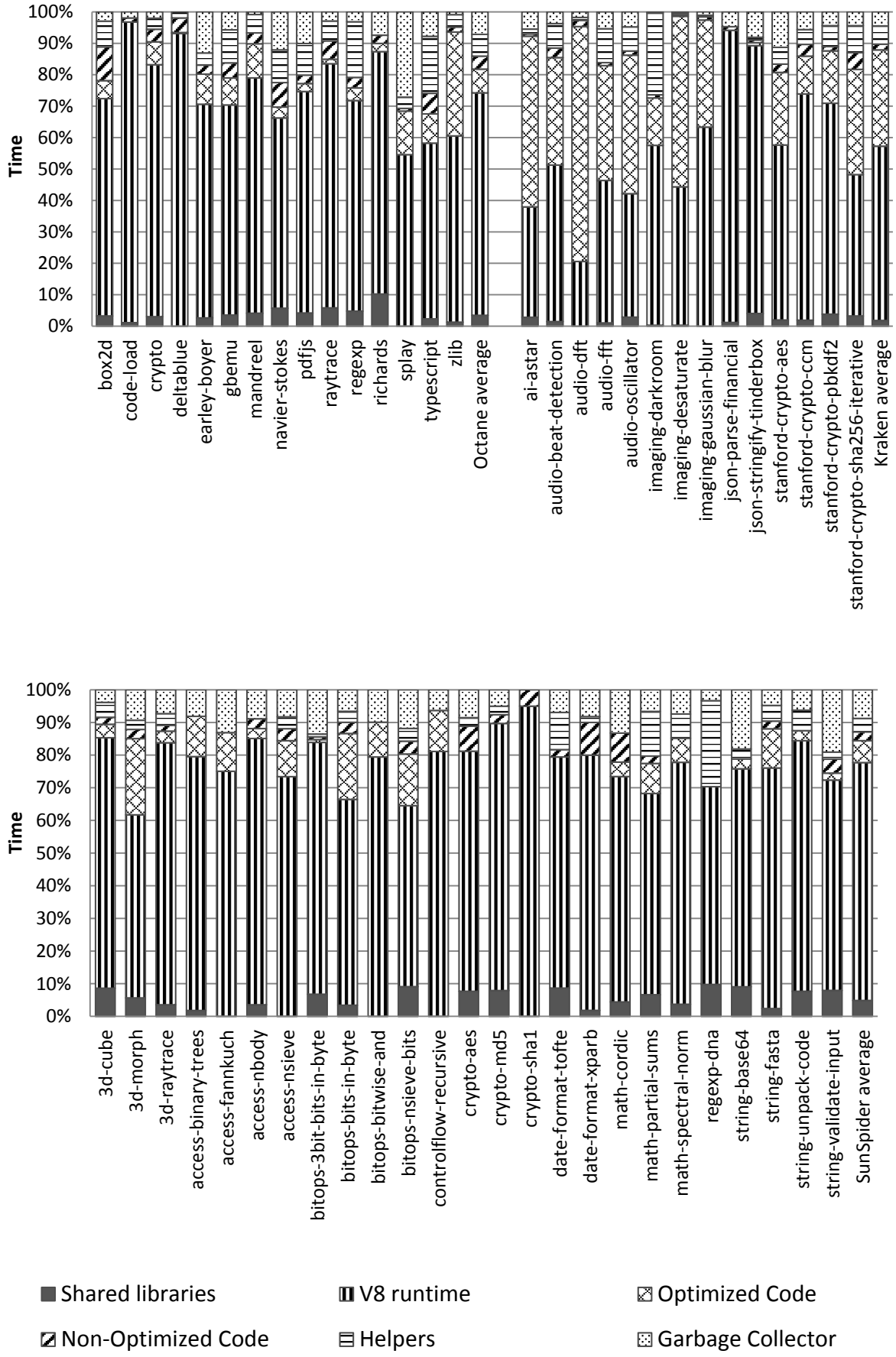

Figure 5.2: V8 execution breakdown in the first execution.

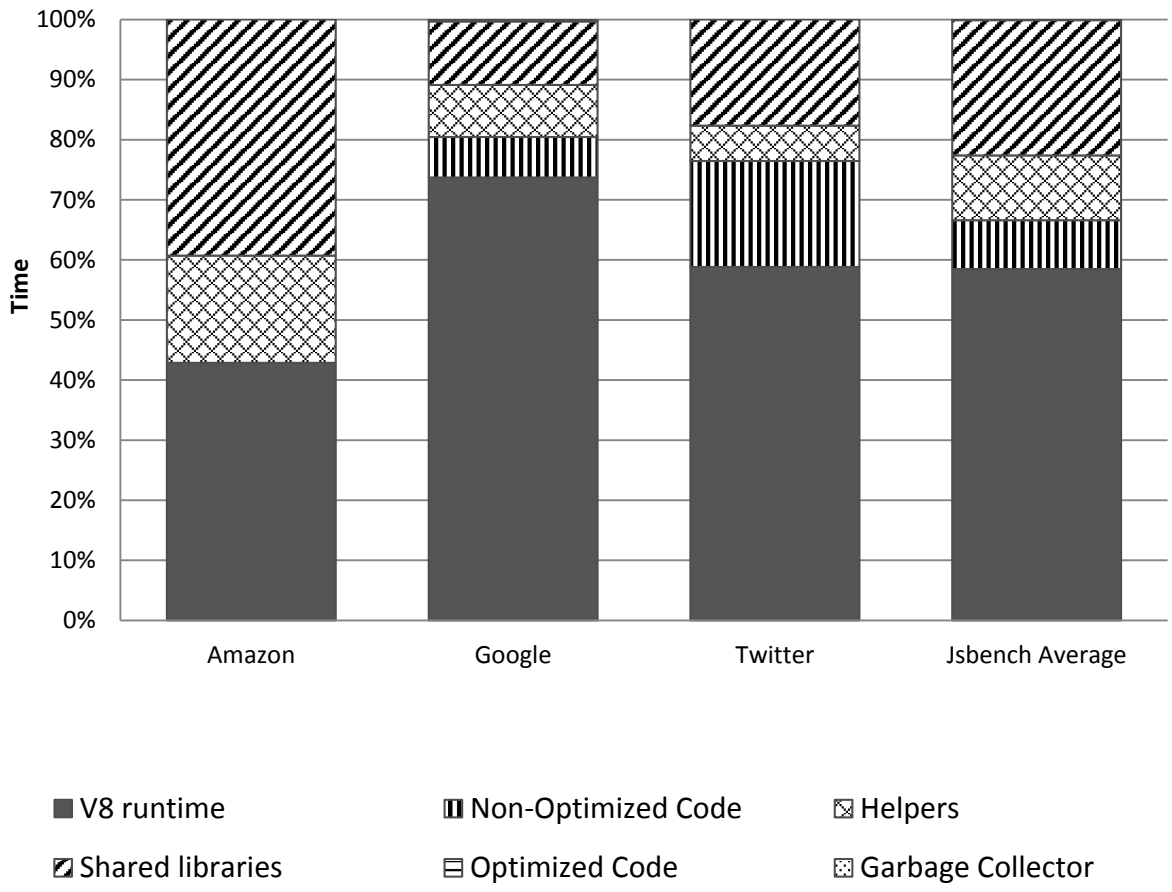

Figure 5.3: V8 execution breakdown in JSBench.

first run, we count the total number of x86-64 dynamic instructions using Pin [16]. In the second run, we introduced additional assembler code (i.e., dynamic counters) to the V8 runtime in order to gather some statistics of interest (e.g., number of times a given check is performed). Results are presented as percentages with respect to the original unmodified code.

The overheads are broken down into three categories: Checks, Tags/Untags and Math Assumptions which are described in section 3.2.4.2. Figure 5.4 shows the breakdown of these overheads. *Rest of code* means the non-optimized code, the garbage collector, V8 runtime tasks, auxiliary libraries, etc. We can observe similar overheads for the three suites. They have a similar percentage of Checks, and these are the most frequent operations. In addition, Tags/Untags category has an important contribution only for a few benchmarks of SunSpider. Finally Math Assumptions only represent 1.4% of the total dynamic instructions. Next, each of the above overheads is further broken down into subcategories.

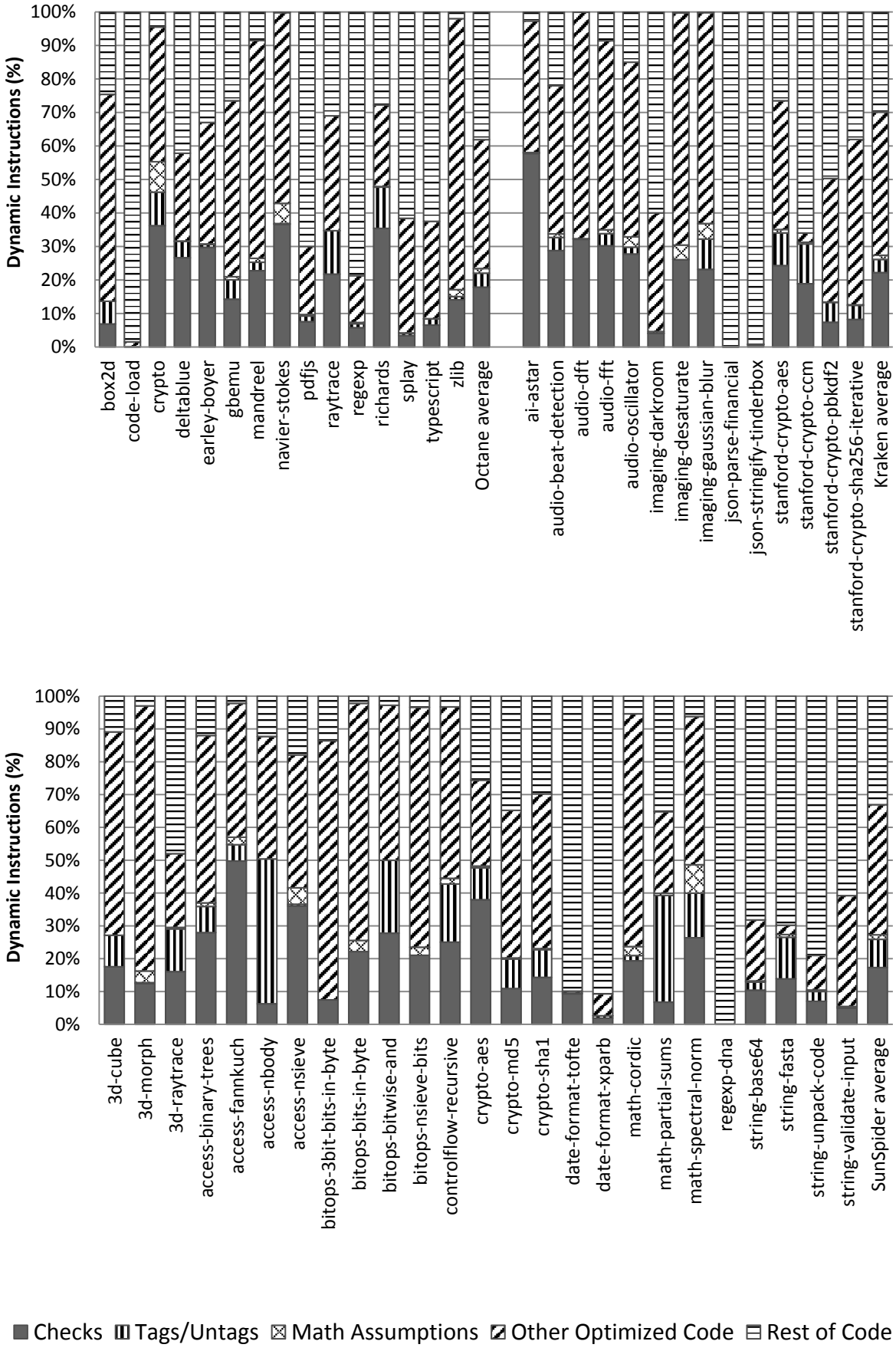

Figure 5.4: Breakdown of main overheads.

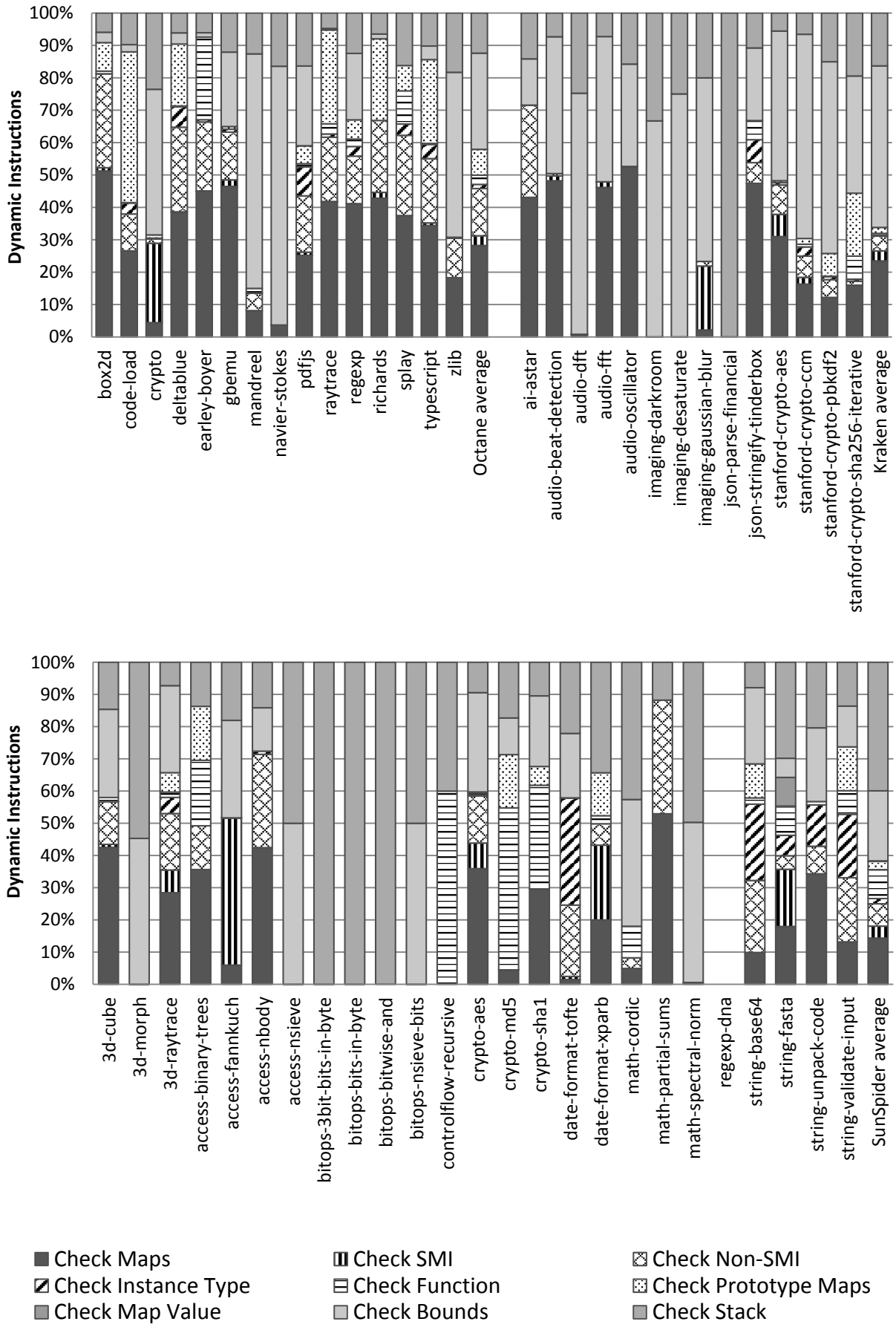

Figure 5.5: Breakdown of checking operations.

5.2.1 Checking Operations

Figure 5.5 shows the dynamic instruction breakdown for Checks. *Check Map*, *Check Stack*, and Check Bounds are the most important checks for SunSpider and Kraken, whereas *Check Maps*, *Check Non-SMI*, *Check Bounds*, *Check Stack* and *Check Prototype Maps* are the most important ones for Octane. Some programs, such as *3d-morph, access-nsieve*, and *math-spectral-norm*, from SunSpider, only have *Check Bounds* and *Check Stack* operations as overhead. It means that these are loop-intensive benchmarks, because V8 generates a *Check Stack* operation for every loop iteration, in order to know if an external exception has happened. Finally, benchmarks with a high percentage of both *Check Maps* and *Check Non-SMI* operations are also very common.

5.2.2 Tagging/Untagging Operations

Figure 5.6 shows a breakdown for Tag and Untag operations. The vast majority of the benchmarks present either *SMI tagging/untagging* operations or *Number tagging/untagging* operations, the former being the most frequent for all suites. Note that some of the *untagging* operations also perform *Check Maps*, *Check Non-SMI* and *Check SMI* operations before the value is untagged, in order to verify that the number to be untagged has the expected type (i.e., either SMI or Non-SMI number). We have included these additional checking operations in the *tagging/untagging* category.

5.2.3 An Example of JavaScript Code

In Figure 5.7, we show an example of a JavaScript function called *findGraphNode*, which is extracted from *ai-astar* benchmark, from Kraken. This benchmark implements the A* graph search algorithm, which finds the best path (i.e., the path with the minimum cost) between two nodes of a graph. The *findGraphNode* function is a member method of a Hidden Class (i.e. *nodeList*) that represents a list of nodes structure. This function checks whether a particular node of the graph is contained in that list. This consists of a loop that compares the *position* property of the node with all the nodes of the list. Note that *this* variable refers to the object itself, which in this case is the list of nodes. In Figure 5.8, we show the main Hidden Classes that involve *nodes* object of Figure 5.7, which belongs *to nodeList* Hidden Class. The node objects contained in *nodes* belong to *GraphNode* Hidden Class and they are stored in the *elements array* of *nodes.*

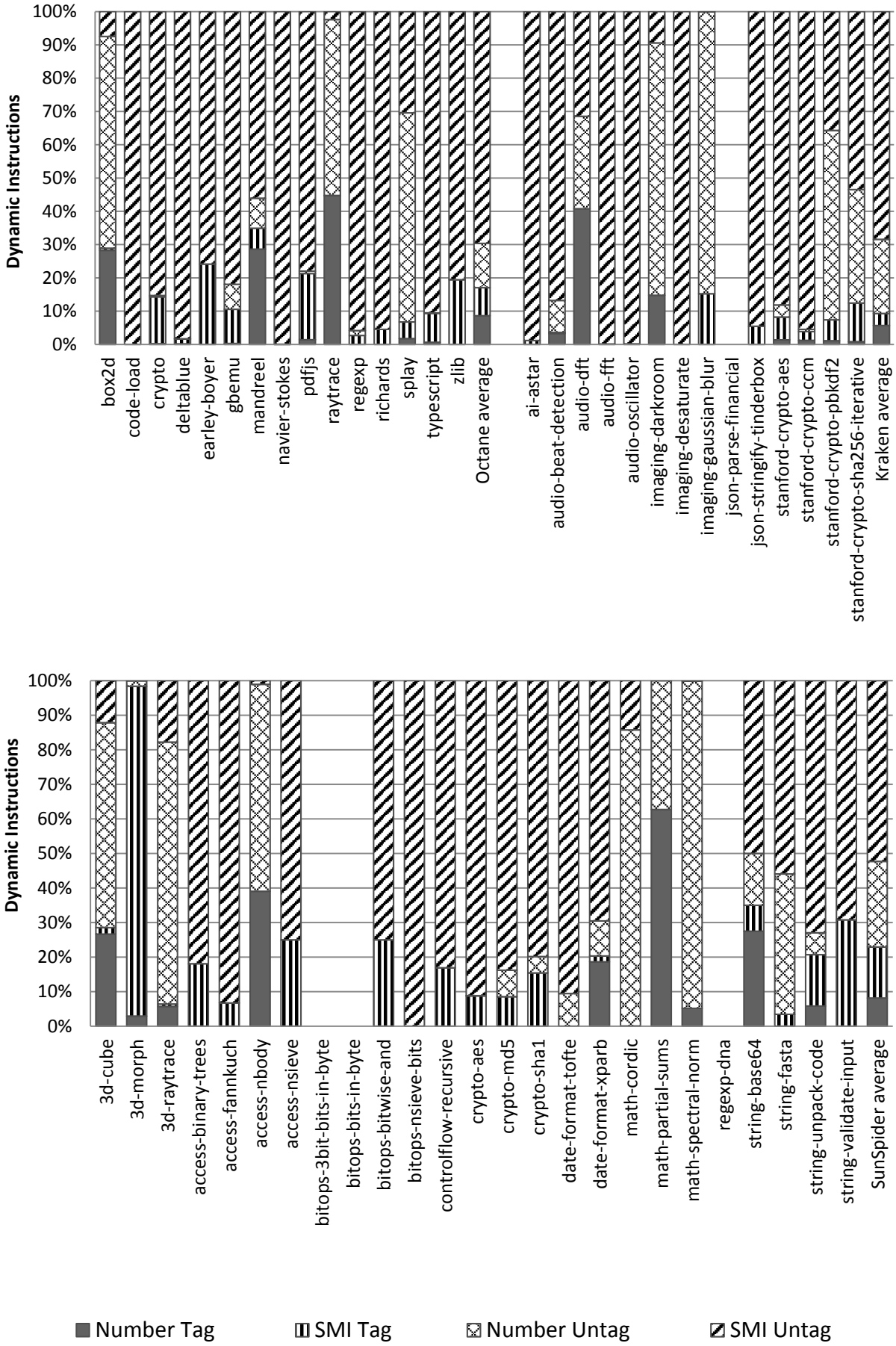

Figure 5.6: Breakdown of Tagging/Untagging operations.

```
1       findGraphNode = function(node) {
2               for(var i=0;i<this.length;i++) {
3                       if(this[i].position == node.position) {
4                               return true;
5                       }
6               }
7                return false;
8       }
```

Figure 5.7: Example of a JavaScript function.

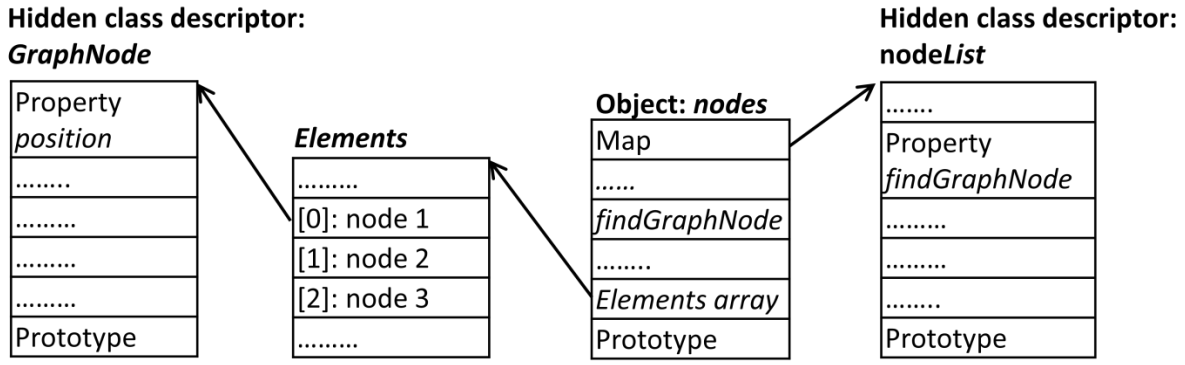

Figure 5.8: *nodes* object structure.

In Figure 5.9, we show the generated x86-64 optimized code corresponding to the *findGraphNode* function of Figure 5.7. Instruction I1 loads the *this* variable, which contains the address of the *nodes* object (i.e., the list of nodes on which the *findGraphNode* function is applied). Then, instructions I2 and I3 perform a *Check Non-SMI* operation, which checks whether its last bit is set to *1*, in order to verify that *this* contains an address. If so, then a *Check Maps* operation takes place, which is executed by instructions I4 to I6. *Check Maps* verifies that the expected Hidden Class (i.e., *nodeList* Hidden Class, which has been profiled during the execution of the non-optimizing code) and the Hidden Class of the *nodes* object is the same. If this comparison is successful, then the *elements array* and the *elements length* properties of *nodes* object are obtained by instructions I7 and I8, respectively. In addition, instruction I9 performs a *SMI Untag* operation over the register (*rbx*) that contains the *elements length* property, which does not require any previous check to verify that the value is a SMI because it is a special property that always contain SMI values. Finally, before entering the loop, the same process is repeated (I10 to I16) with *node* object *(*i.e., the input parameter of the function*)* and the property *position* is obtained. Moreover, Instructions I17-I21 verifies that the Hidden Class of *position* is the expected (*classPosition*).

```
I1    movq rax, this              // load this
I2    test rax, 1                 // check non-smi
I3    jz code_deoptimization      // check non-smi
I4    movq r10, nodeList          // check maps
I5    cmpq (rax-1), r10           // check maps
I6    jnz code_deoptimization     // check maps
I7    movq rdx,(rax+15)           // load elements array
I8    movq rbx, (rax+23)          // load elements length
I9    shrq rbx, 32                // SMI Untag
I10   movq rcx, (rbp+16)          // load node
I11   testb rcx, 1                // check non-smi
I12   jz code_deoptimization      // check non-smi
I13   movq r10, GraphNode         // check maps
I14   cmpq (rcx-1), r10           // check maps
I15   jnz code_deoptimization     // check maps
I16   movq rsi,(rcx+47)           // load node.position
I17   testb rsi, 1                // check non-smi
I18   jz code_deoptimization      // check non-smi
I19   movq r10, classPosition     // check maps
I20   cmpq (rsi-1), r10           // check maps
I21   jnz code_deoptimization     // check maps
loop:
I22   cmpl rdi, rbx               // compare i to this.length
I23   jge return_false           // node not found: return false
I24   cmpq rsp, (stack_limit)     // check stack
I25   jc external_exception       // check stack
I26   cmpl rbx,rdi                // check bounds
I27   jna code_deoptimization     // check bounds
I28   movq r8, (rdx+rdi*8)        // load this[i]
I29   testb r8, 1                 // check non-smi
I30   jz code_deoptimization      // check non-smi
I31   movq r10, GraphNode         // check maps
I32   cmpq (r8-1), r10            // check maps
I33   jnz code_deoptimization     // check maps
I34   movq r9,(r8+47)             // load this[i].position
I35   testb r9, 1                 // check non-smi
I36   jz code_deoptimization      // check non-smi
I37   movq r10, classPosition     // check maps
I38   cmpq (r9-1), r10            // check maps
I39   jnz code_deoptimization     // check maps
I40   cmpq r9,rsi                 // compare property position
I41   jz return_true              // node found: return true
I42   addl rdi,0x1                // i++
I43   jmp loop                    // loop back edge
```

Figure 5.9: x86-64 optimized code corresponding to *findGraphNode* function

Once the loop is entered, instruction I22 is used to compare the loop control variable, *i*, with the total number of loop iterations (i.e., *this.length*). If this comparison is equal, then the function ends (I23) and the *false* value is returned, which means that the node has not been found in the *nodes* list. Otherwise the loop body is executed. At the beginning of the loop body, a *Check Stack* operation is performed by instructions I24 and I25, which compare the stack pointer (contained in *rsp* register) with the *stack limit value*, in order to know whether an

external exception has been produced (i.e., when an external exception occurs, the stack pointer is reset to the *stack limit value*). Then, instructions I26-I27 compare the loop control variable *i* with the *elements length*, in order to check whether the array position is not out of bounds (i.e., *Check Bounds* operation). After this check, the load access to the corresponding *i* position of *elements array* is performed by instruction I28. Again, the obtained value is verified to contain an address by instructions I29 and I30 (*Check Non-SMI*) and the Hidden Class of the object is compared to the expected one by instructions I31 to I33 (*Check Maps*). If these checks are successful, then the property *position* is obtained by instruction I34. Again, *position* is checked to belong to the expected Hidden Class, which is performed by instructions I35 to I39. Lastly, the property *position* of *this[i]* and *node* variables are compared (I40) and if they are equal, the function ends (I41) and the *true* value is returned, which means that the node has been found in the *nodes* list. Otherwise, the control loop variable is incremented by 1 (I42) and the next loop iteration is executed (I43).

5.3 Overheads in the Initial Phase

As stated above, at the beginning of the application, the execution is dominated by the V8 runtime tasks, the non-optimized code and some helpers. Furthermore, during this initial phase the overhead is high. This is due to the time spent in both compilation of the code and initialization and warm-up of the Inline Caches. Note that Inline Caching is managed not only by the non-optimized code, but also by some compilation tasks and helpers. Moreover, as we discussed in section 3.2.2, the most common scenarios for Inline Caching are object property loads and object property stores. In this regard, this section analyzes the overhead produced by this mechanism in these common scenarios during the initial phases of the application. We consider the first iteration of Octane, SunSpider and Kraken applications. For JSBench suite, we consider as initial phase the fourth iteration of the applications, as explained in section 5.1.

Figure 5.10 shows the overhead of Inline Caching for Octane, Kraken and SunSpider suites, due to object property accesses. The same results for JSBench suite are showed in Figure 5.11. Results are broken down into overheads produced during the optimized code and the non-optimized code. It can be seen that this overhead is quite important in practically all programs of JSBench and Octane suites, with an average overhead of 23.4% and 16.7% respectively. However, for SunSpider and Kraken, this overhead is not very important, due to the low amount of non-optimized code and helpers in these initial phases. Moreover, the

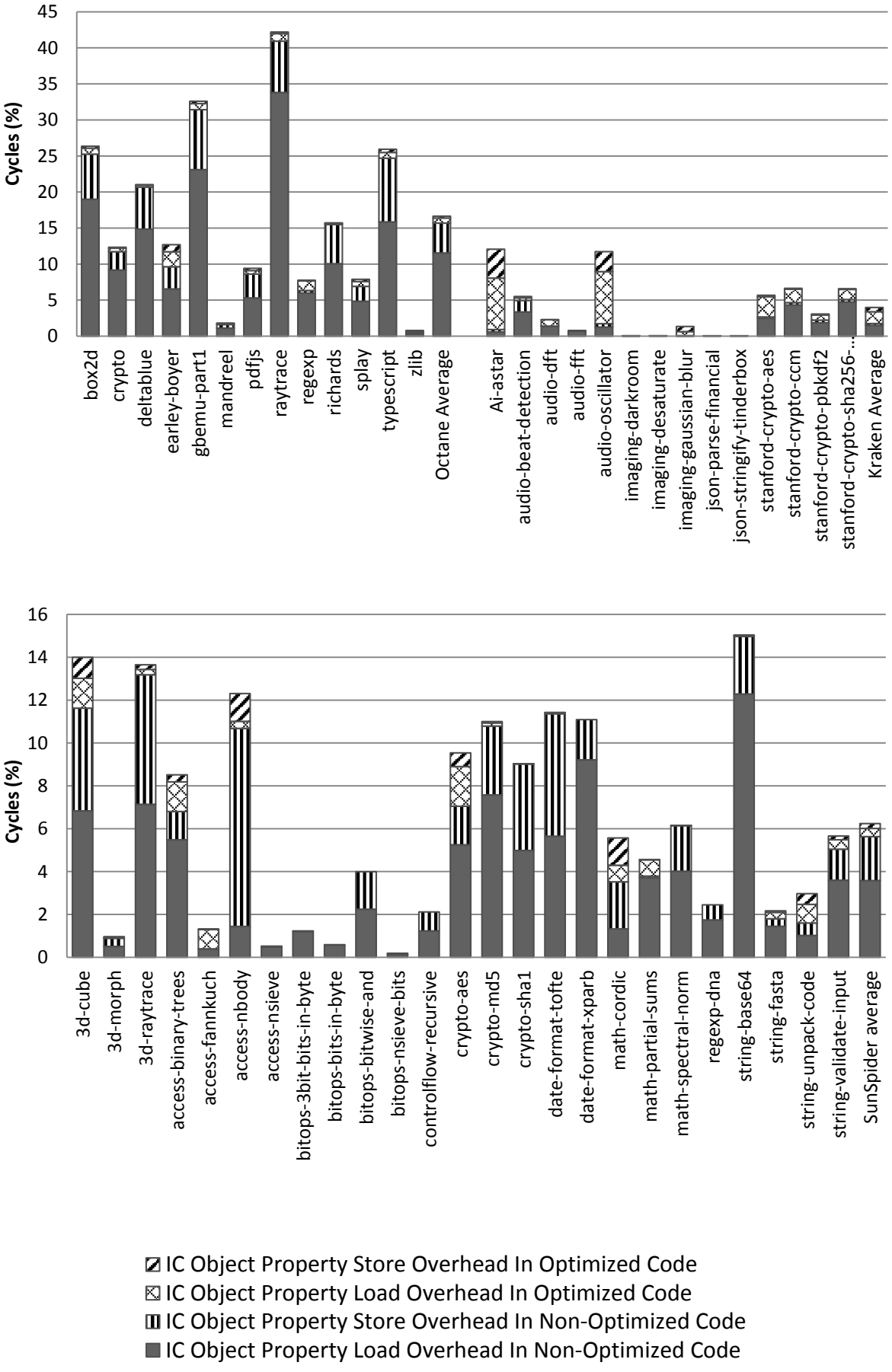

Figure 5.10: Object property accesses overhead.

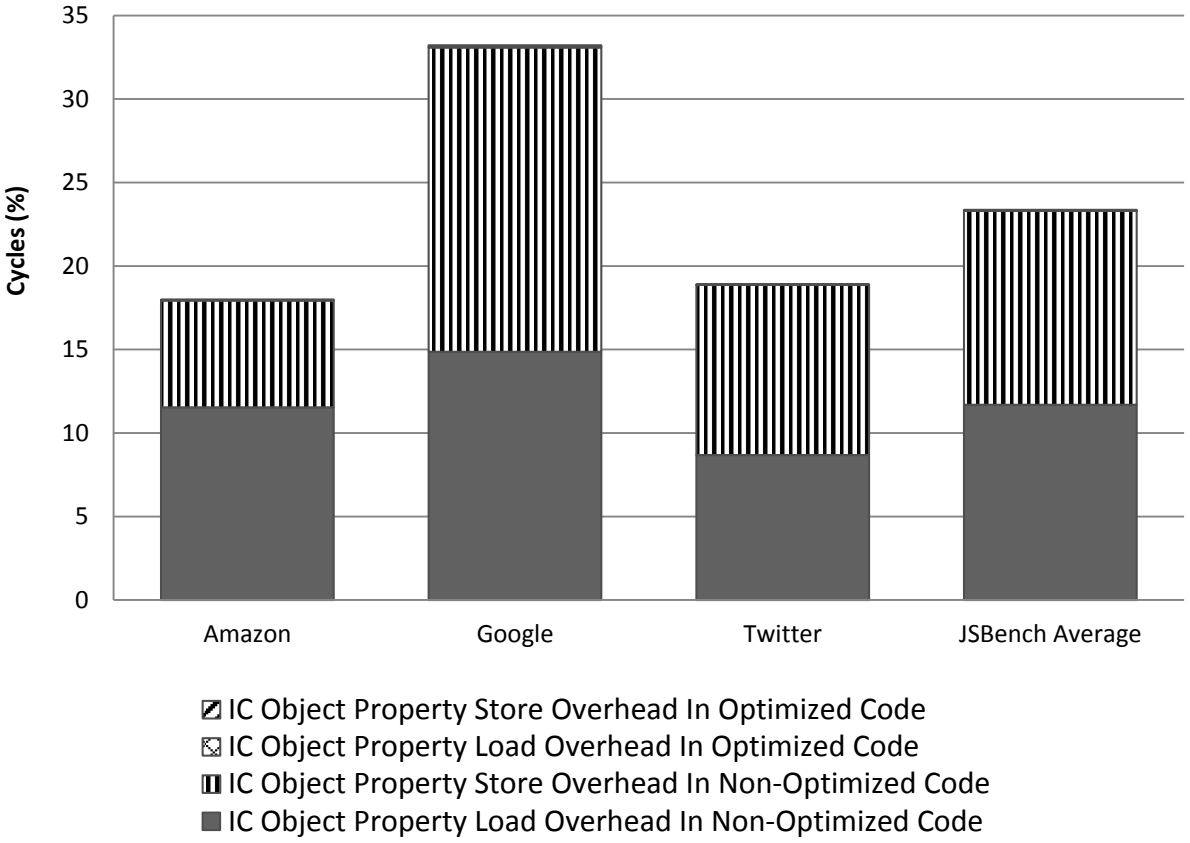

Figure 5.11: Object property accesses overhead for JSBench.

overhead incurred in the optimized code in these latter suites is also low. Note that although the percentage of optimized execution for Kraken is 30%, and the percentage of checking operations in the optimized code is also significant, these checks not only target object property accesses, but also other common scenarios of the Inline Caching mechanism, such as the accesses to object variables that are indexed by a number (i.e., numbered variables). We have not considered these other scenarios because the addition of these variables to an object do not change the object structure (i.e., the Hidden Class of the object keeps being the same) and therefore, the Inline Caches for these scenarios do not reflect the most important issues about dynamic typing. These Inline Caches are used to deal with other more specific optimizations, such as the efficient access to arrays that contain only one particular type of objects (e. g., SMI arrays).

5.3.1 A Simple Example of a JavaScript Application

In this section, we illustrate how the Inline Caching mechanism works during the execution of the non-optimized code. In Figure 5.12 we show an example of a simple JavaScript application, which calculates the total sum of salaries of a university department staff. In lines

```
1   Researcher=function(id,p,n,s){
2       this.id=id;
3       this.position=p;
4       this.numberOfPublications=n;
5       this.salary=s
6   }
7   Technician=function(id,p,s){
8       this.id=id;
9       this.position=p;
10      this.salary=s;
11  }
12  Other=function(id,s){
13      this.id=id;
14      this.salary=s;
15  }
16  departmentStaff=function(name,p){
17      this.departmentName=name;
18      this.people=p;
19      this.calculateTotalSalaries=function(){
20          var total=0;
21          for(var i = 0; i<this.people.length; i++){
22              total=total+this.people[i].salary;
23          }
24          return total;
25      }
26  }
27  var g=new Array(9);
28  g[0]=new Researcher(1,"Phd student",2,900);
29  g[1]=new Technician(2,"Travel administrator",1300);
30  g[2]=new Other(3,1200);
31  g[3]=new Researcher(4,"Postdoc",4,1800);
32  g[4]=new Researcher(5,"Professor",30,2800);
33  g[5]=new Researcher(6,"Phd student",1,900);
34  g[6]=new Technician(7,"Phd administrator",1300);
35  g[7]=new Researcher(8,"Phd student",3,900);
36  g[8]=new Other(9,1000);
37  staff= new departmentStaff("DAC",g)
38  print(staff.calculateTotalSalaries());
```

Figure 5.10: Example of JavaScript code.

1-14, we define the Hidden Classes *Researcher*, *Technician* and *Other*, which represent the different kind of the staff people. For example, people from *PDI* have four properties, which correspond to their id number, position, number of publications and salary. Note that all kinds of staff have the property *salary*. Then, we create another class that represents all the staff of a department (lines 16-26). This class has three properties: the name of the department, the people of the department, and a function called *calculateTotalSalaries*, which is used to calculate the total sum of salaries. The body of this function is based on a loop that traverses a

list that contains all the people from the department and accumulates their salary on the variable *total*, which is returned at the end of the function. Finally, the *main* of this program creates the objects that represent a particular department staff (lines 27-36) and invokes the *calculateTotalSalaries* function for this department.

In Figure 5.13, the generated x86-64 optimized code corresponding to the JavaScript code line 22 of *departmentStaff* function from Figure 5.12 is showed. Concretely, we show the code corresponding to *this.people[i].salary* statement. First, the Inline Cache mechanism is used to obtain the property *people* of the object pointed by *this*. For this purpose, the *this* pointer is obtained from the stack and loaded into the *rdx* register (I1-I2), which corresponds to the first argument of the Inline Cache. Then, the address that contains the "people" string object is loaded (I3) into *rcx*, which is the second argument of the Inline Cache. Finally the incline cache is called (I4), which finds a property called "people" inside the *this* object.

As second step, instructions I5 to I9 are used to obtain the object corresponding to the numbered variable *i* (i.e. the *i*-th element from the *elements array*) of the *this.people* object. In this regard, Instructions I6 and I7 load the control loop variable *i* into the *rcx* register and the instructions I5 and I8 load the address of *this.people in*to the *rdx* register. Finally, the Inline Cache is called again, which finds the object corresponding to the position *i* of the *elements array* contained in *this.people*.

As last step, instructions I10 to I12 are used to obtain the property *salary* of the object obtained by the previous Inline Cache (i.e., *this.people[i]* object). For this purpose, the same process as in the first step is repeated, but using the string "salary" and the *this.people[i]* object as Inline Cache parameters.

```
I1    movq rax, [rbp+16]    // load this
I2    movq rdx, rax         // move this to rdx
I3    movq rcx, "people"    // load "people" to rcx
I4    call IC_loadNamed     // call to property load Inline Cache
I5    push rax              // push this.people on stack
I6    movq rax, [rbp-32]    // load control loop variable i
I7    movq rcx, rax         // move i to rcx
I8    Pop rdx               // pop this.people from the stack to rdx
I9    call IC_loadNumbered  // call to numbered variable load Inline
                               Cache
I10   movq rdx, rax         // move this.people[i] to rdx
I11   movq rcx, "salary"    // load "people" to rcx
I12   call IC_loadNamed     // call to property load Inline Cache
```

Figure 5.11: Generated x86-64 optimized code corresponding to the JavaScript code line 23.

If we focus on the Inline Cache load scenario of *salary* property of Figure 5.13, when the *call* instruction I12 is executed for the first time (i.e., in the first loop iteration), the state of the Inline Cache is not initialized, which means that no Hidden Class has been profiled yet, as shown in Figure 5.14.

In this state, the Inline Cache branches directly (*jump* instruction I5, from Figure 5.14) to a handler routine that searches the relative position that *salary* occupies inside *this.people[0]* object, which depends on its Hidden Class *Researcher*. Then, the value of *salary* is obtained. Finally, the Inline Cache evolves to a monomorphic state, which incorporates the Hidden Class *Researcher* and the relative position (i.e. the offset) that *salary* occupies inside objects belonging to *Researcher,* in order to accelerate future accesses with the same Hidden Class.

Figure 5.15 shows the Inline Cache in monomorphic state. Instruction I3 is used to obtain the current Hidden Class of the object, which is compared to *Researcher* (i.e., the expected Hidden Class) by instruction I5. If this comparison is successful, then the code branches (I6) to specialized code, which obtains the corresponding value of *salary* in a very efficient manner, because the relative position of this property is already known. Otherwise, the code branches to the Inline Cache miss handler (I7).

At the second iteration of the loop of Figure 5.13, the *call* instruction I12 is patched to branch to the new version of the Inline Cache (i.e., the monomorphic state). However, in this loop iteration, the Hidden Class of *this.people[1]* is *Technician,* which is not registered yet by the Inline Cache. Therefore, the code branches to the Inline Cache miss handler, in order to obtain the corresponding position of *salary* for the new Hidden Class. Then, the Inline Cache evolves to polymorphic state, which incorporates the two Hidden Classes seen until now (i.e., *Researcher* and *Technician*), as we can see in Figure 5.16. The polymorphic version is very similar to the monomorphic one, but with the addition of instructions I7 to I9, which cover the case when the Hidden Class of *this.people[i]* is *Technician*.

At the third iteration of the loop, the *call* instruction I12 has been patched to branch to the new polymorphic state of the Inline Cache. However, now the Hidden Class of *this.people[2]* is *Other,* which is not registered yet by this Inline Cache. Therefore, the Inline Cache of figure 5.16 is extended to incorporate this new Hidden Class, as shown in Figure 5.17, where instructions I10 to I12 cover this new scenario. Note that for the remaining loop iterations, *this.people[i]* will always belong to one of the three Hidden Classes already profiled

by this Inline Cache and therefore, it will never miss again to the property handler.

```
I1    pop rbx
I2    push rdx   // Push this.people[i]
I3    push rcx   // Push "people"
I4    push rbx
I5    jmp LoadIC_Miss // Jump to load property IC miss handler
```

Figure 5.14: Inline Cache of the property load scenario in uninitialized state.

```
I1    testb rdx,0x1         // check non-smi
I2    jz miss               // check non-smi
I3    movq rax,[rdx-1]      // load hidden class of this.people[i]
I4    movq rbx, Researcher  // load pointer to Researcher hidden class
I5    cmpq rax,[rbx+7]      // hidden class comparison
I6    jz loadPropertyStub   // branch to load property code (fast)
miss:
I7    jmp LoadIC_Miss       //jump to load property IC miss handler
```

Figure 5.15: Inline Cache of the property load scenario in monomorphic state.

```
I1    testb rdx,0x1         // check non-smi
I2    jz miss               // check non-smi
I3    movq rax,[rdx-1]      // load hidden class of this.people[i]
I4    movq rbx, Researcher  // load pointer to Researcher hidden class
I5    cmpq rax,[rbx+7]      // hidden class comparison
I6    jz loadPropertyStub   // branch to load property code (fast)
I7    movq rbx, Technician  // load pointer to Technician hidden class
I8    cmpq rax,[rbx+7]      // hidden class comparison
I9    jz loadPropertyStub   // branch to load property code (fast)
miss:
I10   jmp LoadIC_Miss       //jump to load property IC miss handler
```

Figure 5.16: Inline Cache of the property load scenario in polymorphic state.

```
I1    testb rdx,0x1         // check non-smi
I2    jz miss               // check non-smi
I3    movq rax,[rdx-1]      // load hidden class of this.people[i]
I4    movq rbx, Researcher  // load pointer to Researcher hidden class
I5    cmpq rax,[rbx+7]      // hidden class comparison
I6    jz loadPropertyStub   // branch to load property code (fast)
I7    movq rbx, Technician  // load pointer to Technician hidden class
I8    cmpq rax,[rbx+7]      // hidden class comparison
I9    jz loadPropertyStub   // branch to load property code (fast)
I10   movq rbx, Other       // load pointer to Other hidden class
I11   cmpq rax,[rbx+7]      // hidden class comparison
I12   jz loadPropertyStub   // branch to load property code (fast)
miss:
I13   jmp LoadIC_Miss       //jump to load property IC miss handler
```

Figure 5.17: Inline Cache of the property load scenario in polymorphic state.

Chapter 6

Fusion of Common Instructions Patterns

In JavaScript long-running, compute-intensive applications, the execution time is dominated by specialized code. Although this code is similar to the code produced by other compilers tailored to statically typed languages, it incorporates important inefficiencies, mainly due to the checking of some assumptions. In this chapter, we propose some techniques to reduce the impact of these overheads.

6.1 Introduction

JavaScript applications are dynamically typed and therefore, object types cannot be inferred at compile time because variables of these applications (including all the object variables) can change their type at any time. In order to deal with this issue, modern JavaScript compilers perform a dynamic profiling of the types of objects. Then, the code is optimized and specialized with the information collected. Moreover, some checking operations and tagging/untagging operations are inserted to this specialized code, in order to check the assumptions about object types. It represents an important overhead, as we saw in chapter 5.

The objective of this chapter is to reduce the overhead of checking and tagging/untagging operations. In this regard, we propose three HW/SW optimizations that reduce the dynamic instruction count and number of cycles for the most common instruction patterns used for checks and tagging/untagging operations. These optimizations require new ISA instructions and some software changes in V8.

There are few works in the literature that reduce these overheads using a HW/SW approach. The most relevant one [47] introduces automatic checking of types of objects, in order to reduce the overhead of these checks. However, that work only deals with checks that target property accesses. The technique presented in this chapter, addresses all types of checking operations, which have been analyzed in Chapter 5. Furthermore, it also improves

the performance of some other patterns of instructions, such as the *SMI Untag* operations.

In the rest of the chapter we first motivate our proposal. Next, the proposed technique is presented and lastly, the results are shown and discussed.

6.2 Motivation

In Figure 6.1, we show the overheads related to checks and tagging/untagging operations, which were quantified in section 5.2. These overheads represent 25.4% of the total dynamic instructions for long-running, compute intensive applications. If we take into account only the optimized code, these overheads represent 37.4% of the total dynamic instructions, which is a very important fraction of the total activity.

We have observed that most of these operations follow the same pattern of machine instructions. The most common patterns consist of the execution of two or three instructions that verify that a particular assumption is correct and a branch to deoptimization code when the assumption is not fulfilled. The most frequent checking operation is *Check Maps*, which usually executes the pattern of three x86-64 instructions showed in Figure 6.2a. The first instruction is a *move*, which is used to load the expected Hidden Class to register *regA*. The second instruction is a *cmp*, in order to compare the expected Hidden Class with the Hidden Class of the current object, which is contained in *regB.* Finally, the third instruction branches to a deoptimization code, in case that the comparison is not equal. Another example of these patterns is showed in Figure 6.2b, which corresponds to the *Check Non-SMI* operation. In this example, two x86-64 instructions are executed. The first one is a *test* instruction, which is used to check whether the last bit of the register *regA* is set to *1*, which means that the register contains the address of an object, instead of a SMI. The second instruction is a branch instruction, which branches to code deoptimization, in case that the register contains a SMI. Figure 6.2c shows the two-instruction sequence for a *Check Stack* operation. The first instruction compares the *stack pointer* with the *stack limit*, which is contained in the memory position pointed by *(regA + imm).* If the *stack pointer* is below the stack limit, then the code branches to an exception routine, instead to deoptimization of the code.

The instructions used to verify math assumptions follow patterns similar to checking operations. However, these patterns do not perform any comparison or testing before branching to deoptimization code. Instead, the branch to deoptimization is based on the

outcome of an arithmetic or a logical instruction. In Figure 6.2d, we show an example of a math assumption scenario, which consists on a 32-bit addition (I1) between a SMI value contained in a register (*regA*) and an immediate value (*imm*). For the rest of the execution, V8 assumes that the resulting value is also a SMI value, which needs less costly tagging/untagging operations, compared to non-SMI numbers. However, this fact has to be validated by Instruction I2, which branches to code deoptimization, in case that the result overflows (i.e., the result needs more than 32 bits for its representation, which means that it is not a SMI value).

Regarding tagging/untagging operations, the most common instruction pattern corresponds to the *SMI Untag* operation, which is shown in Figure 6.2e. In this case, the first two instructions check whether the register *regA* contains a SMI (i.e., *Check SMI* operation). If so, a 32-bit shift right operation is performed, in order to untag the SMI value contained in *regA*. Otherwise, the code is deoptimized.

Finally, we have detected other frequently executed instructions patterns, which are composed of various checking operations. The most common one is a *Check Non-SMI* followed by a *Check Maps*, as shown in Figure 6.2f.

In conclusion, there is a great opportunity to improve the performance of JavaScript platforms by reducing the latency and number of instructions used to perform the overhead operations described above.

6.3 Optimization of Common Instructions Patterns

Below we present three proposed HW/SW optimizations that target the patterns described in the previous section. In this regard, we extend the ISA with new x86-64 instructions and we describe the required hardware to implement these new instructions.

6.3.1 HW Exception Mechanism

We have observed that when the code may branch to deoptimization code (due to math assumptions and checking operations, with the exception of *Check Stack*, as described in Section 3.2.4.2), in the vast majority of cases (almost 100%) the code is not deoptimized. Every time the optimized code is checked to potentially deoptimize it, two instructions are used. The first one is an instruction that changes a flag. This instruction is usually a *test* or a *cmp* instruction, but can also be an arithmetic or a logical instruction, such as the *add* instruction of

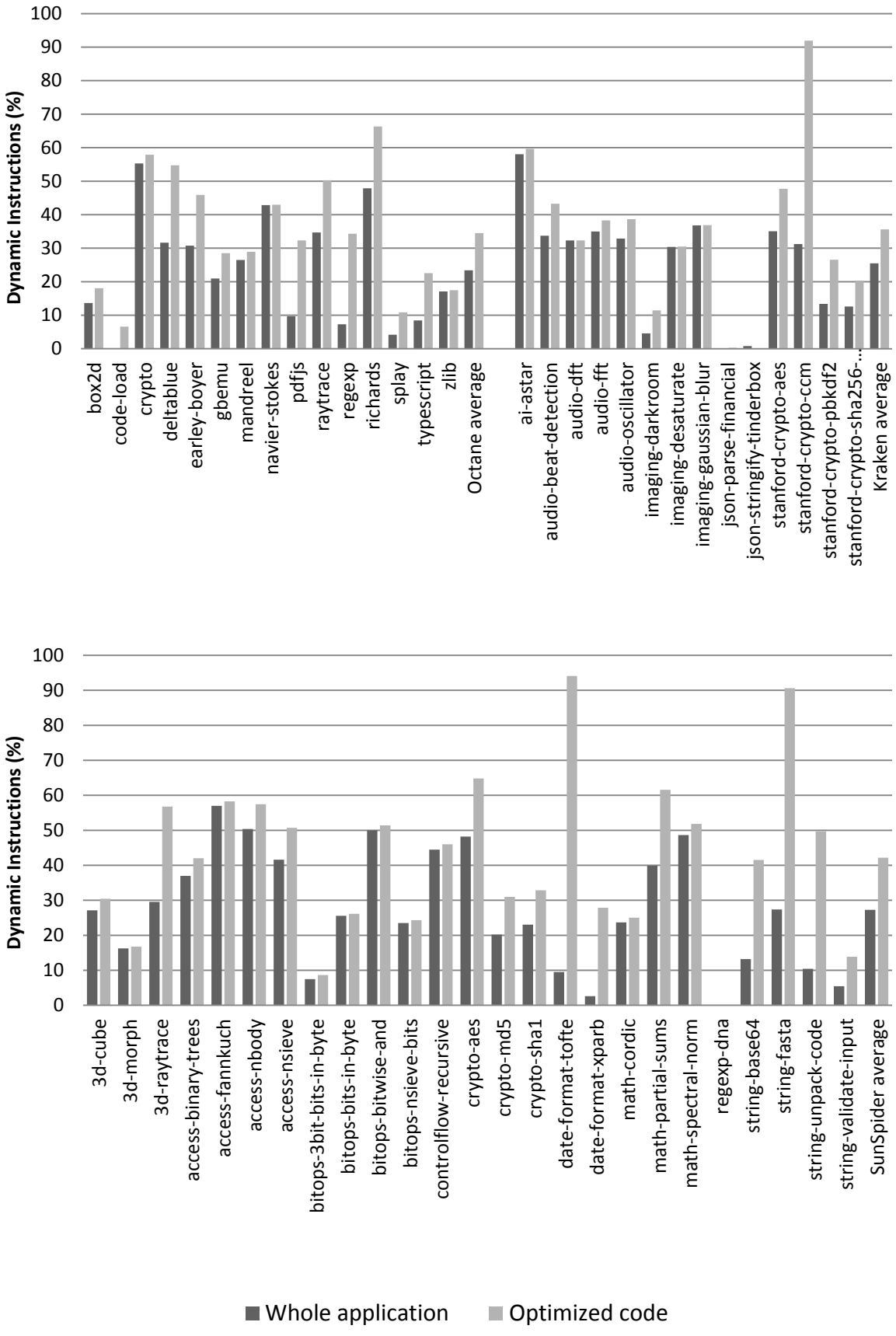

Figure 6.1: Overhead produced by Checking operations, tagging/untagging operations and Math Assumptions.

```
I1   mov regA, expected_type    // load expected type
I2   cmp (regB-1), regA         // compare expected type with
                                   the type of the object
I3   jnz deoptimization bailout // if not equal, branch to
                                   deoptimization
```

a) *Check Maps.*

```
I1   test regA, 1               // test last bit of the object
I2   jz deoptimization bailout  // if it is not set to 1, branch to
                                   deoptimization
```

b) *Check Non-SMI.*

```
I1   cmp rsp, (regA+imm)        // stack pointer with stack limit
I2   jc external exception      // if stack pointer is below stack
                                   limit, branch to external
                                   exception
```

c) *Check Stack.*

```
I1   add regA,imm32             // add immediate to regA
I2   jo deoptimization bailout  // if the result overflows, branch
                                   to deoptimization
```

d) *Integer Addition.*

```
I1   test regA, 1               // test the last bit
I2   jnz deoptimization bailout // if it is not set to 1, branch to
                                   deoptimization
I3   shr regA, 32               // perform shift right 32-bits
                                   displacement
```

e) *SMI Untag.*

```
I1   test  regA, 1              // test last bit of the object
I2   jz deoptimization bailout  // if it is not set to 1, deoptimize
I3   mov regB, expected_type    // load expected type
I4   cmp (regA-1), regB         // compare expected type with object
                                   type
I5   jnz deoptimization bailout // if not equal, branch to
                                   deoptimization
```

f) *Check Non-SMI and Check Maps*

Figure 6.2: Instructions patterns for checking operations, tagging/untagging operations and math assumptions.

Figure 6.2e. The second one is a conditional branch to a deoptimization bailout depending on the flag.

Our proposal is to replace these two instructions by a new one. This new instruction performs a scalar operation (*test*, *cmp*, *add*, etc.) and checks the resulting value of a specific flag. Moreover, this instruction can throw a HW exception according to the encoded specific flag. Therefore, branch prediction is not needed and whenever the code is not deoptimized, less dynamic instructions are executed. If the code has to be deoptimized, a hardware exception is thrown. This exception is intercepted by a handler in the V8 runtime and executes a special routine, which finds the action to do according to the current program counter. This action is a jump to an address that targets a specific deoptimization bailout. The overhead of this lookup is negligible compared with the deoptimization routine itself.

The new x86-64 instructions that we have introduced to allow the implementation of this optimization are described in Appendix A. Note that the third argument of these new instructions corresponds to the flag to be checked and the fourth argument indicates the expected value for that flag (*0* or *1*), in order not to throw a HW exception.

In Figures 6.3, 6.4 and 6.5, we show three examples of these optimizations, which correspond to the instructions patterns showed in Figures 6.2a, 6.2b and 6.3d, respectively. We show the changes at x86-64 and microinstruction level. Note that each microinstruction is numbered according to the x86-64 instruction that it belongs to.

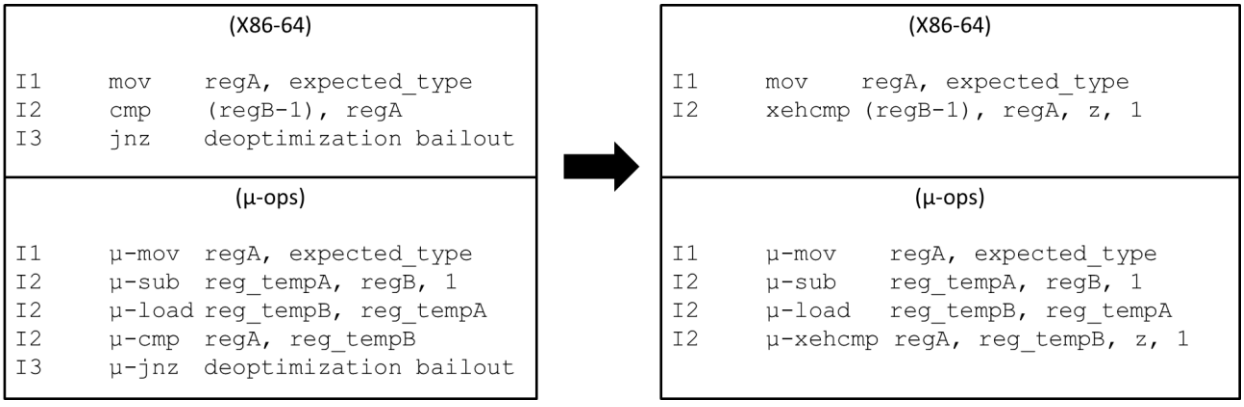

Figure 6.3: HW Exception mechanism improvement for *Check Maps*.

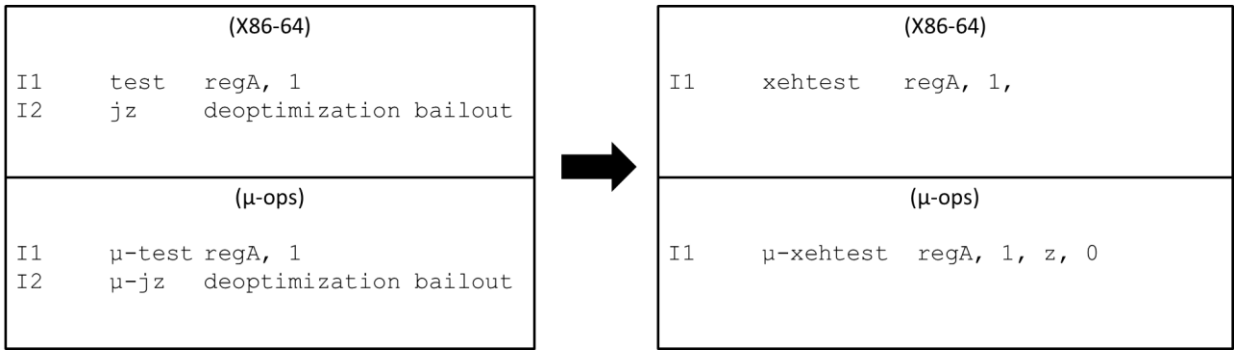

Figure 6.4: HW Exception mechanism improvement for *Check Non-SMI*.

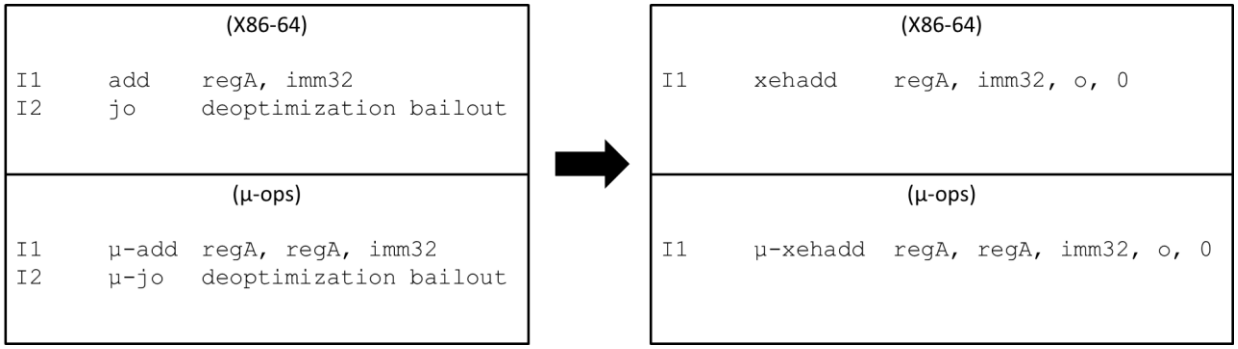

Figure 6.5: HW Exception mechanism improvement for *Integer Addition*.

We can use the same exception mechanism to optimize the *Check Stack* pattern. This check uses the same routine as the deoptimization bailout to find the action to do. However, in this case, the action is a call that interrupts the program because an external exception has taken place, as explained in section 3.2.4.2. Figure 6.6 illustrates this optimization at instruction and microinstruction level.

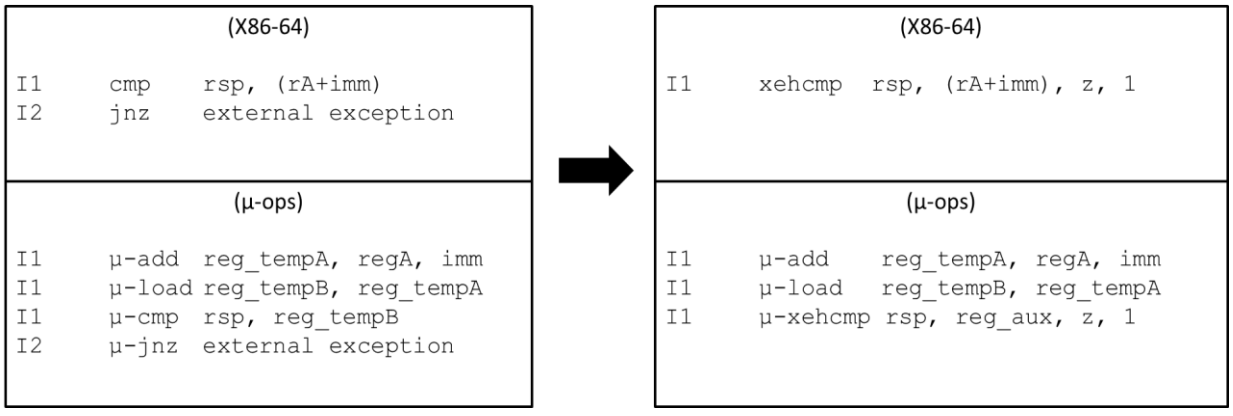

Figure 6.6: HW Exception mechanism improvement for *Check Stack*.

Figure 6.7 outlines the required hardware support for this mechanism. A mux is added in the execute stage to select the flag to check, which is compared with the expected value. The result of this comparison is saved in the exception bit. If this bit is set, a HW exception

will be thrown at the commit stage. Note that the Hardware support for this optimization is very simple and, in addition, the critical path is not affected, because the result is not needed until the commit stage.

These optimizations can also be applied to other common JavaScript engines and other dynamically typed languages. For instance, SpiderMonkey [45] and Nitro [63] introduce a guard just before unboxing an object, in order to ensure that the specific type of the object is the expected one. This guard also consists of a comparison and a subsequent conditional jump to a bailout that can be optimized in the same way as shown for V8. These guards are also used for other type of assumptions (e.g., arithmetic assumptions).

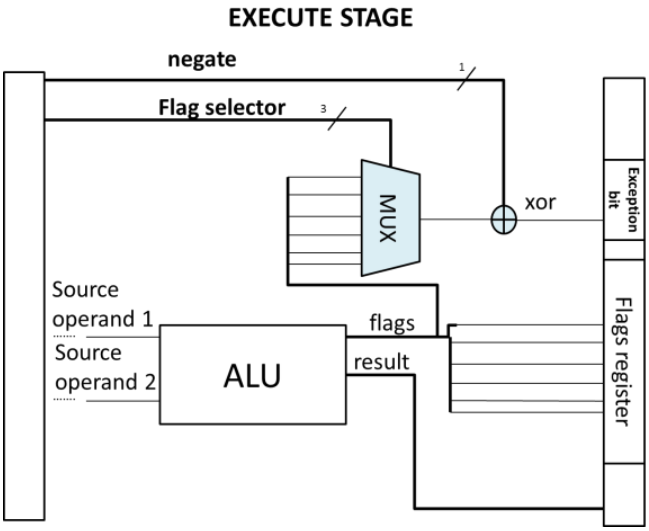

Figure 6.7: Block diagram for the *HW Exception mechanism*.

6.3.2 SMI Untag Pattern

As commented in section 6.2, *SMI Untag* operations follow a very common pattern of instructions for tagging/untagging. These operations are necessary when SMI integer values need to be unboxed. As shown in section 5.2, they represent 5% of the dynamic instructions for SunSpider on average, and can reach up to 9% for some benchmarks.

The optimization that we propose for the *SMI Untag* instruction pattern of Figure 6.2e is based on replacing three x86-64 instructions by a new single one, which is called *xehtestshr*. This new instruction shifts the value of the register at the same time that checks whether the value is a SMI. If it is not, an exception is raised. Figure 6.8 presents the code and micro code for the *SMI Untag* pattern before and after applying this optimization.

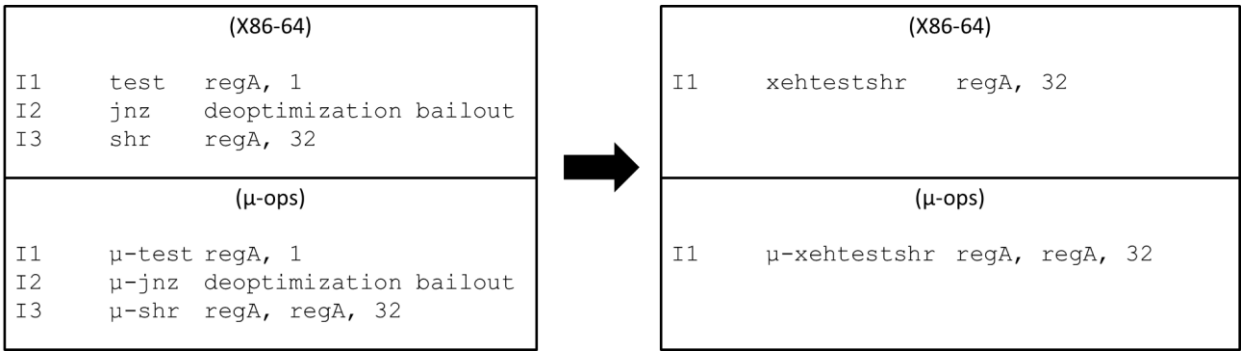

Figure 6.8: *SMI Untag* pattern improvement.

Figure 6.9 presents a block diagram of the hardware required to support the *xehtestshift* instruction, which is an extension of the scheme of Figure 6.7. Note that another mux is introduced to select between the flag indicated by the previous optimization and the least-significant bit, which indicates whether the value is a SMI. In this case, the least-significant bit is compared to *0* (i.e. because the last bit of a SMI is cleared) and the result of this comparison is saved in the exception bit. If this bit is set, then a HW exception will be thrown at the commit stage. Note that both the *negate* bit, the *flag selector* and the *optimization mode selector* bit of Figure 6.9 are directly codified in the *xehtestshift* instruction, instead of as a source operands.

Both Nitro and IonMonkey encode the integer tag into the most significant part of the 64 bit register. This optimization could be easily adapted for them.

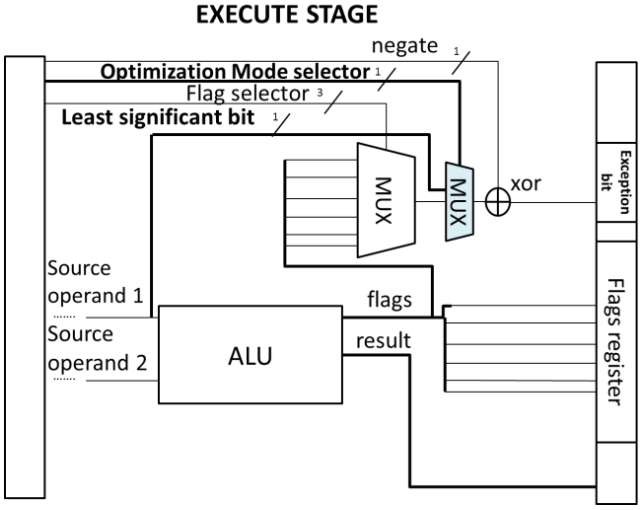

Figure 6.9: Block diagram for the *SMI Untag* pattern optimization

6.3.3 Check Non-SMI and Check Maps Pattern

As explained in section 6.2, there are sequences of different checking operations that are very frequent. The most repeated of these patterns is a *Check Non-SMI* followed by a *Check Map* operation, so in this section we present an optimization for it.

The optimization that we propose consists of replacing the instructions I1, I2, I4 and I5 by a new single instruction called *xehtestcmp*, as we can see in the upper part of Figure 6.10. The optimized instruction sequence consists of the *mov* instruction I3, followed by the *xehtestcmp* instruction. This new instruction has two source operands: the first one is a register that contains an object address. The second is a register that contains the expected *Hidden Class identifier* (type) of the object pointed by the first operand.

The bottom part of Figure 6.10 shows this optimization at microinstruction level. We can see that the x86-64 *xehtestcmp* instruction is cracked into three microinstructions. The first one (*μ-xehtestsub*) checks whether the address of the object is not a SMI, at the same time that calculates the effective address of its first memory position, which contains the *Hidden Class identifier*. If the address is a SMI, then a HW exception is raised. Note that the hardware mechanism for *μ-xehtestsub* is very similar to the mechanism described above for *μ-xehtestshr.* However, *μ-xehtestsub* microinstruction decrements the value contained in the source register *regA*, instead of performing a shift right operation.

The second microinstruction (*μ-load*) performs a memory load operation using the effective address stored in *reg_tempA*. Finally, the third microinstruction (*μ-xehcmp*) is the same used for the mechanism described in section 6.3.1. This instruction raises and exception in case that the values stored in *reg_tempB* and *regB* are not equal. In this way, the number of dynamic x86-64 instructions for *Check Non-SMI and Check Map* pattern is reduced from five to two and the number of microinstructions is reduced from seven to four, as showed in Figure 6.10.

As Both Nitro and IonMonkey encode the integer tag in the same way as the other object types, they do not need to use this optimization. They simply need to perform a single check for the expected type of the object.

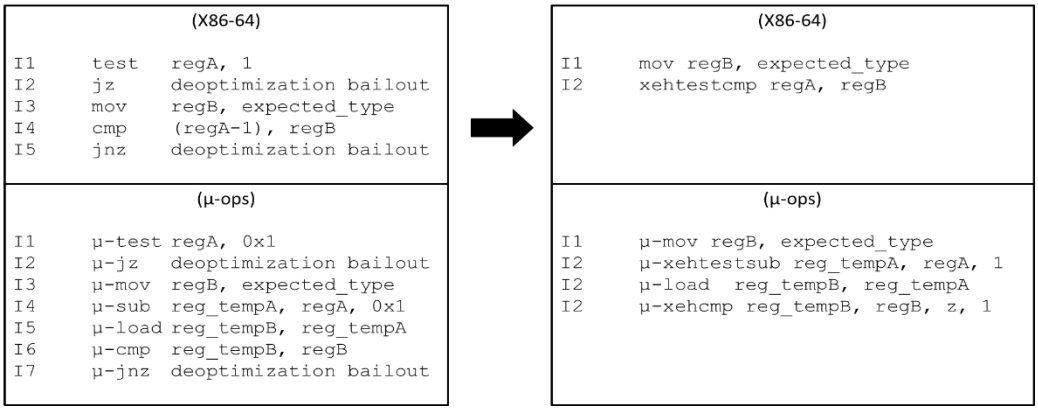

Figure 6.10: *Check Non-SMI* and *Check Maps* pattern improvement.

6.4 An Example of the Proposed Optimizations

As example, Figure 6.11 shows the new x86-64 code for the example described in section 5.2.3, before and after applying our optimizations. The bold Instructions of the left part of this Figure correspond to the instructions patterns optimized in this example. The right part of this Figure shows the resulting code after applying our optimizations, which are also in bold.

Note that Instructions I2-I6, I11-I15, I17-I21, I29-I33 and I35-I39 correspond to the *Check Non-SMI and Check Map* pattern and instructions I24-I25 and I26-I27 correspond to a *Check Stack* and *Check bounds* operations, respectively. The code of the outer loop is reduced from 21 to 12 x86-64 dynamic instructions, whereas the code of the inner loop is reduced from 22 to 14 x86-64 dynamic instructions.

6.5 Performance Evaluation

Below we present the performance of the proposed optimizations using Octane, Kraken and SunSpider benchmark suites. As in section 5.2, the results are reported for the tenth iteration, in order to focus on the steady state of the applications.

6.5.1 Dynamic Instruction Count Improvements

We have measured the reduction in instruction count through instrumentalization of the V8 engine. In this regard, we have inserted additional assembler code to the V8 runtime, in order to identify the instructions patterns that our mechanism optimizes. Then, we have quantified the number of removed x86-64 dynamic instructions in these patterns. On the other hand, the number of total dynamic instructions has been obtained using Pin [16].

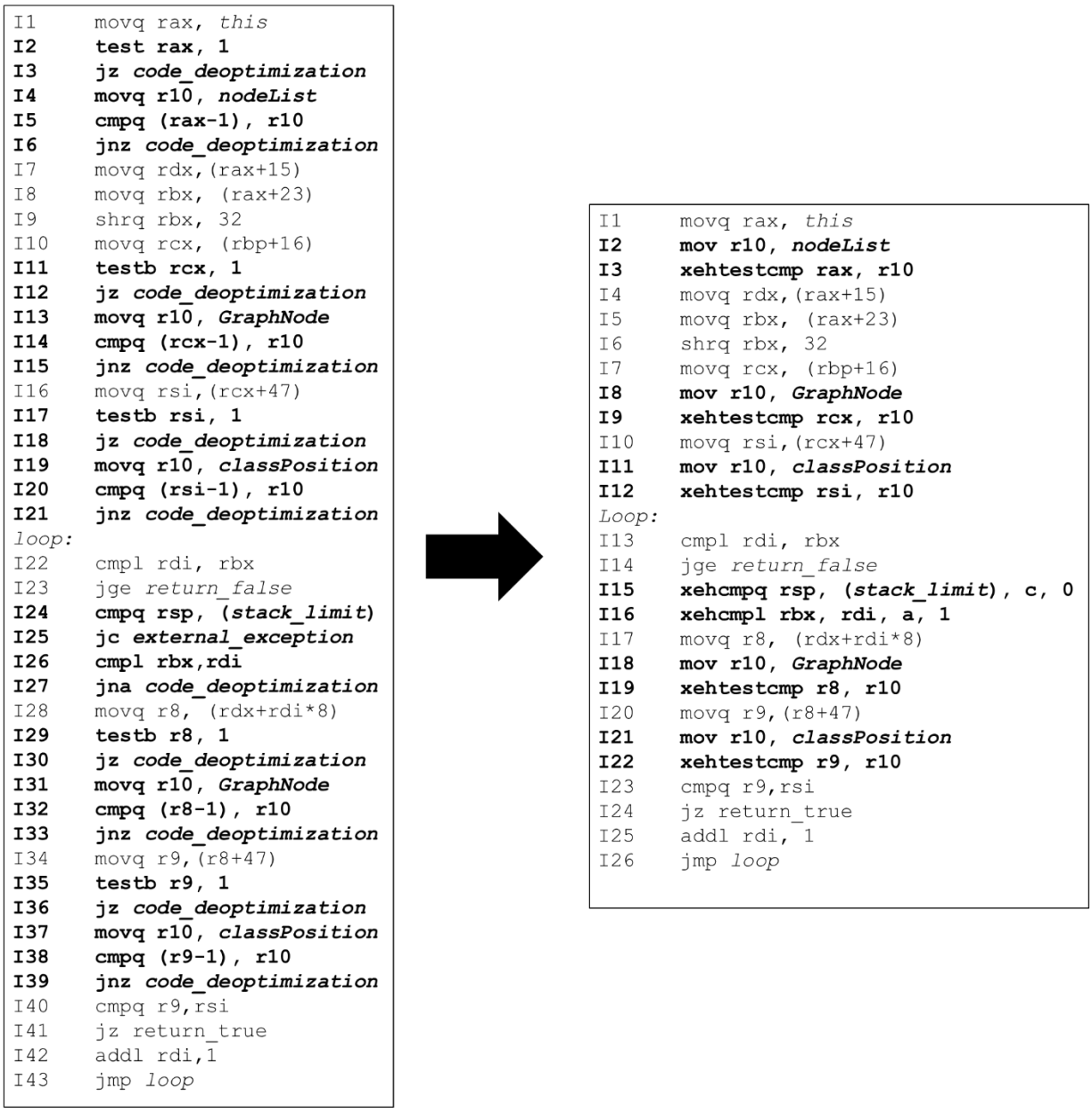

Figure 6.11: Example of the proposed optimizations.

Figure 6.12 shows the results for the three benchmark suites. Overall the proposed techniques achieve an 11.2% dynamic instruction reduction for the whole application and 17.3% reduction for optimized code. All three suites get an important improvement from the HW Exception mechanism optimization because it targets all kinds of check operations. On the other hand, most benchmarks from Octane execute an important amount of *Check Non-SMI and Check Map* patterns so they significantly benefit from our optimization. Finally, *controlflow-recursive* and *bitops-bitwise-and* benchmarks, from SunSpider, and *Richards* and *Crypto*, from Octane, benefit a lot from our *SMI Untag* pattern optimization, which reduces to two thirds the total dynamic instructions used for these operations.

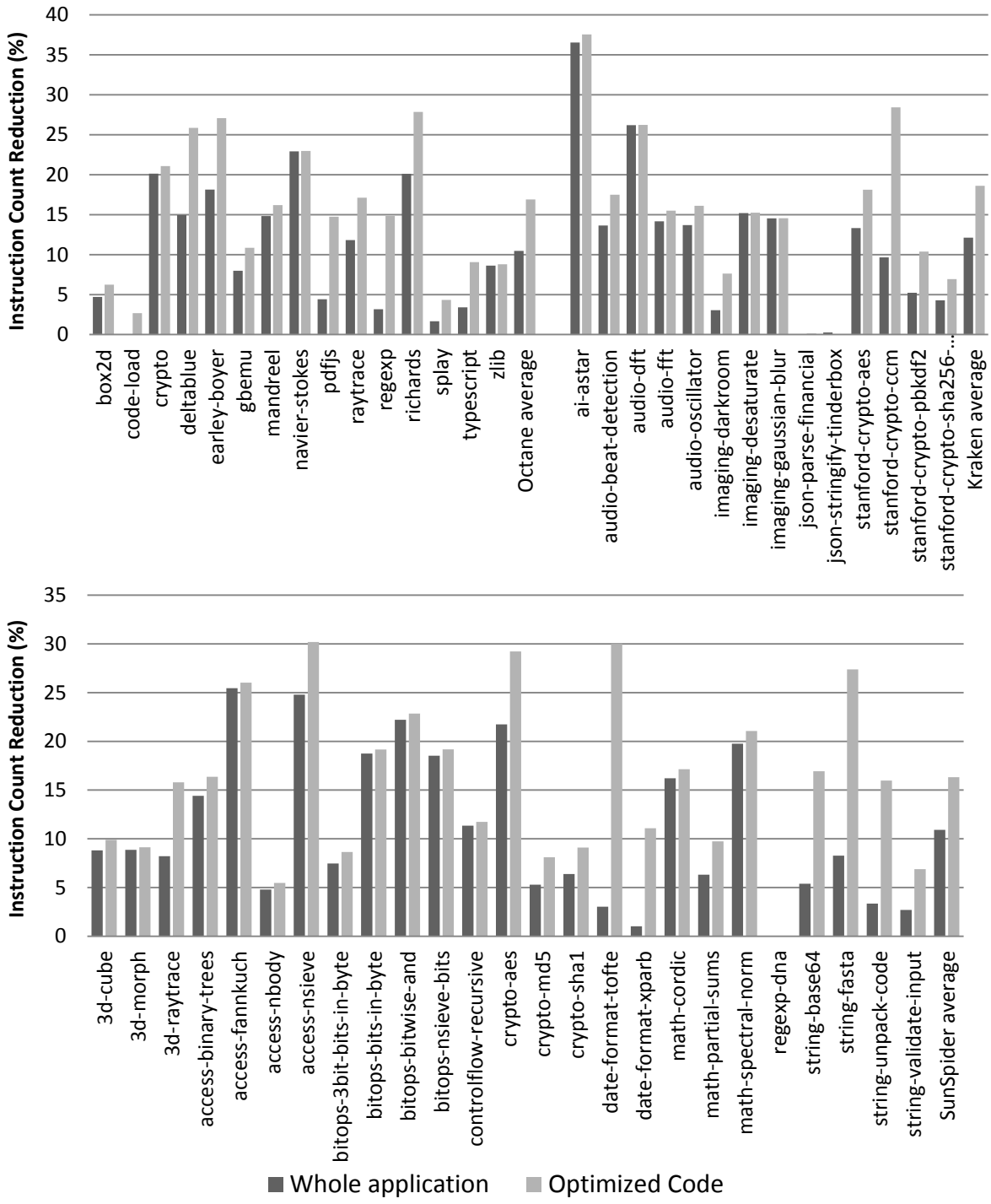

Figure 6.12: Improvement in dynamic instructions.

6.5.2 Cycle Count Improvements

In this section we analyze the execution time improvements of our technique using the Sniper simulator [58]. For this purpose, each benchmark was previously executed nine times for the warm up phase (*mandreel* and *typescript* are not included since they crash in our simulation environment) and statistics were taken from the tenth execution. We have extended Sniper with the ability to detect the patterns mentioned above. Then, for each of these patterns, we have

obtained the number of cycles that the removed instructions (and corresponding microinstructions) take for their execution, in order to subtract this number from the total cycles of the entire simulation.

Figure 6.13 presents the speedups for the three benchmark suites. Overall the proposed techniques achieve a 6.2% cycle count reduction for the whole application and 9% for optimized code. Regarding the whole application, the speedups achieved by the three suites are approximately half the dynamic instruction count reduction results showed in the previous section. This is basically due to two main reasons. On the one hand, the instructions that we are removing belong to checking operations, which usually are not in the critical path of the applications. They consist basically of either a compare instruction or an arithmetic instruction that is followed by a branch that rarely is taken and thus, the branch predictor for these cases is very accurate. Therefore, as long as there are enough resources, these instructions do not suppose a main bottleneck for the application.

Besides, the kind of instructions that we are removing in our optimizations are cracked into only one microinstruction whereas the new x86-64 instructions need more than one microinstruction for their execution.

6.5.3 Energy Consumption

Figure 6.14 shows the energy savings of our technique for the three benchmark suites, which are measured through the McPAT simulator [53]. Energy consumption is reduced by 3.9% on average for the whole application and 5.7% for optimized code. These savings correlate with the reduction of execution time (which results in less leakage energy) and number of executed instructions (which results in less dynamic energy).

6.6 Conclusions

The analysis performed in section 5.2 showed that around 25% of the overhead produced in a steady state execution of representative benchmarks is due to checking, tagging/untagging and math assumptions operations. In addition, we have found that most of these operations follow the same pattern of instructions. In this regard, three optimizations are proposed in this chapter, in order to reduce the dynamic instruction count and number of cycles due to these patterns, which represent an important fraction of the quantified overhead. These optimizations are

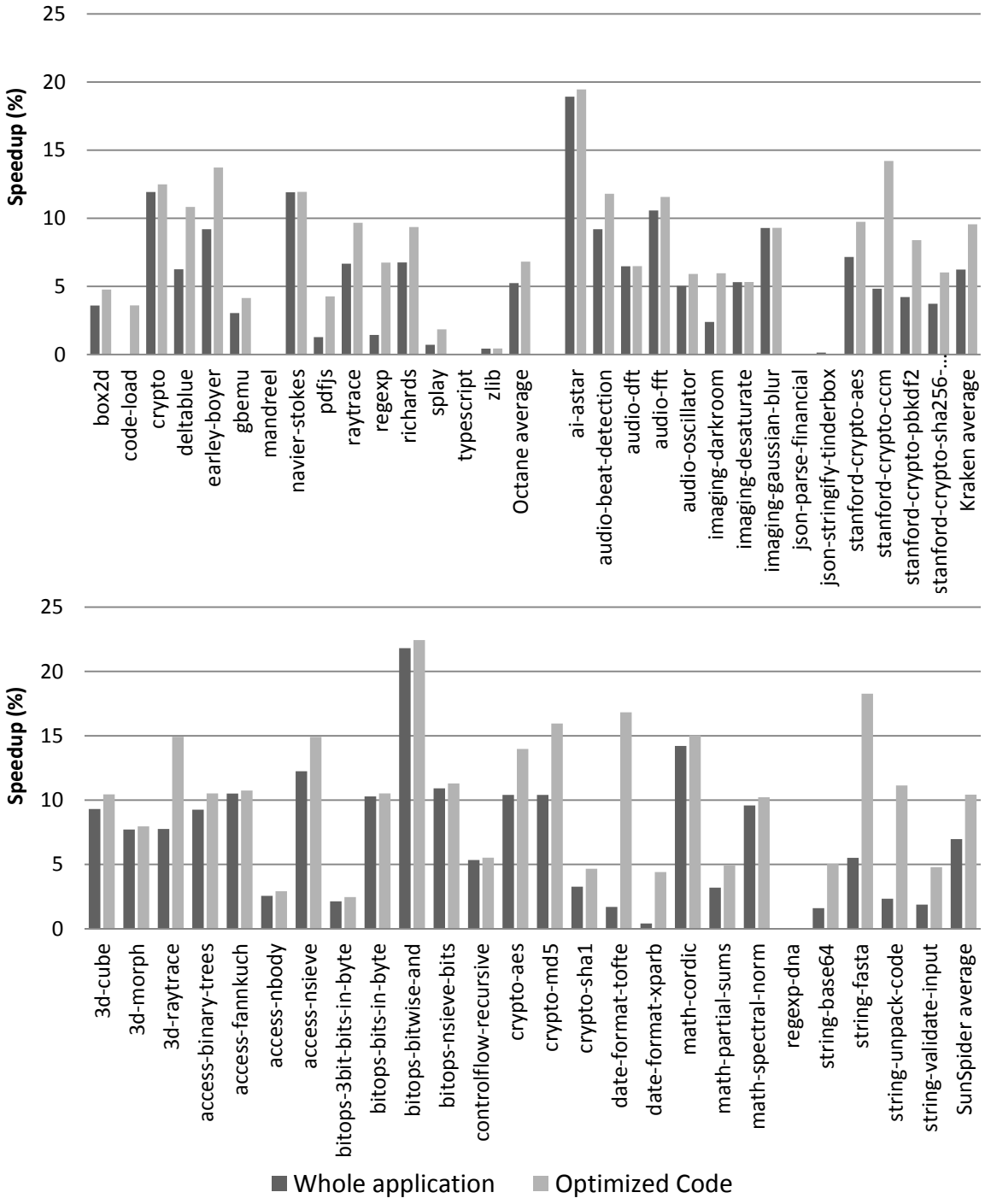

Figure 6.13: Improvement in number of cycles.

based on a hybrid HW/SW approach that requires the introduction of some new machine instructions, some additional hardware support and some changes in the code generated by the dynamic compiler.

We have shown that these optimizations result in an average 6.2% speedup and 3.9% reduction in energy consumption for representative benchmarks. Although the techniques are implemented on V8 JavaScript engine, these optimizations can be extended to other engines for dynamically typed languages using similar type profiling mechanisms.

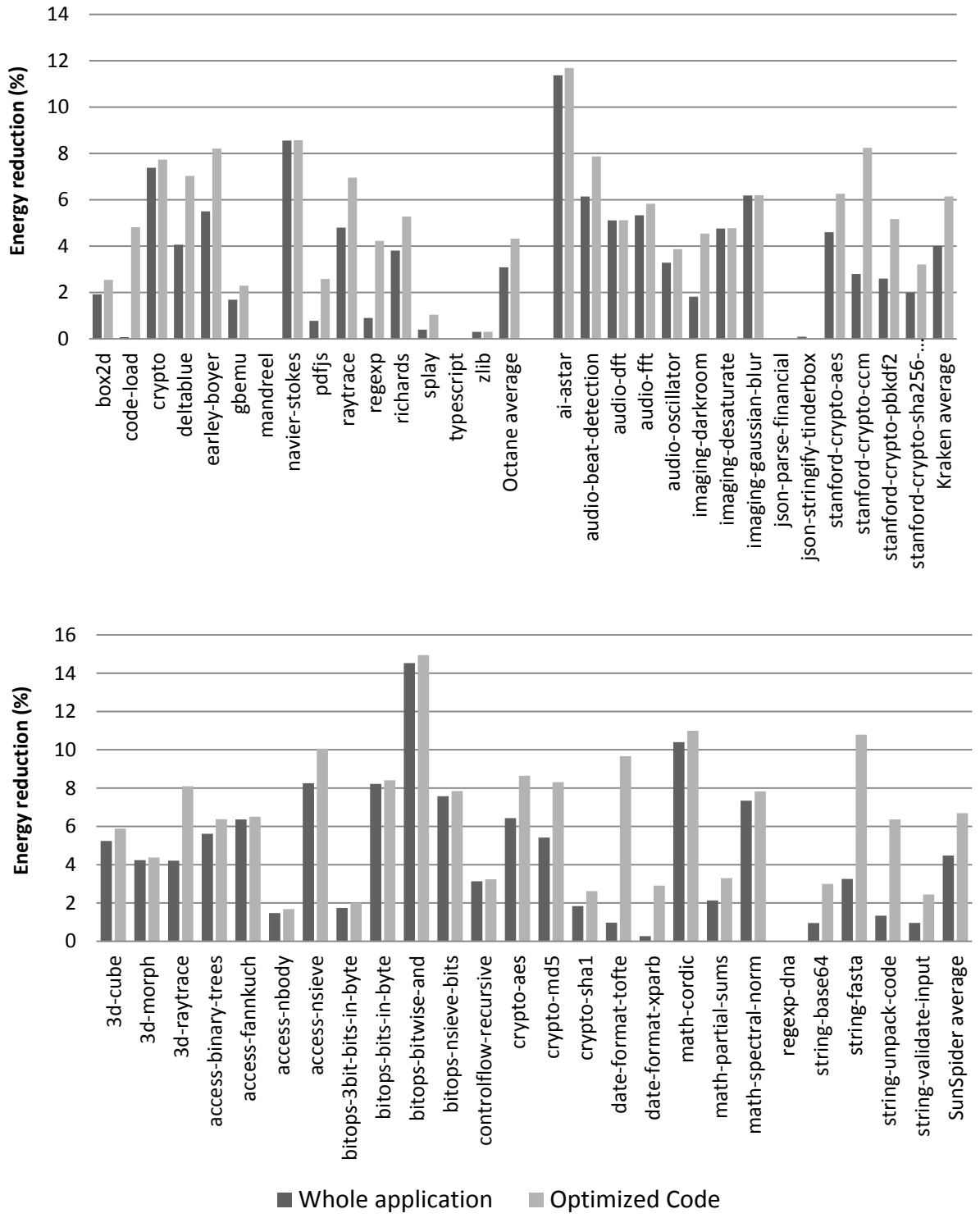

Figure 6.14: Improvement in energy consumption.

Chapter 7

The Class Cache Mechanism

In this chapter, we present the Class Cache, a HW/SW hybrid mechanism that allows the removal of checking operations executed in the optimized code by performing a HW profiling of the types of object properties and objects contained in the *elements arrays*. As explained in chapter 3, the *elements array* is an internal array owned by each object, which contains all the variables of an object that are indexed by a number. Note that this technique takes a different approach than the optimizations presented in the previous chapter. Before, we improved performance by reducing the dynamic instruction count of these checking operations, while now we are removing the checks completely. Both kinds of techniques are complementary and can be implemented together.

7.1 Introduction

The main characteristic of dynamically typed languages such as JavaScript is that variables are neither declared nor bound to a particular type, 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 dynamically profiled information collected previously by the runtime. This collected information consists of the object types seen in particular static points of the program. However, these assumptions need a verification mechanism that introduces some overhead to this specialized code. The operations used by this verification mechanism are referred to as checking operations. For long-running, compute-intensive applications in which the execution is dominated by specialized code, the overhead produced by checking operations is significant.

Although the assumptions verified by checking operations are fulfilled most of the time, they are not removed either because the compiler cannot ensure that the types will not change during the program execution or because the time spent on an exhaustive dynamic analysis of

the application would not compensate the gains of removing some unnecessary checking operations.

In order to improve this part in an effective way, we propose a HW/SW technique that allows the removal of some checking operations in a safe and efficient manner. The basic idea of this technique is that object properties and *elements arrays* that always contain objects with the same type do not need to be type checked. We refer to them as *monomorphic* properties or *monomorphic elements arrays*. Our technique keeps this information at Hidden Class granularity, which means that it tracks which properties or *elements arrays* of every Hidden Class are *monomorphic*. This information is tracked by a new hardware structure called the Class Cache, which is located next to the L1 data cache.

Once these *monomorphic* properties or *monomorphic elements arrays* are identified, the information is passed to the compiler, which can use it to remove some checking operations assuming that the type of these properties will never change. In order to verify these assumptions, when a store that writes a *monomorphic* property or a *monomorphic elements array* is executed, the Memory Unit sends a request to the Class Cache indicating the type of the object to be stored. If this type is different to the one observed in the past, then the corresponding property or *elements array* will no longer be considered as *monomorphic*. In addition, if any optimization (i.e., the removal of any checking operation) has been previously performed considering this property or *elements array* as *monomorphic*, then a HW exception is triggered. This exception is captured by the runtime, which recompiles all the affected functions (i.e., deoptimizes them to a version that does not consider that property or *elements array* as *monomorphic*).

In the rest of this chapter, we first explain the reasons that have motivated us to devise the proposed mechanism. Next, we present the design and functionality of the technique and then, we describe the optimizations that make use of it. Finally, we provide a performance evaluation of these optimizations.

7.2 Motivation

We have observed that in a significant fraction of the benchmarks, the main source of overhead comes from checking operations of objects obtained from properties or *elements arrays*. In Figure 7.1 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 type checks 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 chapter, 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 the 54 benchmarks. In this regard, we have averaged the benchmarks suites of Figure 7.1 only for these selected benchmarks. We can see that these overheads represent 10.7% of the total dynamic instructions. 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 7.1 are performed over *monomorphic* properties or *monomorphic elements arrays* (i.e., those that stay 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 7.2. 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, as we can see in

Figure 7.3. Therefore, the hardware structure (i.e., the Class Cache) that we use to keep the Hidden Class information about *monomorphic* properties or *monomorphic elements arrays* does not have important storage requirements.

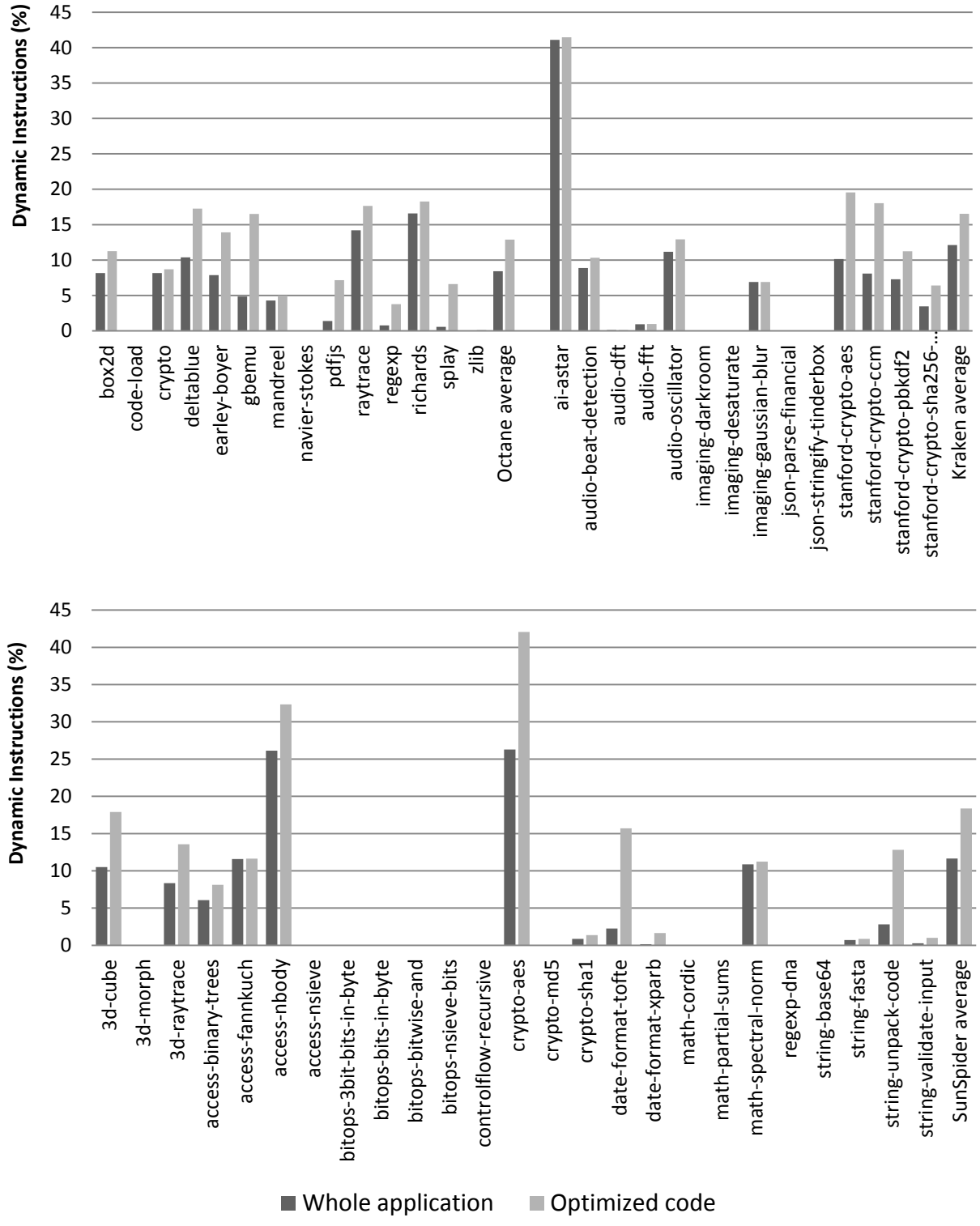

Figure 7.1: Overhead produced by checking and untagging operations after performing object load accesses of properties and *elements arrays*.

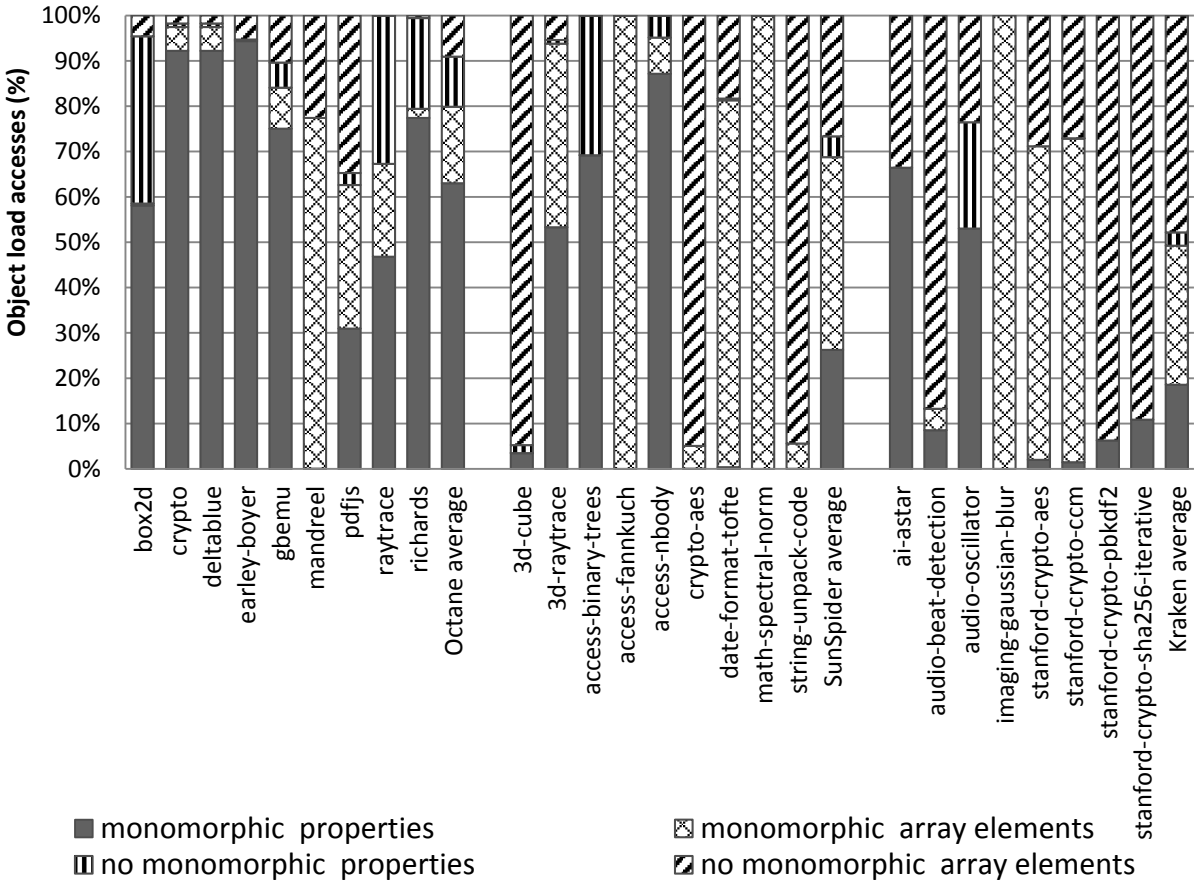

Figure 7.2: Object load accesses to *monomorphic* properties and *monomorphic elements arrays*.

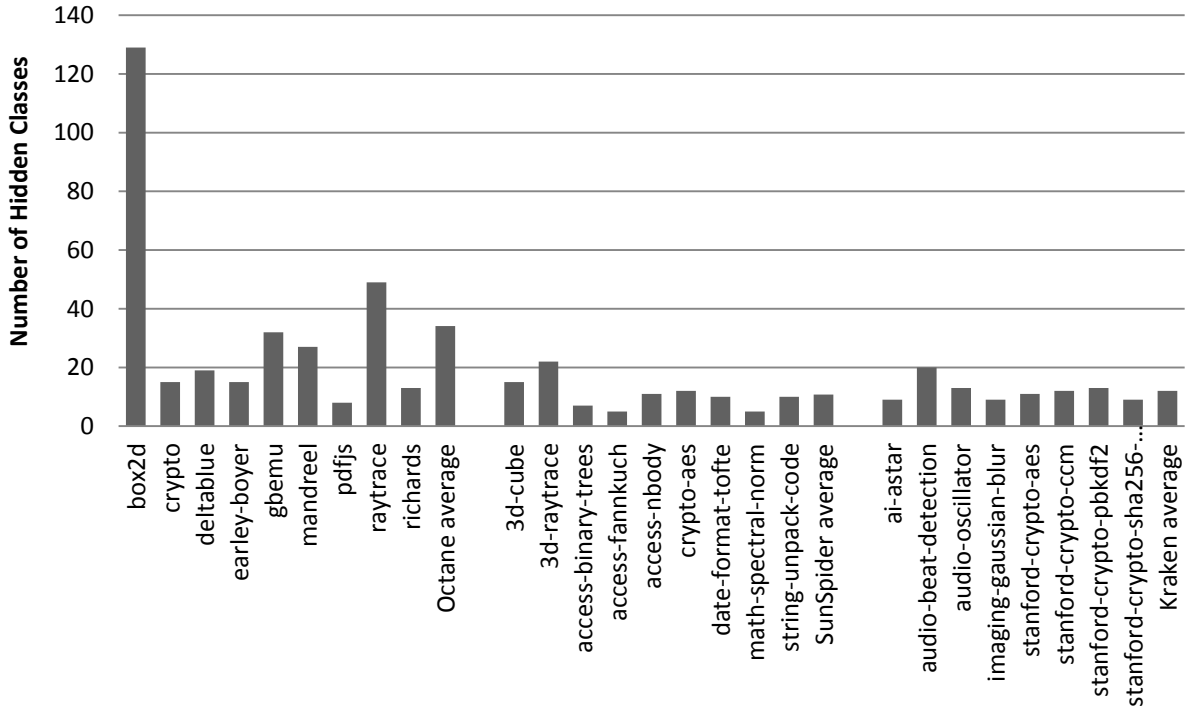

Figure 7.3: Number of different Hidden Classes used for each benchmark.

7.3 The Class Cache Mechanism

The Class Cache mechanism is based on a small, special new HW/SW structure that keeps information about *monomorphic* properties and *monomorphic elements arrays* at Hidden Class level. In other words, it stores which properties and *elements arrays* have the same type (i.e. a particular Hidden Class or SMI) for all the objects of the same Hidden Class during the execution of a program. 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 (i.e., a Hidden Class that is not yet present in the structure) 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.

7.3.1 The New Structures

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

7.3.1.1 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 3.2.1, 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 (one is enough most of the time as we will show later). 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, which need 48 bits for their representation because they are memory addresses of the Hidden Class descriptors. Instead, the identifiers for the Hidden Classes that we use 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 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 by the Crankshaft compiler. This field is initialized to zeros, indicating that no speculation has been applied yet.
- **Prop1 … Prop7**: Seven 1-byte fields that contains the *ClassID*s 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 7.1 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 properties. 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 7.1, 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.*

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

Table 7.1: Class List Structure**.**

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 table. Note that the Class List entries are together in the same 1 MB memory region (i.e., 16 bytes reserved for each entry) 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.

7.3.1.2 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 object 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 *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, which need a source operand (i.e., *regArray*) to indicate which of the four registers they use.

Appendix B details the mnemonics of these new instructions. Note that identifying these object stores is straightforward for the dynamic compiler, since it knows the semantics of the code being generated.

7.3.1.3 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* (i.e., a *movStoreClassCache* or an *movStoreClassCacheArray* instruction) 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 (2 in case of a

movStoreClassCacheArray instruction*)* and the *ClassID* of the object to be stored.

In Figure 7.4 we depict a Class Cache request for a *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 [47] and both Nitro [63] and Mozilla JavaScript engines [45] 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.

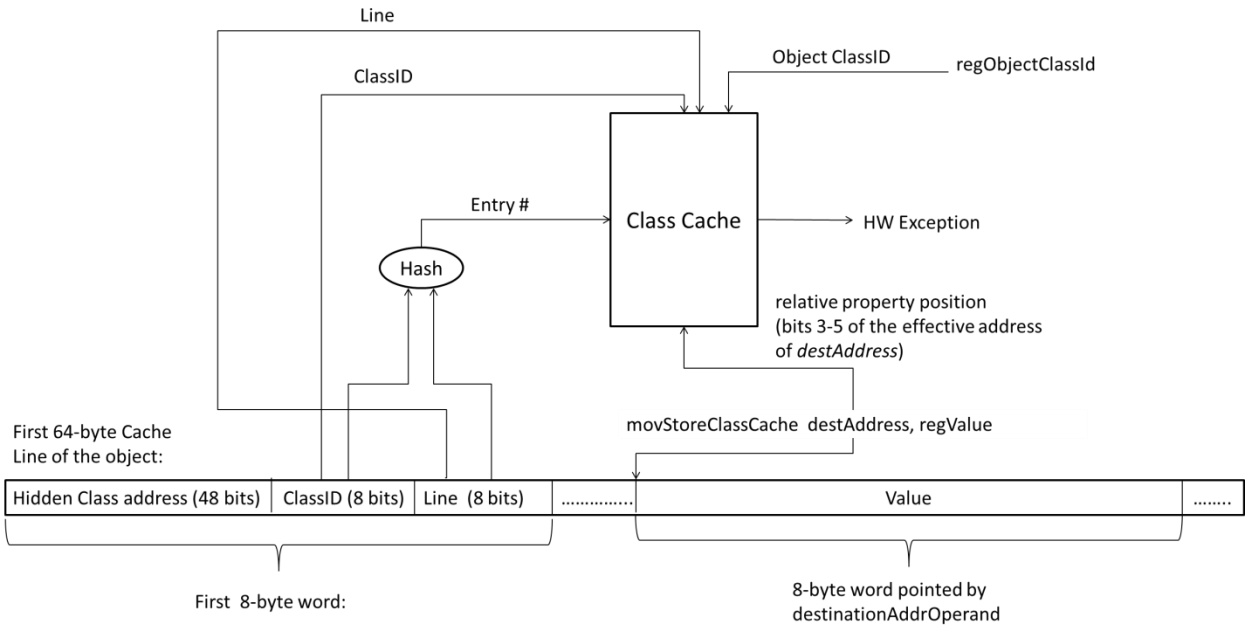

Figure 7.4: Block diagram of a Class Cache access for a *movStoreClassCache* instruction.

In Figure 7.5 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 *2* and *0*, respectively. As commented above, the field inside the Class Cache that is

reserved for the *elements array pointer* (i.e., the second property of each Hidden Class) 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 *elements 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 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 as *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.

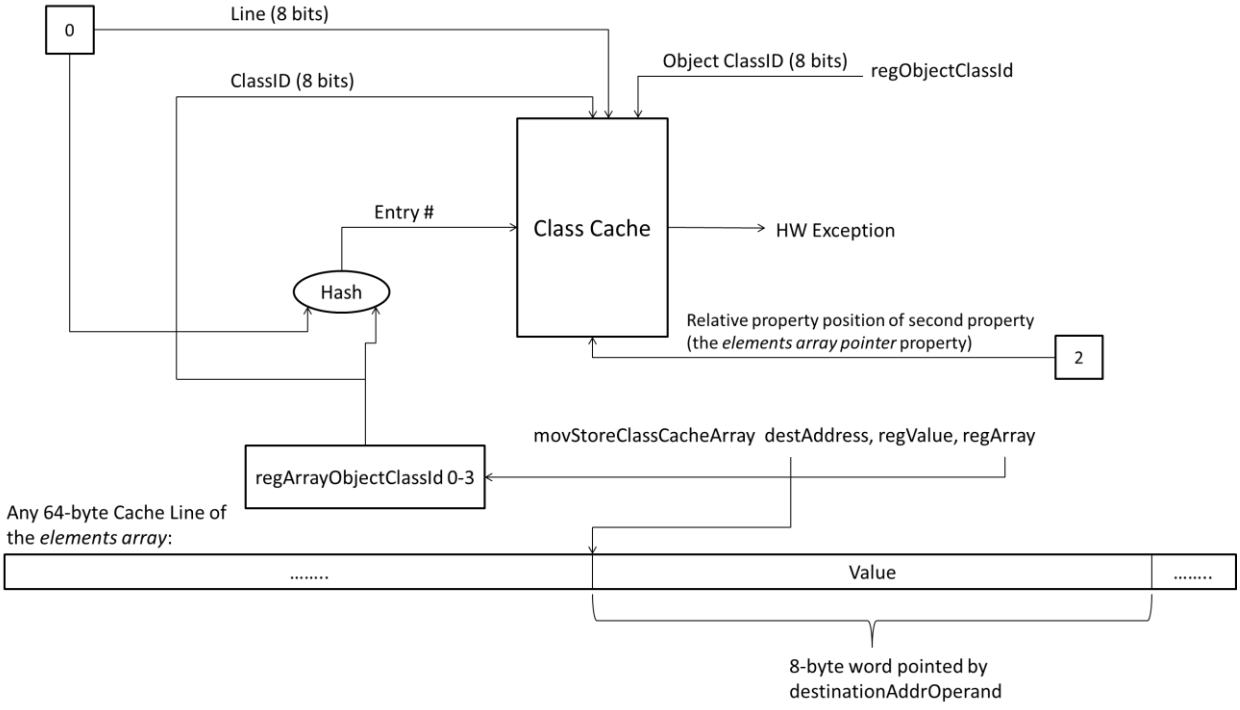

Figure 7.5: Block diagram of a Class Cache access for a *movStoreClassCacheArray* instruction.

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 7.6. 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 7.6. 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. 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.

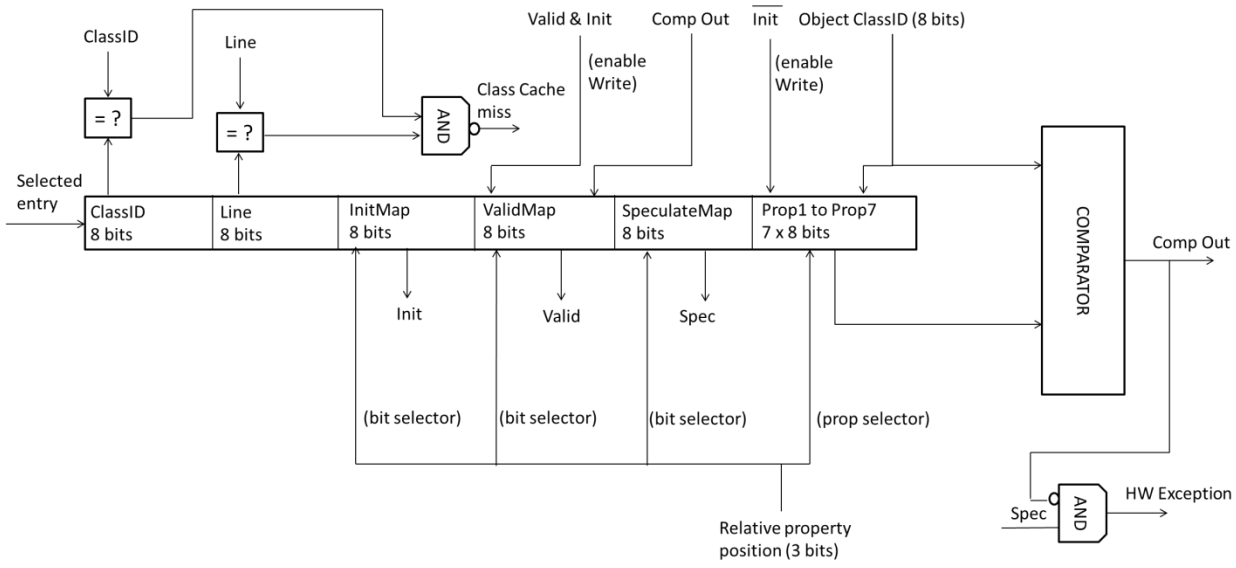

Figure 7.6: Scheme of a Class Cache entry.

7.3.2 How the Mechanism Works

As explained in chapter 3.2, 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 7.3.3), based on the assumption that the *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 7.7 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 a *monomorphic 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.

Although this technique introduces some overheads (extra *movClassID* and *movClassIDArray* instructions, larger objects, managing misspeculations, 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.

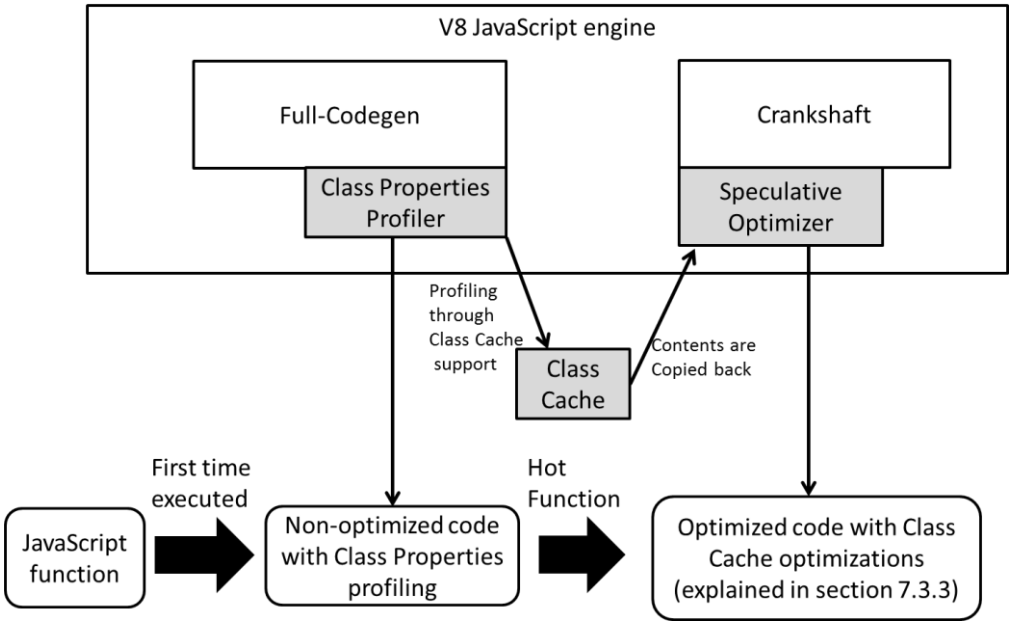

Figure 7.7: Optimization process.

7.3.3 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.

Check Maps Elimination

We remove the *Check Maps* operations that verify the *monomorphic* properties or *monomorphic elements arrays*. Note that this optimization also includes the *Check Maps* operations that are necessary for the *Number Untags* commented in section 5.2.2.

Check Non-SMI Elimination

We remove the *Check Non-SMI* operations that verify *monomorphic* properties or *monomorphic elements arrays* that are profiled as non-SMI. Note that this optimization also includes the *Check Non-SMI* operations that are necessary for the *Number Untags*.

Check SMI Elimination

We remove the *Check SMI* operations that verify the *monomorphic* properties or *monomorphic elements arrays* that are profiled as SMIs. Note that this optimization also includes the *Check*

SMI operations that are necessary for the *SMI Untags*.

7.3.4 An Example of the Proposed Optimizations

In Figure 7.8 we show an example of our optimizations for the *findGraphNode* function explained in section 5.2.3. As we have seen in Table 7.1, our mechanism has optimized this function by considering that the 6th property (*position)* from *GraphNode* and the *elements array* from *NodeList* are *monomorphic*. The *position* property of *GraphNode* has been profiled with a single *ClassID* (i.e., *classPostion*) and all the objects stored in the *elements array* of *NodeList* have also been profiled with a single *ClassID* (i.e., *GraphNode*).

The left part of Figure 7.8 shows that instructions I17-I21 and instructions I35-I39 are used to perform a *Check Non-SMI* and a *Check-Maps* operations to the values obtained from the *position* property of *GraphNode* objects. The right part of Figure 7.8 shows that these instructions are removed by our Class Cache mechanism, because up to this point all *position* properties of *GraphNode* are *monomorphic* properties that contain objects that belong to the *classPosition* Hidden Class.

On the other hand, the left part of Figure 7.8 shows that instructions I29-I33 are used to perform a *Check Non-SMI* and a *Check-Map* operations to the values obtained from the *elements array* of *NodeList* objects. The right part of Figure 7.8 shows that these instructions are removed by our Class Cache mechanism, because up to this point the *elements array* of *NodeList* is *monomorphic* since all objects contained in this array belong to the *GraphNode* Hidden Class.

7.4 Performance Evaluation

In this section, the benefits of the Class Cache mechanism are evaluated for a subset of Octane, Kraken and SunSpider benchmark suites. The V8 JavaScript engine has been extended to include the proposed optimizations. As in section 5.2, the reported results refer to the tenth iteration, in order to achieve a steady state of the benchmarks.

In the experiments below, 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.

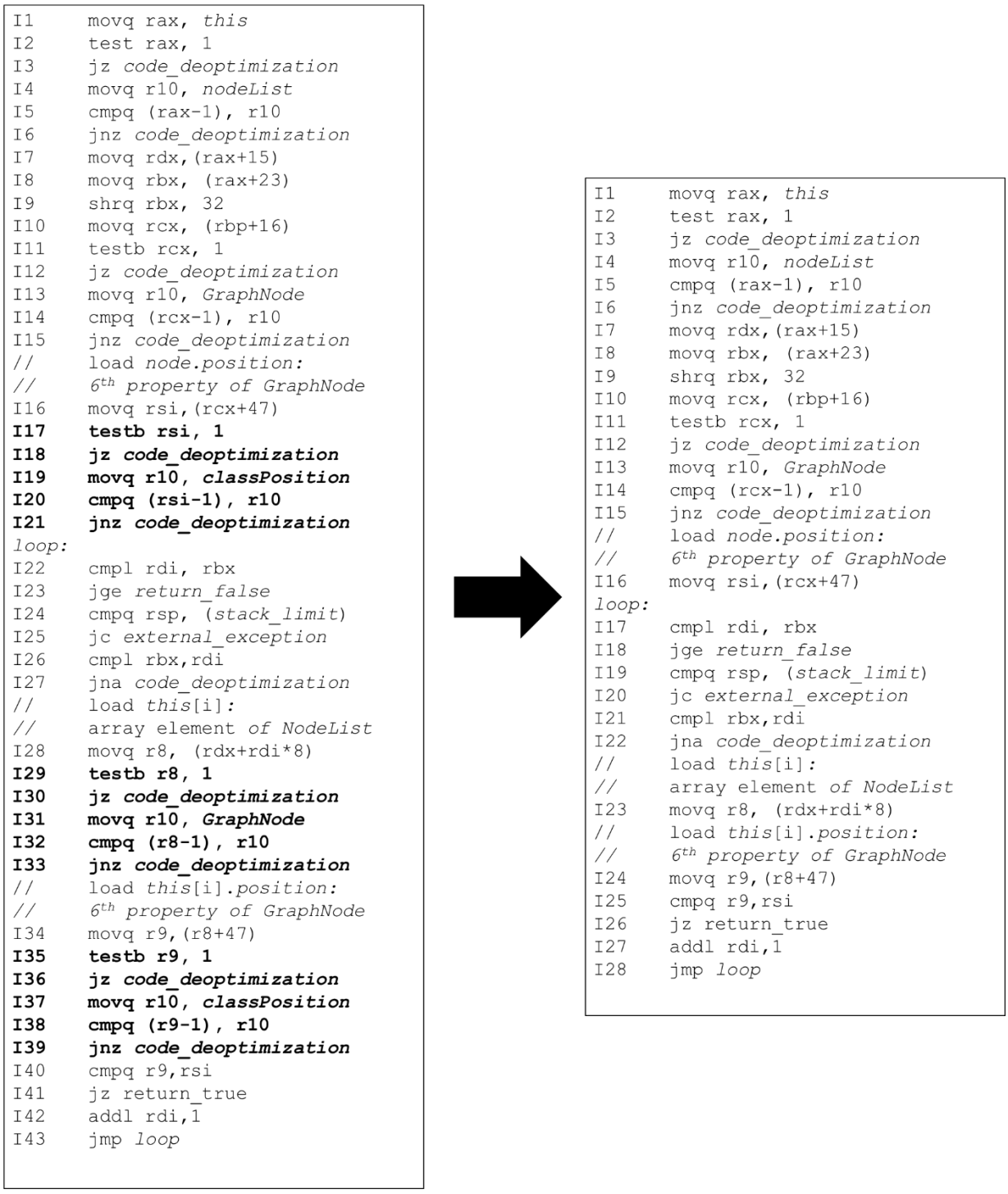

Figure 7.8: Example of the proposed optimizations.

As evaluation methodology, we have implemented the Class Cache mechanism in V8 JavaScript engine, in order to remove the execution of the corresponding checking operations. Besides, we have also inserted additional *mov* x86-64 instructions before the corresponding stores to properties or array elements, in order to obtain their *ClassID* parameters. Finally, in a separated simulation of the Class Cache, we have quantified the number of dynamic instructions and cycles taken by all the misses of the Class Cache.

7.4.1 Dynamic Instruction Count Improvements

In this section we analyze the dynamic instructions that are executed with and without our technique. Figure 7.9 shows the results for the three benchmark suites, considering both the whole application and the optimized code. Our technique reduces the number of instructions in optimized code by 7.5% on average and up to 21% in the best case. We achieve similar improvements for all three benchmarks suites. If we consider not only the optimized code, but the compiler, garbage collector and the rest of the runtime as well (i.e., the whole application), instruction count gains are still important with an average improvement of 5.2% and up to 20.5%.

Note also that our technique reduces the overhead quantified in section 7.1 by 51% on average. The overhead that is not removed by our mechanism consists basically of checking operations of properties or *elements arrays* that are not *monomorphic*.

As we have seen in Section 5.2, the percentage of dynamic checks in Kraken is similar to Octane and SunSpider. However, as shown in Figure 7.9, Kraken gets 5.7% instruction count reduction for the whole application, which is a bit better than the other suites. This is mainly due to the fact that Kraken has a higher fraction of dynamic instructions in optimized code. There are benchmarks that do not have much optimized code and therefore our technique provides small benefits for these cases. However, as JavaScript applications become more compute intensive, we expect that the relative overhead of the compiler will decrease and the weight of the optimized code will become more important, which will increase the benefits of our technique.

7.4.2 Cycle Count Improvements

In this section we evaluate the performance benefits of our technique, as measured with Marss [2] cycle-level microarchitectural simulator. Figure 7.10 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

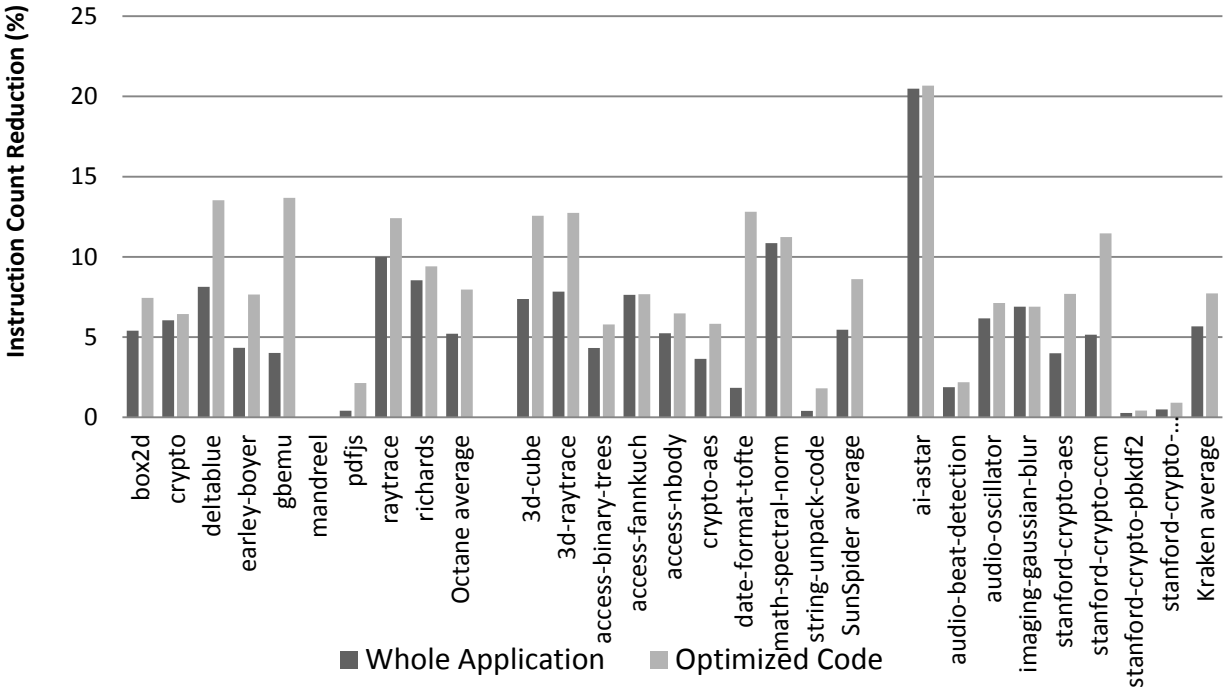

Figure 7.9: Improvement in number of instructions.

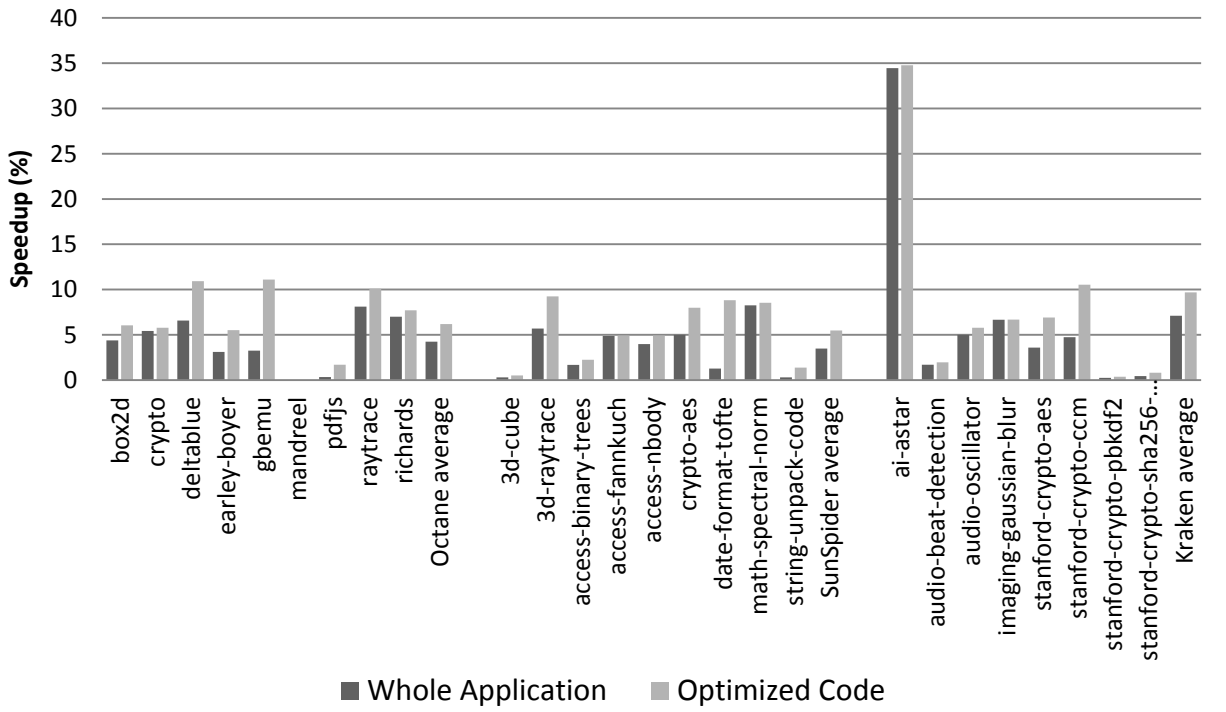

Figure 7.10: Improvement in number of cycles.

housekeeping tasks decreases.

These results correlate significantly with the ones presented in the previous section for dynamic instructions. For Octane and SunSpider suites, the obtained speedup is somewhat lower than the dynamic instruction count reduction. The reason is that some of these checking

operations are not in the critical path of the application, so they have a small impact on performance. On the other hand, we see the opposite effect for Kraken suite, for which the speedup is higher than the reduction in dynamic instruction count. 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. More than half are *Check-Maps* operations and as commented in chapter 5, 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 accesses, 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.

7.4.3 Energy Reduction

Figure 7.11 shows the energy savings of our technique for the three benchmark suites, which are measured through the McPAT simulator [53]. We used CACTI [55] 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

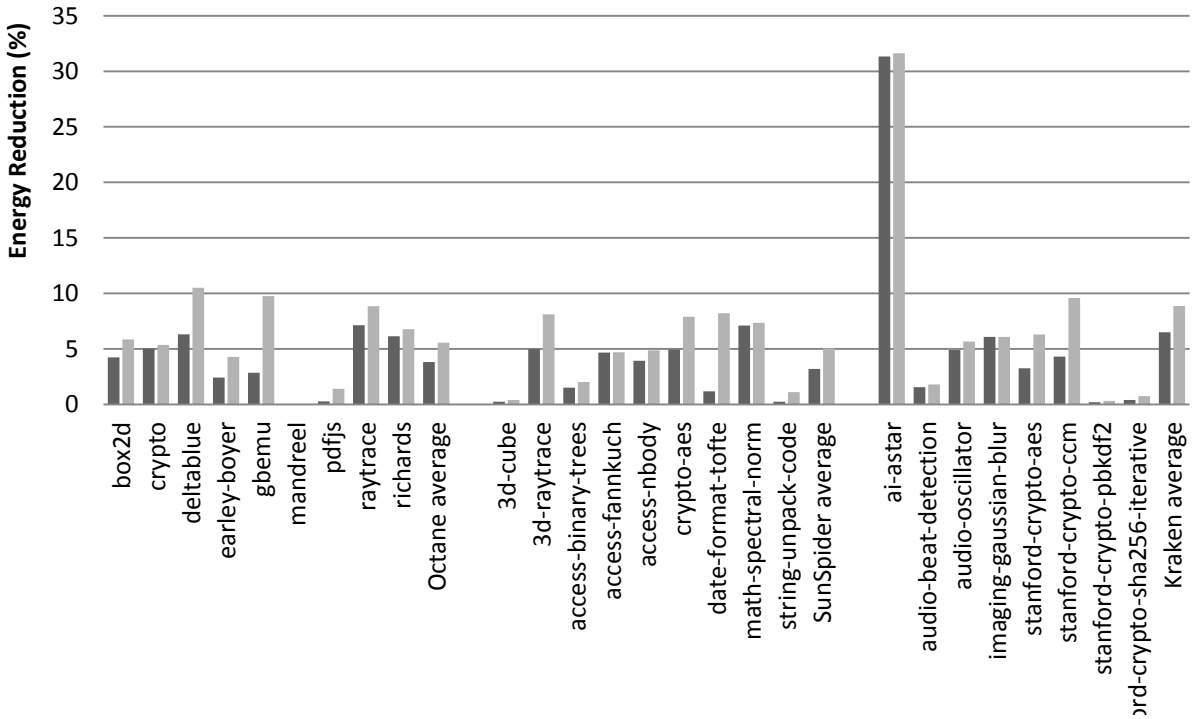

Figure 7.11: Improvement in energy consumption.

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.

7.4.4 Incurred Overheads

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

7.4.4.1 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.

7.4.4.2 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 a rather slow operation. However, the hit rate of a Class Cache of just 128 entries and 2-way set associativity is higher than 99.9% for all benchmarks and thus the penalty of misses is negligible.

7.4.4.3 Misspeculations and Recompilations

When a misspeculation occurs for a particular property or *elements array* (i.e., a property or *elements array* that has been used to optimize at least one function changes its profiled *ClassID*), a hardware exception is triggered and all the functions that have been optimized using that property or *elements array* have to be recompiled. This exception is captured by the runtime, which manages this recovery mechanism. Identifying the functions to recompile is straightforward because these are kept in the *FunctionList* field of the Class List. Note that all code executed until this point is correct, and by recompiling the speculative functions, the code executed in the future will also be correct. In other words, our scheme never executes incorrect code that has later to be squashed.

Since our results report the tenth iteration of each benchmark, there is not any misspeculation at that point. However, we have verified that in the first iteration, the number of misspeculations is negligible for all the benchmarks. The main reason is that at the

beginning of the application all the functions are executed in non-optimized code. Therefore, during this period the Class Cache performs a very accurate profiling of the *monomorphic* properties and *monomorphic elements arrays*, which does not differ much for the rest of execution.

7.4.4.4 Larger Objects

The objects whose size is higher than 64 bytes (one cache line) require an extra memory word for each extra line (i.e., because the insertion of *ClassID* and *Line* fields), as described in section 7.3.1. The fact that a small fraction of the objects are slightly larger (up to 11% larger) may affect the L1 Data Cache hit rate. However, most of the object property accesses (79%) target the first cache line, as we can see in Figure 7.12. Therefore, the L1 Data Cache miss rate hardly increases and this overhead is not relevant.

7.4.5 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 [53] and CACTI [55]. Similarly, the energy

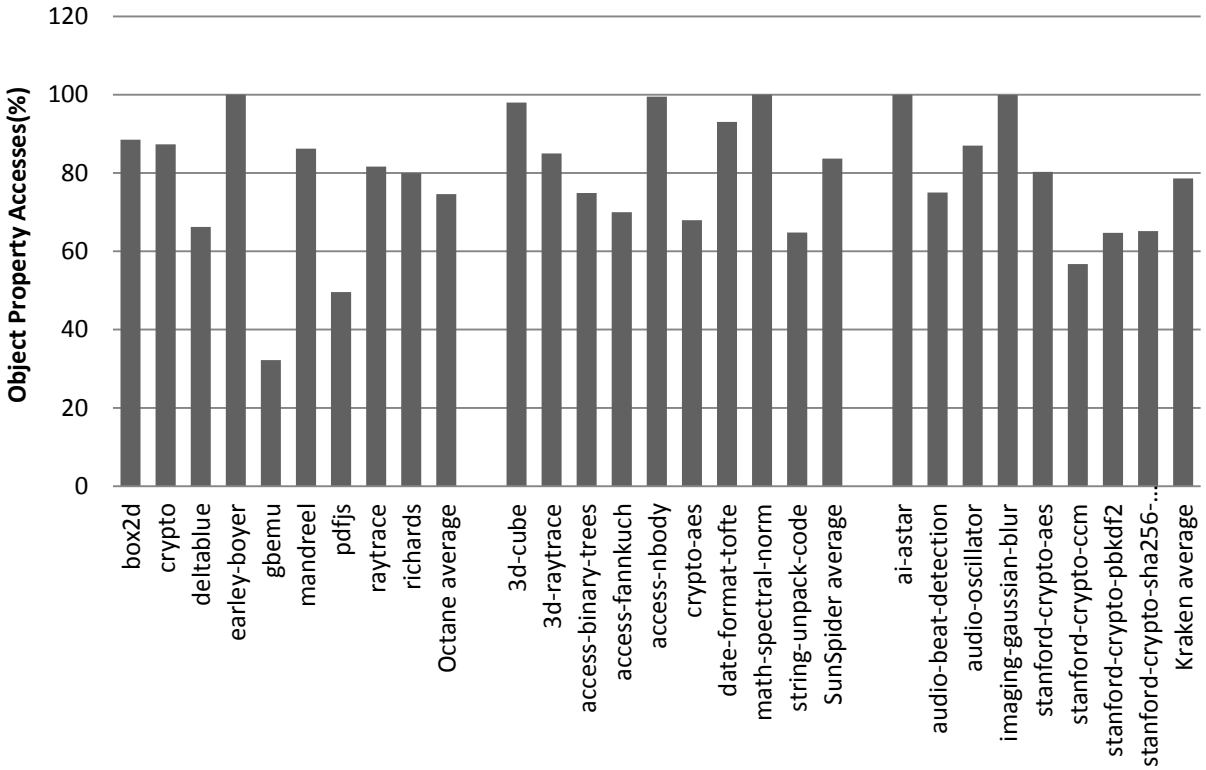

Figure 7.12: Object property accesses that target the first cache line.

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. In particular, several additional instructions (more than seven micro-operations) would be needed for every store to an object, to perform the following steps:

1. Load the class identifier of the property or *elements array* (*ClassID*).
2. Load the relative cache line (*Line*).
3. Load the class identifier of the object to be stored (*Object ClassID*).
4. Hash the *Line* with the *ClassID* to index the corresponding Class Cache entry.
5. Load the Class Cache entry.
6. Compare the first two fields of this entry (i.e., *ClassID* and *Line* attributes) with the *ClassID* and *Line* fields of the object.
7. If they are not equal, then branch to the Class Cache miss routine.
8. Check the corresponding *InitMap* bit. If this bit is 0, then write the *Object ClassID* to the corresponding *Prop1-Prop7* field of the entry and set the *InitMap* bit to 1. Otherwise, go to step 9.
9. Check the corresponding *ValidMap* bit. If this bit is 1, compare the *Object ClassID* with the corresponding *Prop1-Prop7* field of the entry.
10. If they are not equal, set the *ValidMap* bit to 0 and check the corresponding *SpeculationMap* bit.
11. If the *SpeculationMap* bit is 1, branch to the routine that deoptimizes the functions that have been optimized considering the stored property as *monomorphic*.

7.5 Conclusions

In this chapter, 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), dynamic instruction count reduction (7.5% on average) and energy consumption (6.5% on average) for optimized JavaScript code.

Chapter 8

The Property Cache Mechanism

The execution of short JavaScript web applications for event-driven scripts is dominated by non-optimized code, helper routines and runtime tasks (i.e. compilation tasks). Furthermore, a significant fraction of time is dedicated to access object properties, due to the fact that program variables are not tailored to any specific type, which is known as the dynamic binding problem. In this regard, when a property is accessed by the first time, the type of the object has to be obtained, in order to compute the correct address for that property. Then, this access is improved by specializing the code for that particular type. Note that every time that a particular access encounters a new type, this process can be very time-consuming.

In this chapter, we present a HW/SW mechanism that performs the accesses to object properties in a more efficient manner than state-of-the-art techniques.

8.1 Introduction

In dynamically typed languages, variables are neither declared nor bound to a particular type (i.e., Hidden Class), and their types may change during the execution. One of the major issues with this feature is that when a property (i.e., an attribute or method of an object) of a particular variable is accessed, the corresponding address (i.e., offset) for that property is not known at compile time. Therefore, a time-consuming process is needed to obtain the corresponding address according to the type of the object that is contained in the variable.

The state-of-the-art technique used by current JavaScript virtual machines to address this overhead is known as Inline Caching [40][15]. This technique has a twofold purpose: record information concerning the types of objects and improve the performance of the system lookup routine used to disambiguate the type of objects when they are accessed. As explained in Chapter 3, both Full Codegen and Crankshaft compilers from V8 apply this technique, but in a different manner.

In the code produced by Full Codegen, each property access is represented by a x86-64 *call* instruction, which is constantly patched by the runtime. The first time that a particular property access is performed, this *call* instruction targets a lookup routine that performs a sequence of steps that determine the Hidden Class of the object and find the offset for that property, in order to perform the access. Then, this lookup routine is specialized for that particular Hidden Class, in order to accelerate future accesses. This code is preceded by a checking operation that verifies that the Hidden Class of the object is the expected one. This specialized code is kept in a software structure called Inline Cache (IC), which is unique to each property access. The *call* instruction is patched to point to this Inline Cache and therefore, the subsequent accesses are substantially faster as long as the Hidden Class of the object keeps being the same. Otherwise, the default lookup routine is executed.

On the other hand, the information (i.e., the Hidden Class of the objects) recorded by the Inline Caches during the process explained above is later used by Crankshaft to perform more aggressive optimizations for hot code. In this regard, the specialized code generated by Crankshaft 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. Therefore, *Check Maps* operations are also introduced in this specialized code in order to verify that the encountered Hidden Class is the expected one; otherwise (i.e. when a *Check Maps* fails), the optimized code falls back to non-optimized code through a deoptimization bailout.

The technique presented in this chapter takes an innovative HW/SW approach to remove most of the overhead produced by the Inline Caching mechanism for short-running event-based applications. Concretely, it targets loads of object properties, which is the most frequent scenario. This new approach is based on a small hardware structure called the Property Cache that caches the addresses of the most commonly used object properties, which are also stored on a runtime-built software structure. Therefore, when a particular object property is found in the Property Cache, the access is performed with minimal overhead and without executing any lookup routine.

In the rest of this chapter, we first explain the reasons that have motivated us to devise this new technique. Next, we present the design and functionality of the mechanism and finally, we evaluate the performance of this technique.

8.2 Motivation

In Figure 8.1 we show the overhead of to the Inline Caching mechanism for object property loads, for both non-optimized and optimized code. As explained in section 5.1, we have chosen for our experiments the first and fourth iterations of Octane and JSBench suites, respectively, in order to reflect typical short-running event-based applications. We can see that this overhead is significant, being 12% on average.

On the other hand, note that the offsets of all properties of any Hidden Class are known before the corresponding loads are performed. In this regard, we propose a mechanism to obtain the corresponding offsets for each object property load in an efficient manner, which require small hardware extensions. Unlike Inline Caching, our mechanism does not require any dynamic profiling neither the dynamic creation of specialized code for property loads. In addition, it is not speculative.

The proposed technique does not target stores because the first store to a property creates a new Hidden Class that contains the new property and the corresponding offset is not known until then. Besides, when a store is executed, a write barrier operation is performed, in order to notify the garbage collector of new pointers. These issues would significantly reduce the benefits of the proposed technique and increase its complexity.

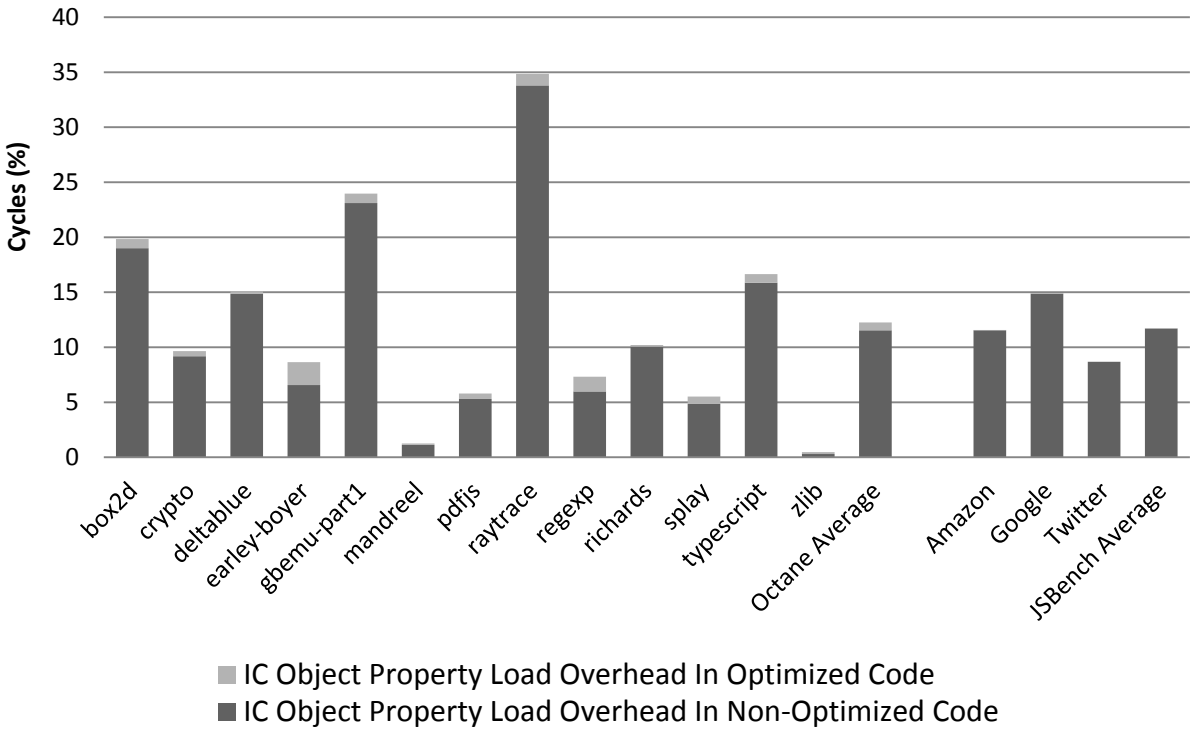

Figure 8.1: Object property loads overhead due to the Inline Caching mechanism.

8.3 The Property Cache Mechanism

In this section we present the Property Cache, a HW/SW mechanism that reduces some of the most important inefficiencies due to dynamic typing. First, we present a general overview of the technique. Then, the required software and hardware structures are presented. Next, the entire process is described and finally, we detail some particularly important scenarios.

8.3.1 Overview

Our mechanism is based on obtaining on demand and efficiently the offsets for each load to an object property which is indexed by name. In order to obtain these offsets, we keep in memory a data structure called Property List, which contains all property names and their corresponding offsets for all Hidden Classes. In order to efficiently access this information, we extend the hardware with two new structures that cache the information needed to compute the effective address of the latest accessed properties: the Property Cache and the Prototype Cache.

For every property load, we first check these caches, and in case of hit, the address is obtained in a very efficient manner. Otherwise, a software trap is generated and control is transferred to a subroutine that traverses the Property List in order to obtain the information related to this property.

8.3.2 The New Structures

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

8.3.2.1 The Property List

The Property List is a software structure that contains as many entries as different property names encountered during the execution of a JavaScript program. In figure 8.2a, we show an example of the Property List for the JavaScript program explained in section 5.3.1. Each entry contains the following information:

- **Property name**: The name of the property.
- **Property identifier**: a number that identifies the *property name*.
- **Hidden Classes table pointer**: A pointer to a table that contains as many entries

as the number of Hidden Classes that use this *property name*. For each entry of this table there are two fields: the *Hidden Class identifier* and the corresponding *offset* of this property in this Hidden Class (see Figure 8.2b1-b7). In other words, each pair of *Hidden Class identifier* and *property identifier* has a particular *offset*.

Besides, there is a special register that has a pointer to this Property List in memory, in a similar way that there is a pointer to memory translation tables (i.e., the *Property List special register* in Figure 8.2a).

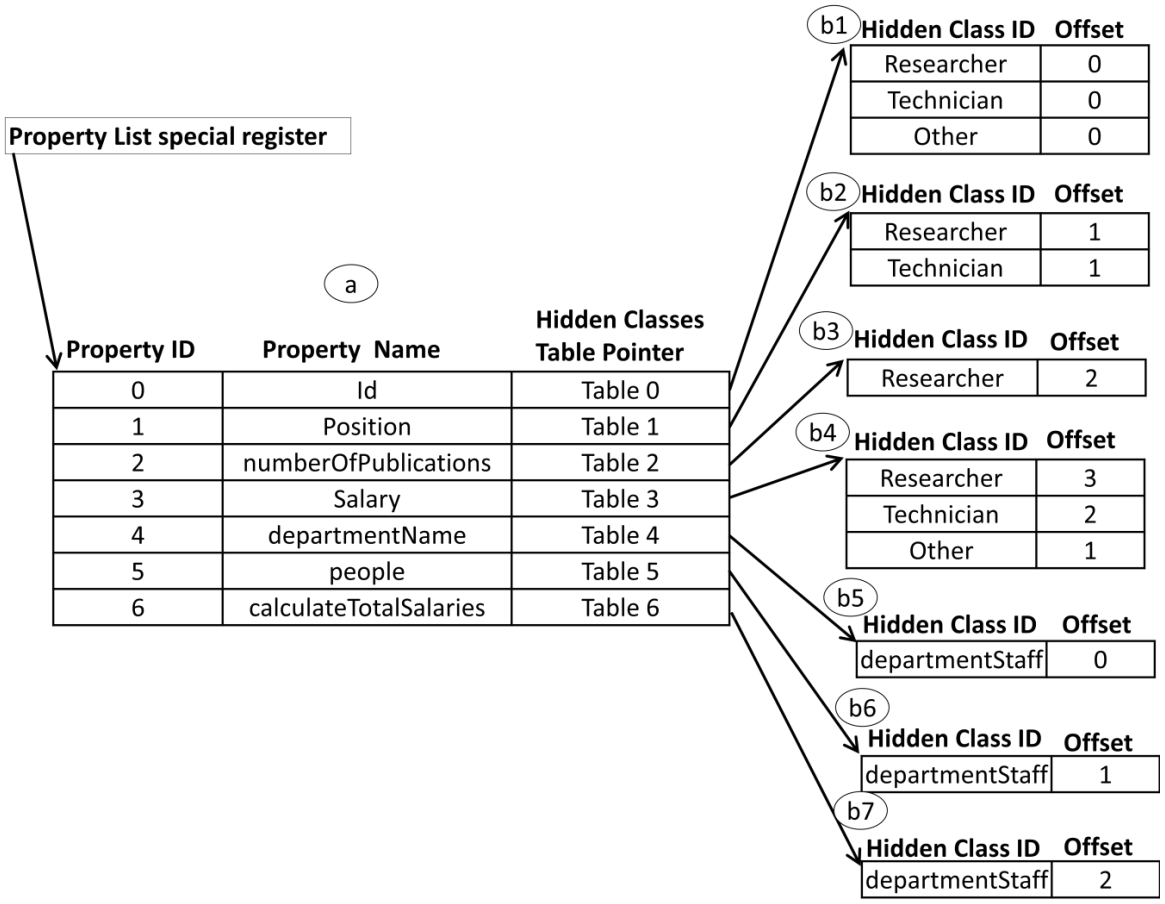

Figure 8.2: Property List structure.

8.3.2.2 The Property Cache

The Property Cache keeps the most recently used information of the Property List. In Figure 8.3, the basic scheme of a Property Cache entry is shown. When a particular cache entry is selected through a hash function, the *property identifier* and the *Hidden Class identifier* fields are used as cache tags. If a hit occurs, then the *offset*, *P* and *I* fields are returned. *P* field indicates whether the property comes from a prototype object instead of the object itself (see section 3.1). The *I* field indicates whether the property is contained in a *property dictionary*

collection structure (see section 3.2.1). If a miss occurs, then the control is transferred to the runtime to obtain the *offset* from the Property List and this information is stored in the Property Cache by replacing one of the entries. Section 8.3.4.2 describes the miss subroutine.

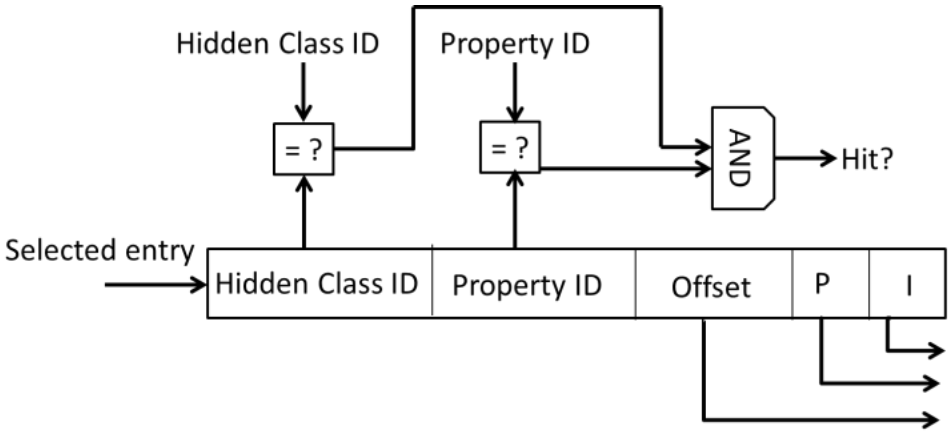

Figure 8.3: Scheme of a Property Cache entry.

8.3.2.3 The Prototype Cache

When the property comes from a prototype object instead of the object itself, we need to obtain the address of the prototype object in addition to the offset since the load instruction only knows the address of the object but not the address of its prototype property (see section 3.1). However, prototype object addresses are not kept in the Property Cache for space efficiency reasons because the majority of requested properties are contained in the object itself and thus, the prototype address is not required for these cases. For this purpose, we use the Prototype Cache, which contains the most recently used prototype addresses.

In Figure 8.4, a block diagram of a Prototype Cache entry is shown. The *property identifier* and the *Hidden Class identifier* fields are used as cache tags. In case of a hit, the corresponding prototype address field is transferred to the output. In case of a miss, a software exception is generated only when the P bit from the Property Cache is set to 1 (which means that the object prototype address is necessary).

8.3.2.4 Two New Machine Instructions

Besides this new hardware support we extend the ISA with two new special machine instructions, which are used to interact with this hardware. We call these two instructions *specialMovMap* and *specialMovOffset*, whose mnemonics are detailed in appendix C.

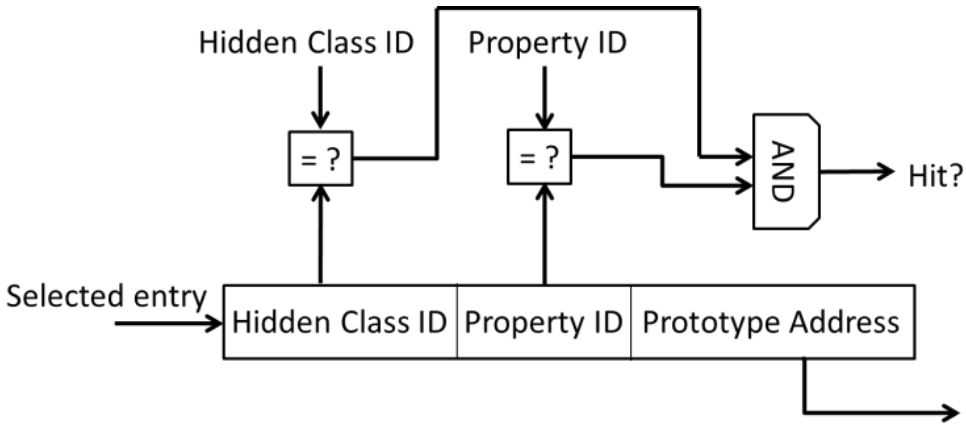

Figure 8.4: Scheme of a Prototype Cache entry.

SpecialMovMap instruction has a source memory operand that is the address of an object (the object to which the property belongs) and returns its *Hidden Class identifier*, which is located in the first 8-byte word of the object. This *Hidden Class identifier* is stored into the first 8-byte word of a special 64-byte register that we call *hiddenClassReg*. Moreover, the following 56 bytes of this register are filled with the rest of the cache line obtained from the memory request, which corresponds to the first properties of the object. In this regard, heap-allocated values need to be cache line aligned, which is a common constraint of current JavaScript engines. However, in rare occasions the source operand might contain a SMI value, instead of an object address. We can easily identify these cases because the least-significant bit of a SMI value is *0*. Therefore, when the hardware detects that this bit is *0*, a special value (i.e., to indicate that it is a SMI) is directly stored to the *hiddenClassReg* register, instead of obtaining it from memory.

SpecialMovOffset instruction has two source operands, the address of the object and the *property identifier*, and a destination operand that is a register where the value of the property will be stored. This instruction performs the access to the value of the property and stores it in the destination register. In this way, each property load is translated to a sequence of these two instructions (the reason for having two separate instructions rather than a single one is described in Section 8.3.5.1).

8.3.3 How The Mechanism Works

In Figure 8.5, we show the main components of our mechanism and the steps taken by a load of a property. First, (1a) the *specialMovMap* instruction is executed in order to obtain the *Hidden Class identifier* of the object to which the property belongs, and (1b) store it in the

special 64-byte register *hiddenClassReg,* along with the rest of cache line. Then, (2) the *specialMovOffset* instruction is executed, which results in the following actions: First, (3) the *Hidden Class identifier* and the *property identifier* are used to index both the Property and Prototype Caches. Although the Prototype Cache is always accessed, its information will be used only when the accessed property is contained in a prototype object, instead of the object itself. In this regard, (4) the base address used to compute the effective address of the accessed property comes from either the object address, the prototype object address, or the second 8-byte word of the *hiddenClassReg* register (i.e., this word contains the *property pointer*, but only for those objects that have their properties stored in a *property dictionary collection*), which is selected by both the *P* and *I* signals. Then, this value is added to the *offset* (5) obtained from the Property Cache, in order to compute the effective address, which is used to perform a memory request to read the property value (6a). Then, this value is written to the destination register (7).

As commented in the previous section, when the *specialMovMap* instruction stores the result in the *hiddenClassReg* register, not only the *Hidden Class identifier* is kept, but also the entire cache line is transferred to this special register, in order to optimize the access to nearby properties. The number of prefetched properties is equal to the size of the cache line (i.e. the number of the properties that fit in a cache line) minus one. Therefore, when the *offset* obtained by the Property Cache is less than or equal to this number of prefetched properties, the accessed property is obtained from the *hiddenClassReg* register and a new access to memory is saved. We illustrate this situation in the step 6b of Figure 8.5.

In rare occasions, it may happen that the requested property does not exist for this object (i.e., it is because a program error) and therefore, it is not found in the Property List. When this occurs, the runtime sets the *NoExist* signal to 1 and the special *undefined* value is returned. On the other hand, when the *hiddenClassReg* contains a SMI (see previous section), the special *undefined* value is also returned. We illustrate this situation in step 6c of Figure 8.5.

8.3.4 The New Runtime Subroutines

In this section we detail the subroutines that are used to manage the Property List.

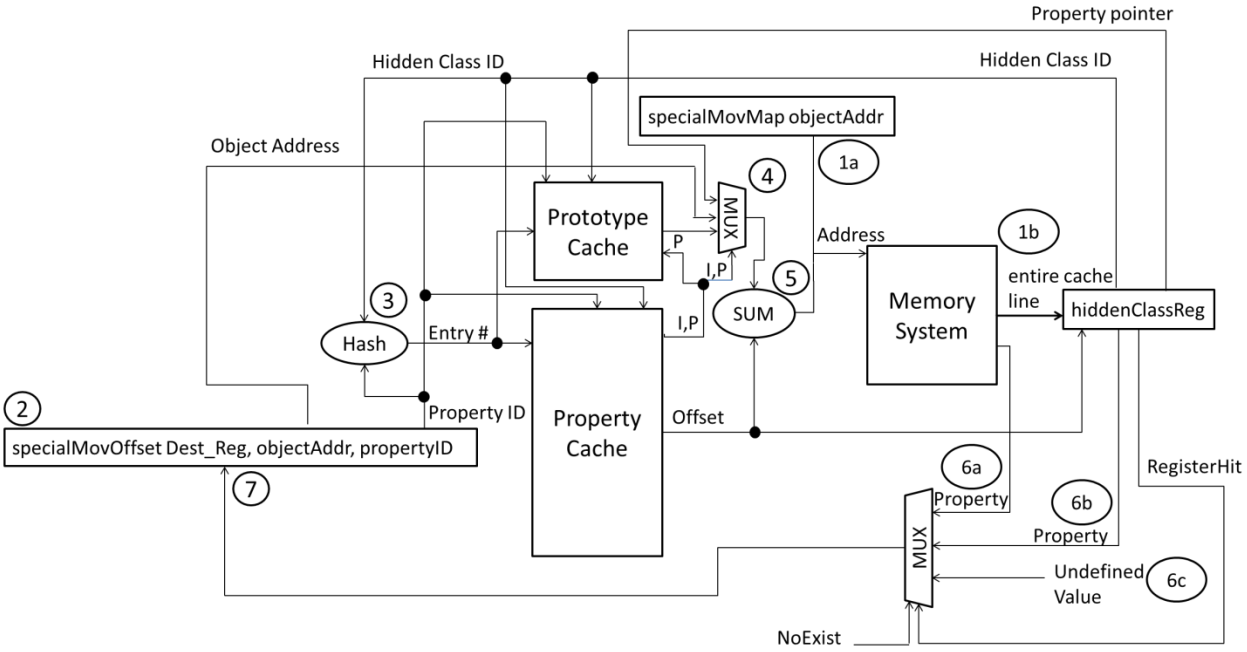

Figure 8.5: Block diagram of the proposed mechanism.

8.3.4.1 The Creation Subroutine

When at compile time a new *property name* is found, a new entry is created in the Property List, and the corresponding *Hidden Classes table* for this name is also created, which does not contain any entry at this point in time. Note that the *property identifiers* are natural numbers assigned sequentially as new names are encountered.

As we saw in Section 3.2.1, the V8 engine creates Hidden Classes dynamically as new properties are encountered at runtime. Therefore, the Property List is updated whenever a new Hidden Class is created. For this purpose, the name of the property is searched in the Property List, and a new entry is added to the corresponding *Hidden Classes table*.

8.3.4.2 The Miss Subroutine

When a Property or Prototype cache miss occurs, a software trap is generated and the runtime executes a subroutine that searches the required information from the Property List. For this purpose, the Property List is indexed by the *property identifier*, in order to obtain the pointer to the corresponding *Hidden Classes Table*. In this table, the subroutine searches the entry that matches the *Hidden Class identifier*. If the entry is found, then its *offset* field is transferred to the Property Cache and the exception routine finishes. If the entry is not found, it probably means that the property is contained in the prototype chain of the object. In this case, we obtain

the prototype of the object, and the table is searched again for this *Hidden Class identifier* of the prototype. This is an iterative process that is repeated successively with all the prototype chain, until the *Hidden Class identifier* matches an entry or the end of the prototype chain is reached. When the former occurs, the *offset* field is transferred to the Property Cache and the *P* field for that *offset* is set to 1 (see figure 8.2). Moreover, the address of the prototype whose *Hidden Class identifier* has matched the entry is transferred to the Prototype Cache, since in this case the accessed property is contained in this prototype instead of the object itself. If the *Hidden Class identifier* has not matched any entry and the end of the prototype chain is reached, the *NoExist* signal of Figure 8.5 is set to 1 and the special *undefined* value is returned, as explained in section 8.3.3

8.3.5 Other Issues

In this section we describe some special cases and optimizations.

8.3.5.1 Two Special Instructions

As we have explained in section 3.2.2, Crankshaft generates specialized code that contains checking operations and an important amount of these operations are inserted just before every property access. However, there are some situations where the compiler removes the execution of some *Check Maps* operations that guard loads of properties when they target objects that have already been checked in the same basic block.

In figure 8.6a, we observe a scenario where for every property load of the same object *obj*, a *Check Maps* operation is initially inserted. If no stores for this object are performed in between, all these *Check Maps* operations can be removed, except for the first one. The optimized version is showed in Figure 8.6b.

The reason for having two new instructions (as described in section 8.3.2.4) instead of just one is to optimize the scenario described above in Figure 8.6b. For the first load both *specialMovMap* and *specialMovOffset* instructions are needed, whereas for the other loads, just the *specialMovOffset* is sufficient since the *specialMovMap* instruction is redundant (all would return the same *Hidden Class identifier*).

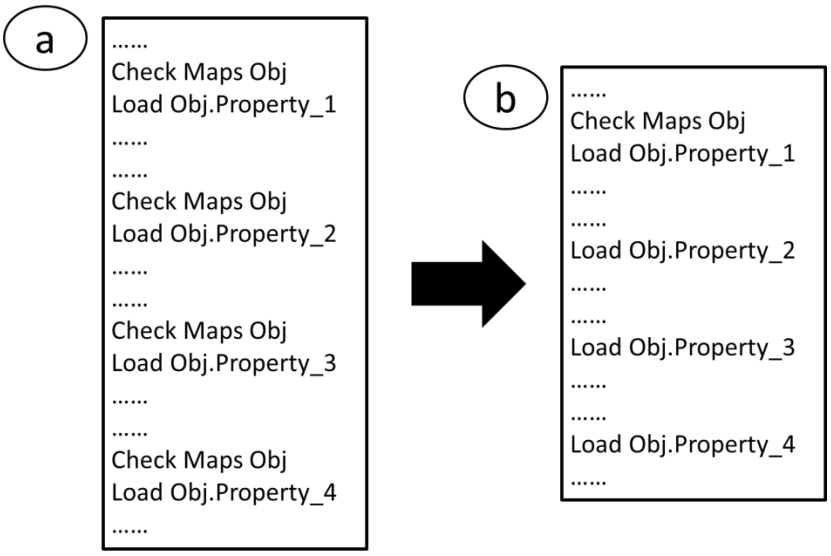

Figure 8.6: Specialized code with *Check Maps* operations.

8.3.5.2 Prototype Cache Optimizations

We have observed that most of the accessed prototypes are located in the first level of the prototype chain (i.e. the first prototype after the object). When all objects of a particular Hidden Class have accessed only up to the first prototype level, we refer to that Hidden Class as *single-prototype*. The information about which Hidden Classes are *single-prototype* is kept in a new field of the Hidden Class descriptors. When a new Hidden Class is created, this field is initialized to *single-prototype* and it is updated by the runtime when either a Property Cache miss, or a Prototype Cache miss occurs, if the accessed property is located in a prototype beyond the first level of the prototype chain.

Therefore, when the Prototype Cache is accessed using a *single-prototype* Hidden Class, then the tag comparison regarding the *property identifier* is not required. This is because we are sure that only one prototype (i.e. the first one) contains the requested property, no matter which property we access. We can exploit this fact to reduce the size of the Prototype Cache since all properties of a *single-prototype* Hidden Class can share the same entry. The hardware modifications to implement this optimization are shown in Figure 8.7. Note that in this case, the *property identifier* is set to a special value (all bits are set to 1) which does not belong to any property. If the *Hidden Class identifier* matches the searched one, there will be a hit no matter which property is being searched.

We have experimentally observed that this optimization is very effective since about 90% of the property loads are performed to *single-prototype* Hidden Classes.

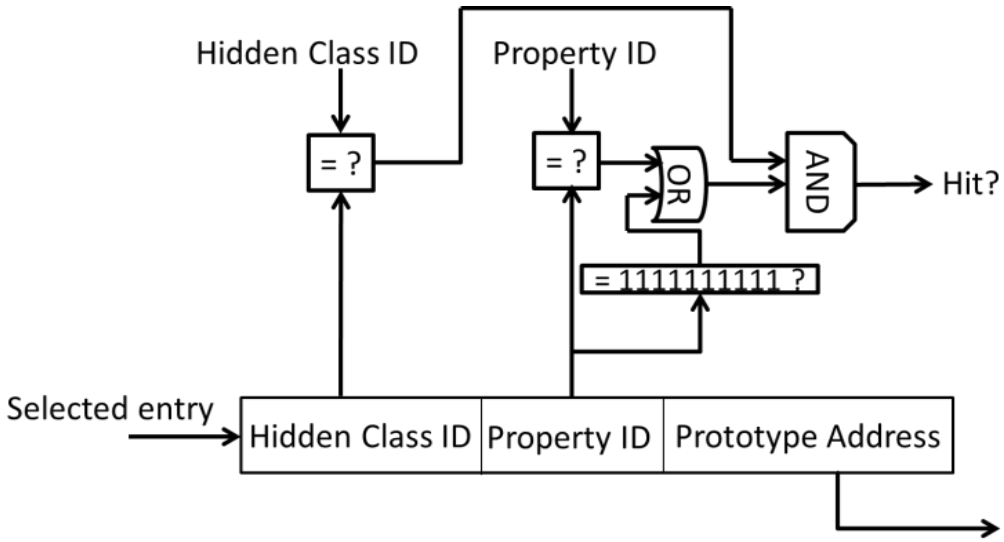

Figure 8.7: Block diagram of the Property Cache optimized for *single-prototype* Hidden Classes.

8.3.5.3 Offset Invalidations

There are two scenarios that may require the *offset* invalidation of a Prototype Cache entry: The addition of a new property in a prototype object and the overwriting of the *prototype* property in a prototype object. It is important to recall that a prototype is any object that is located within at least one prototype chain. Next, we describe in detail these two scenarios.

In Figure 8.8a, we show the prototype chain of a particular object *O* and the current state of the Prototype Cache. This prototype chain contains two prototype objects: *prototype 1* and *prototype 2*. Each prototype has two different properties with a different identifier (*property identifiers 3* and *4* in *prototype 1*, and *5* and *6* in *prototype 2*). In this scenario, the Prototype Cache has an entry for each one of these four properties, since they have been previously accessed by object *O*.

In Figure 8.8b, we show the prototype chain of figure 8.8a after adding a new property (*property identifier* 6) in *prototype 1*. Note that this property has the same identifier as the second property of *prototype 2*, which means that both have the same name (but not necessarily the same type). In this scenario, the Prototype Cache state of Figure 8.8a is incorrect, because according to the JavaScript inheritance mechanism, we have to obtain the closest property of the prototype chain. Therefore, the property with ID 6 has to come from *prototype 1* and the corresponding entry in the Prototype Cache is incorrect.

In Figure 8.8c, we show the prototype chain of figure 8.8a after overwriting the *prototype* property of *prototype 1* (its corresponding prototype is changed to another object:

object *prototype 3*). In this scenario, the Prototype Cache state of Figure 8.8a is also incorrect. This is because the prototype chain has changed, which now is composed by *prototype 1* and *prototype 3*. Therefore, all the entries that their property comes from *prototype 2* are incorrect (see in Figure 8.8c).

In summary, every time that the above scenarios occur, all entries in the Prototype Cache whose Hidden Class contains the modified object in its prototype chain should be invalidated. Since identifying all these entries can be costly, and we have observed that this scenario is relative rare, we have adopted a conservative simple solution consisting in invalidating all the Prototype Cache entries that do not contain a *single-prototype* Hidden Class.

We do not need to invalidate *single-prototype* Hidden Class entries because on the one hand, according to the problem described in Figure 8.8b, we are sure that the properties represented by these entries are the closest ones of the prototype chain. Therefore, other new properties with the same name in subsequent levels of the prototype chain will not be owned by the object. On the other hand, according to the problem described in Figure 8.8c, for *single-prototype* Hidden Class entries we only care about the first prototype of the chain (i.e. which is contained in the Hidden Class of the object). If this prototype is overwritten, then the class of the object also changes (see section 3.1) and therefore, next time that the object is accessed, it will miss in the Property and Prototype Caches.

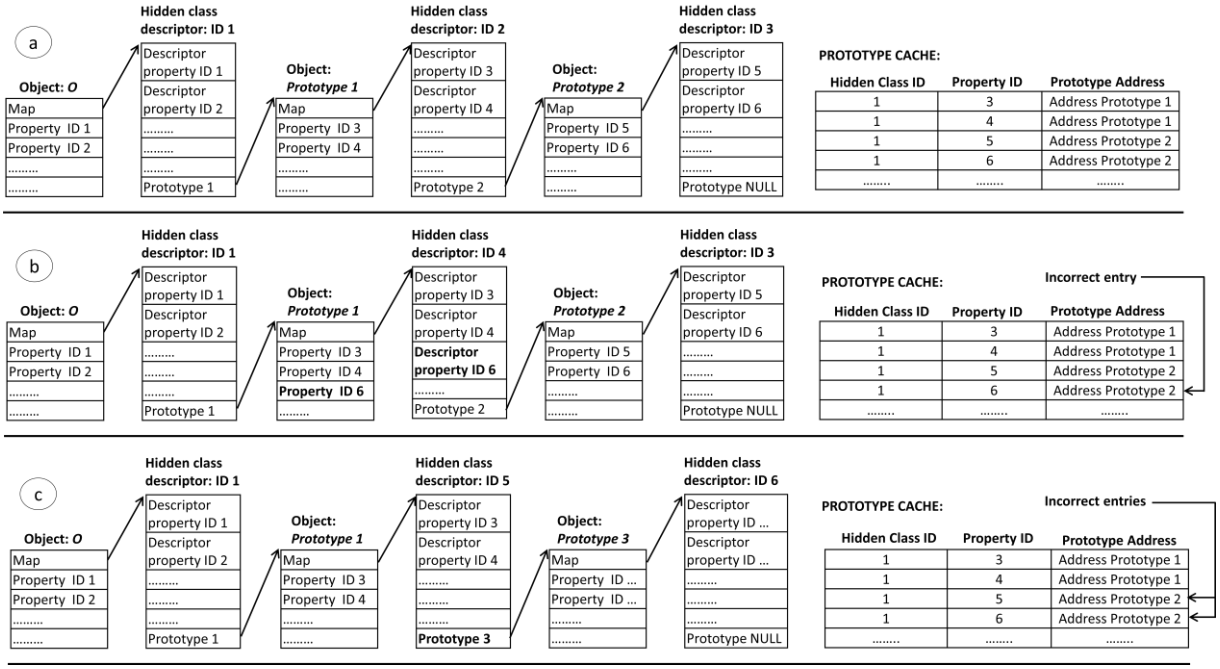

Figure 8.8: Prototype Cache invalidations.

8.3.6 An Example of the Proposed Optimizations

In Figure 8.9, we show an example of our proposed optimizations for the line 22 of the *departmentStaff* function explained in section 5.3.1, taking into account the Class List example showed in Figure 8.2. The left part of the Figure shows the original assembly x86-64 code, which performs two object property loads (highlighted in bold). For the first property load (instructions I2-I4), a *call* instruction is executed, which targets the corresponding Inline Cache for this access, in order to obtain the value of a property called *people* (*property identifier 5*) from the object contained in the *rax* register (then moved to *rdx* as an input parameter for the *call*). For the second property load (instructions I10-I2), the same process is repeated for a property called *salary* (*property identifier 3*).

The right part of the Figure contains the x86-64 assembly code after applying our optimizations. The first property load described above is performed by instructions I2 and I3. The former instruction is a *specialMovMap* instruction, which loads the Hidden Class of the object to the special *hiddenClassReg* register. The latter is a *specialMovOffset* instruction, which accesses the Property Cache, in order to obtain the corresponding *offset* for the property called *people* (*property identifier 5*) from the Hidden Class stored in *hiddenClassReg*. Then, a memory access is performed using the obtained *offset* and the resulting value is stored to the *rax* register. The second property load is performed by instructions I9 and I10, which work very similar than instructions I2 and I3, but with *salary* as a property name (*property identifier 3*).

Note that with our optimizations we are avoiding the execution of *call* instructions to miss handler subroutines or Inline Caches, as long as no Property nor Prototype Cache misses are produced.

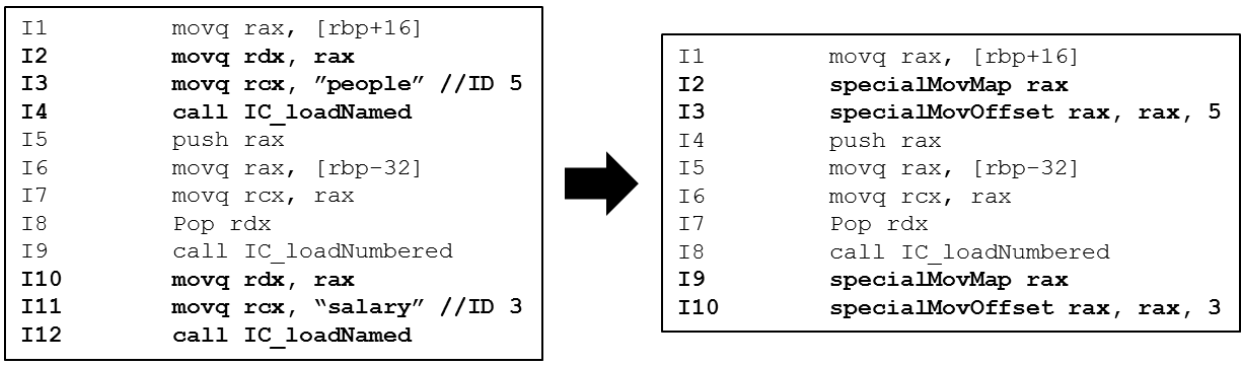

Figure 8.9: An example of the proposed optimizations.

8.4 Performance evaluation

In this section we evaluate the benefits of the proposed technique in terms of performance and energy consumption. We have implemented the software changes required by the above mechanism in V8. To model the hardware, we have used the Marss [2] cycle-level microarchitecture simulator. For energy consumption we have used McPat [53] and CACTI [55] power models. We have used Octane [26][27] (*navier-stokes* and *code-load* are not included since they crash in our simulation environment) and JSBench [29][36] benchmark suites for the evaluation of the proposed mechanism. We have discarded SunSpider and Kraken benchmark suites for the evaluations of this technique because they are not representative of typical short-running web applications, as explained in chapter 5, and therefore they hardly benefit from this mechanism.

We have chosen a 256-entry, 4-way set associative configuration for the Property Cache and a 64-entry, direct-mapped configuration for the Prototype Cache. Next section provides a sensitivity analysis to varying these parameters.

The additional hardware incurs in a 2-cycle penalty for each load of an object property. This overhead is mainly due to both the addition operation used to obtain the effective address and the access to the Property and Prototype Caches, which are very small structures (1.25 KB and 0.5 KB respectively). However, when the accessed property is directly obtained from the *hiddenClassReg* register (see section 8.3.3), the additional hardware incurs only a 1-cycle penalty because the addition operation of the step *5* from Figure 8.5 is not necessary for these cases.

As evaluation methodology, we have measured the total number of cycles and dynamic instructions of our technique by adding the results of two separate simulations. In the first simulation, we have executed our technique with a perfect Property and Prototype Caches (i.e., without cache misses). In this simulation, we have also measured the penalty (i.e., the time spent by executing specialMovMap instructions and the extra 1-cycle or 2-cycle latency for specialMovOffset instructions) incurred by accessing the new hardware structures in a hit scenario. In the second simulation, we have obtained the number of cycles and dynamic instructions for the Property and Prototype Cache misses and the updates to the Property List, which is modified (i.e., extended) for each new hidden class creation.

8.4.1 Execution Time

Our technique achieves an important improvement in the execution time of JavaScript applications, with an average speedup of 11% as shown in Figure 8.10. A remarkable case is *raytrace* benchmark, with a 33% speedup. This benchmark performs an important number of property loads during the non-optimized code, which are optimized by our technique. *Gbemu* and *typescript* benchmarks also obtain an important benefit with our technique. These two benchmarks use a large number of different properties and Hidden Classes during execution, which increases the degree of polymorphism of the Inline Caches. The larger the degree of polymorphism, the slower the Inline Cache is. On the other hand, *mandrel* and *zlib* benchmarks present a very poor improvement since they perform very few property loads. In addition, all applications from JSBench suite achieve important improvements, which confirm the effectiveness of the Property Cache Mechanism for short-running web applications.

Figure 8.11 shows the reduction of the overhead produced by loads of object properties (original overhead is shown in Figure 8.1). We can see that our technique reduces drastically this overhead, by 90% on average, and the reduction is quite high for all programs, which proves that our technique addresses a rather common source of overhead in JavaScript applications.

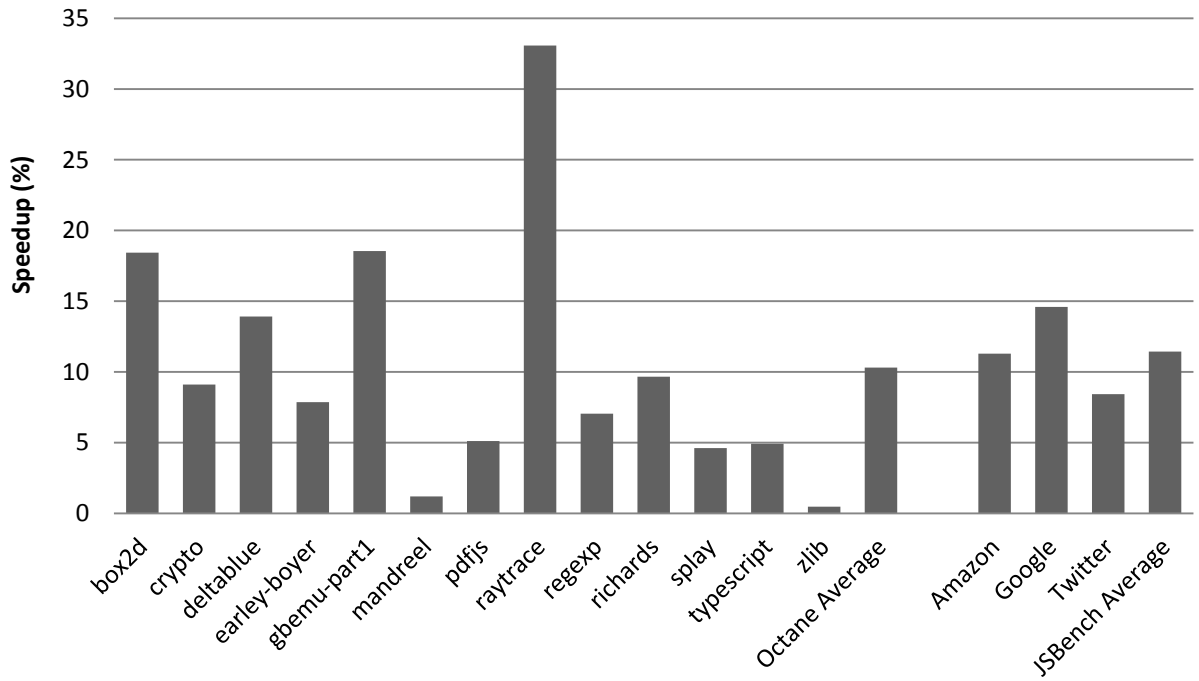

Figure 8.10: Improvement in execution time.

8.4.2 Sensitivity Analysis

The main motivation of this sensitivity analysis is to identify a good tradeoff between cost and benefits of the proposed mechanism. We have evaluated the Property Cache mechanism with different number of entries and associativity for the Property Cache. Table 8.1 shows the resulting overhead due to misses for the different configurations.

To identify the capacity requirements, let us first look at the Full-Associative row. We can observe that a cache with 128 entries still suffers from these misses, whereas using 256 or more entries practically removes all miss penalties. Regarding conflict misses, we discarded a direct-mapped configuration because its high miss rate. Both, 2-way and 4-way configurations seem reasonable, so we finally chose a 256-entry, 4-way set associative cache as the best trade-off between cost and benefit. Figure 8.12 shows the Property Cache hit rate using this configuration.

The total size of the Property Cache is 1.25 KB since each entry occupies 5 bytes: 20 bits for the *Hidden Class identifier*, 10 bits for the *property identifier*, 9 bits for the *offset*, and

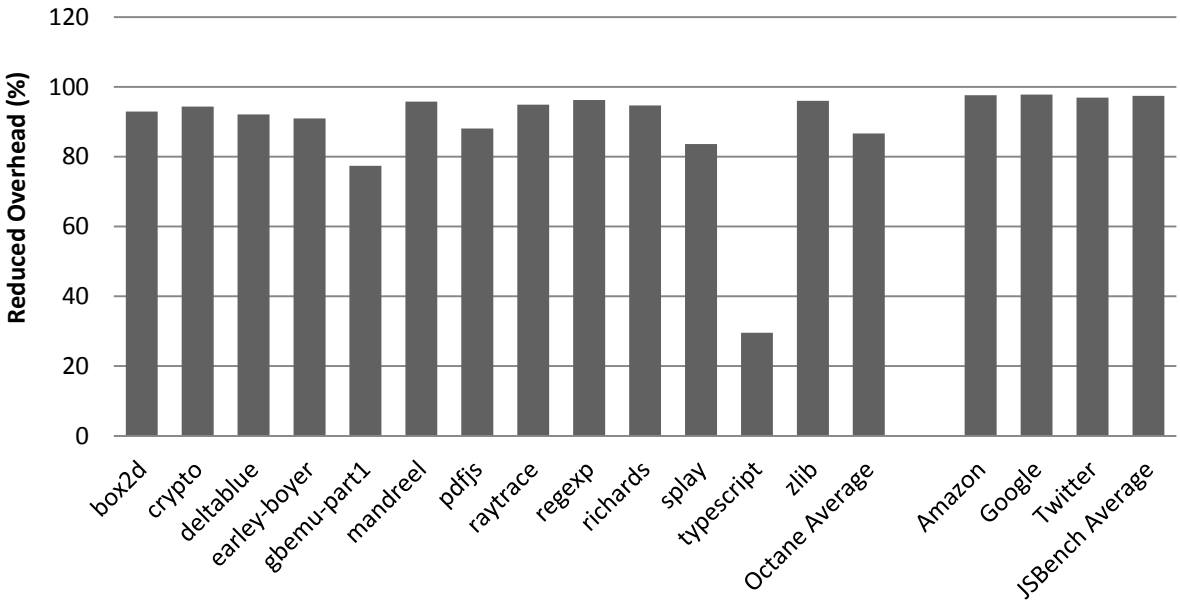

Figure 8.11: Overhead reduction in number of cycles.

	128 entries	256 entries	512 entries
Direct-mapped	7,28%	3,48%	1,29%
2-way	2,71%	1,22%	0,61%
4-way	2,15%	0,82%	0,45%
Full-associative	1,51%	0,23%	0,01%

Table 8.1: Overhead produced by Property Cache Misses.

1 bit for the *P* field. In the extremely rare case that an application requires more bits for any of these fields, the proposed mechanism would be simply not used for this particular application (this never happened in our benchmarks). Note that the *Hidden Class identifier* is an address that points to the Hidden Class descriptor (i.e. it occupies 64 bits), but we have observed that only the 20 least-significant bits or less change for typical applications, since these structures are put together in consecutive memory locations.

The Prototype Cache has 64 entries and is direct-mapped. We have chosen a simple and small configuration for the Prototype Cache, as the majority of object property accesses target the object itself, instead of the prototype chain.

The total size of the Prototype Cache is 0.5 KB since each entry occupies 8 bytes: 20 bits for the *Hidden Class identifier*, 10 bits for the *property identifier*, 9 bits for the prototype address, and 1 bit for the invalid field. Note that for the prototype address needs we only keep the 32 least-significant bits. This is because V8 only reserves 4 GB of virtual memory for the heap and therefore, the remaining bits are the same for all object addresses.

8.4.3 Energy Consumption

Figure 8.12 shows the energy savings of our technique for Octane and JSBench benchmark suites, which are measured through the McPAT simulator [53] and CACTI [55] (i.e. CACTI has been used to obtain the energy consumption of the Property and Prototype Caches). Energy

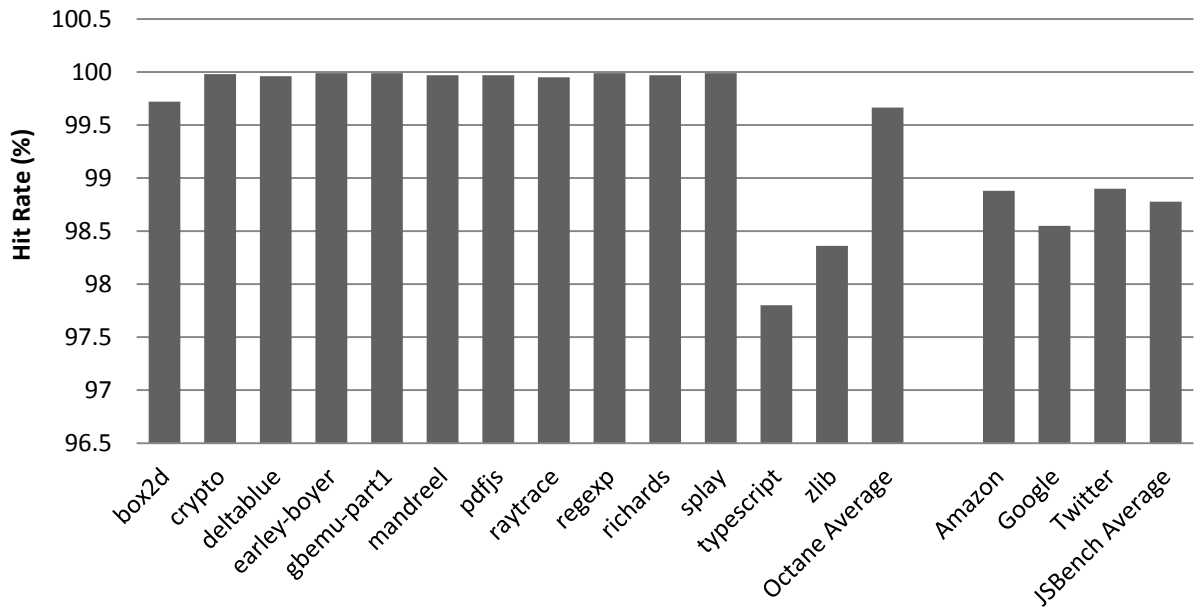

Figure 8.12: Hit rate of the Property Cache for 256 entries and 4-way associativity.

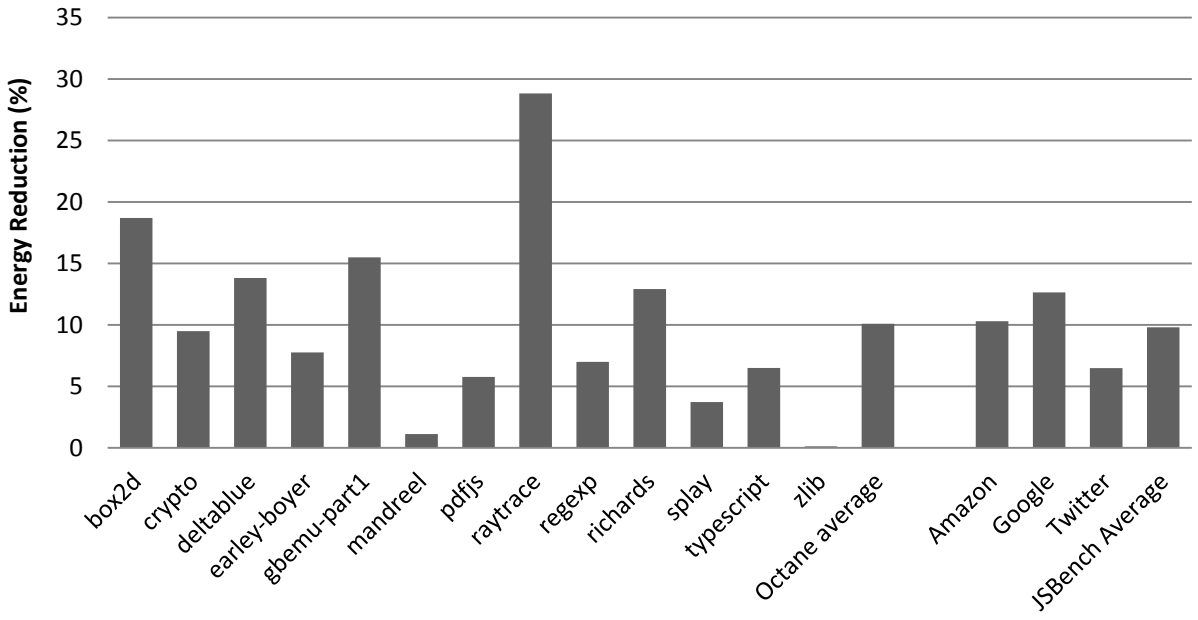

Figure 8.13: Improvement in energy consumption.

consumption is reduced by 9.9% on average and is close to 30% in some applications (e.g. *raytrace*). These important 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).

8.5 Conclusions

In this chapter, we have proposed a HW/SW mechanism that removes most of the overhead due to the Inline Caching mechanism in short-running applications. This mechanism requires small hardware extensions, mainly two new specialized memories with a total capacity less than 2KB and two new machine instructions.

We have shown that the proposed mechanism produces important benefits both in execution time and energy consumption. This technique opens a new avenue in the way to deal with code specialization in dynamically typed languages. In future work we plan to investigate how to apply a similar approach to remove other overheads related to other scenarios of the Inline Caching mechanism, such as the array accesses.

Chapter 9

Summary and Future Work

In this chapter, the main thesis conclusions are summarized and some future work is presented.

9.1 Summary

Dynamically typed languages are ubiquitous in today applications. These languages ease the task of programmers but introduce significant runtime overheads. Since variables are neither declared nor bound to a particular type, for efficiency reasons, the code generated at runtime is specialized for certain types and assumptions about the types of variables require to be constantly validated. These validations are an important source of overheads.

Analysis of Overheads. In chapter 5, we have evaluated the overheads for different kind of JavaScript applications, including short-running, event-based applications and long-running, compute-intensive applications. In the former, the overhead mainly occurs during the execution of non-optimized code, which performs an important amount of profiling work and other tasks related to the lookup of object properties. In the latter, the main overheads arise while executing specialized code, and are mainly due to the frequent execution of checking operations that are used to preserve some type assumptions.

Fusion of Common Instruction Patterns. In chapter 6, we propose three instruction-level optimizations, in order to improve the performance of checking operations executed in the optimized code. These optimizations are based on a hybrid HW/SW approach that requires the introduction of some new machine instructions, which improve the performance of the most common instruction patterns related to this overhead. These optimizations require also some changes in the code generated by the dynamic compiler.

The Class Cache. In chapter 7, we demonstrate that in long-running applications, an important amount of checking operations target monomorphic properties or monomorphic *elements arrays*. In this regard, we have proposed a new hybrid HW/SW scheme based on a runtime

profiling that keeps information about these *monomorphic* properties and *elements arrays*, in order to remove the checking operations that target them, in a safely manner. Besides, the proposed scheme detects when this specialized code is no longer correct before executing it, so there is no need for providing a recovery mechanism. In these cases, an exception is triggered and the code is recompiled to a non-specialized version that is guaranteed to be correct.

The Property Cache. In chapter 8, we propose a hybrid HW/SW mechanism that removes most of the overhead in short-running applications. The proposed technique avoids the speculative strategy adopted by state-of-the-art dynamic compilers for property lookup operations. Instead of speculation, our approach relies on a runtime-built structure that provides the information required to identify the addresses of object properties in a very efficient manner. Besides, a hardware cache of this structure stores the most frequently elements to speedup its access. This technique is applied to both optimized and non-optimized code.

9.2 Future Work

The work presented in this thesis can open different research lines, according to the kind of applications.

Long-running applications. The Class Cache mechanism presented in chapter 7 is focused on checking operations for objects properties or object *elements arrays*. However, there are checking operations that target other program variables, such as function parameters or global variables. We can extend the Class Cache to profile the types of these other program variables by providing them a pseudo-*ClassID*, which would be also contained in the first 8-byte word of each cache line that contains any of these variables.

Short-running applications. The Prototype Cache mechanism presented in chapter 8 opens a new avenue in the way to deal with code specialization in dynamically typed languages. In future work we plan to investigate how to apply a similar approach to remove the overheads related to other scenarios of the Inline Caching mechanism, such as the stores to object properties. Although object property stores can be optimized in a similar way as object property loads, they are different because in some cases Hidden Class transitions occur. As explained in chapter 5, 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*. To deal with this situation, a new cache called the Transition Cache could be added to our mechanism, in order to keep the target Hidden Classes of these transitions. Therefore, when a property store that produces a transition is executed, both the Property Cache and the Transition cache would be accessed, and the Hidden Class of the object would be updated with the corresponding Hidden Class of the Transition Cache.

References

[1] 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.

[2] 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.

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

[4] A. 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.

[5] A. Wingo. inside full-codegen, v8's baseline compiler. http://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler, 2011.

[6] Brian Hackett , Shu-yu Guo. Fast and precise hybrid type inference for JavaScript, Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, June 11-16, 2012, Beijing, China.

[7] Carl Friedrich Bolz , Antonio Cuni , Maciej Fijalkowski , Armin Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, p.18-25, July 06-06, 2009, Genova, Italy.

[8] C. Chambers , D. Ungar, Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language, Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, p.146-160, June 19-23, 1989, Portland, Oregon, USA.

[9] 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.

[10] Christopher Anderson , Paola Giannini , Sophia Drossopoulou, Towards type inference for javascript, Proceedings of the 19th European conference on Object-Oriented Programming, July 25-29, 2005, Glasgow, UK.

[11] Christopher Anderson , Paola Giannini, Type Checking for JavaScript, Electronic Notes in Theoretical Computer Science (ENTCS), v.138 n.2, p.37-58, November, 2005.

[12] Craig Chambers , David Ungar , Elgin Lee, An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes, Lisp and Symbolic Computation, v.4 n.3, p.243-281, July 1991.

[13] D. Bonetta, W. Binder, C. Pautasso: TigerQuoll: parallel event-based JavaScript. In: Proc. of PPoPP, pp. 251–260 (2013).

[14] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA), pages 307--318, Feb. 2010.

[15] 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.

[16] Drevor, Tevi, Pin : Intel's Dynamic Binary Instrumentation Engine, Pin Tutorial. Intel Corporation, 2010.

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

[18] ECMA. ECMAScript Language Specification – Fifth Edition. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf.

[19] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In Proceedings of IISWC, pages 1–10, 2010.

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

[21] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In OOPSLA'04 Workshop on Revival of Dynamic Languages, 2004.

[22] 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.

[23] G. Dot, A. Martínez, A. González, "Analysis and optimization of JavaScript engines", 1st Workshop on High Performance Scripting Languages, in conjunction with 20th ACM SIGPLAN Symposium on principles and Practice of Parallel Programming (PPOPP), San Francisco (USA), February 2015.

[24] G. Dot, A. Martínez, A. González, "Removing Checks in Dynamically Typed Languages through Efficient Profiling". Submitted to Dynamic Languages Symposium (DLS), 2016.

[25] G. Dot, A. Martínez, A. González, "ERICO: Effective Removal of Inline Caching Overhead in Dynamic Programming Languages". Submitted to 23rd IEEE Int. Conference on High Performance Computing, Data and Analytics (HiPC), 2016.

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

[27] Google Octane benchmark suite. http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html.

[28] Google V8 JavaScript Engine - http://code.google.com/p/v8

[29] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In OOSPLA, 2011.

[30] Herhut, S., Hudson, R.L., Shpeisman, T., Sreeram, J.: River trail: a path to parallelism in JavaScript. In: OOPSLA, pp. 729–744 (2013).

[31] I. Jibaja, P. Jensen, J. McCutchan, D. Gohman, N. Hu, M. Haghighat, S. Blackburn, and K. McKinley "Vector Parallelism in JavaScript: Language and compiler support for SIMD". In Proc of PACT 2015.

[32] 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.

[33] J. K. Martinsen, H. Grahn and A. Isberg. "Using speculation to enhance JavaScript performance in web applications", *IEEE Internet Computing* , vol. 17 , no. 2 , pp.10 -19, 2013.

[34] J.K. Martinsen and H. Grahn, "An alternative optimization technique for JavaScript engines", appeared in proceeding of the Third Swedish Workshop on Multi-Core Computing (MCC-10), pages 155-160, November 2010, Göteborg, Sweden.

[35] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. IEEE Computer, March 1998.

[36] JsBench. http://plg.uwaterloo.ca/~dynjs/jsbench/.

[37] Justin O. Graver and Ralph E. Johnson. A type system for smalltalk. In Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 136–150. ACM Press, 1990.

[38] J Verdú, A Pajuelo. Performance Scalability Analysis of JavaScript Applications with Web Workers. In Computer Architecture Letters, Volume:PP , Issue: 99. October of 2015.

[39] Karel Driesen, Selector table indexing & sparse arrays, ACM SIGPLAN Notices, v.28 n.10, p.259-270, Oct. 1, 1993.

[40] 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.

[41] Madhukar N. Kedlaya , Jared Roesch , Behnam Robatmili , Mehrdad Reshadi , Ben Hardekopf. Improved type specialization for dynamic scripting languages, Proceedings of the 9th symposium on Dynamic languages, October 28-28, 2013, Indianapolis, Indiana, USA.

[42] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07--10, University of California, Irvine, 2007.

[43] Mike Salib. Static type inference (for python) with starkiller. http://www.python.org/pycon/dc2004/papers/1/paper.pdf, 2004.

[44] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke, "Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism," in Proc.of HPCA, 2011.

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

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

[47] 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.

[48] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. Lecture Notes in Computer Science, 707:247–262, 1993.

[49] Peter Steenkiste , John Hennessy, Tags and type checking in LISP: hardware and software approaches, Proceedings of the second international conference on Architectual support for programming languages and operating systems, p.50-59, October 1987, Palo Alto, California, USA.

[50] Peter Thiemann, Towards a type system for analyzing javascript programs, Proceedings of the 14th European conference on Programming Languages and Systems, p.408-422, April 04-08, 2005, Edinburgh, UK.

[51] Shisheng Li , Buqi Cheng , Xiao-Feng Li, TypeCastor: demystify dynamic typing of JavaScript applications, Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, January 24-26, 2011, Heraklion, Greece.

[52] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In Static Analysis Symposium (SAS), 2009.

[53] 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.

[54] Stefan Brunthaler, Inline caching meets quickening, Proceedings of the 24th European conference on Object-oriented programming, June 21-25, 2010, Maribor, Slovenia.

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

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

[57] Tratt L. (2009). Dynamically typed languages. Adv. Comput. 77, 149–18410.1016/S0065-2458(09)01205-4.

[58] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In International Conference for High Performance Computing, Networking, Storage and Analysis (SC), November 2011.

[59] 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.

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

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

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

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

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

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

[66] Yeoul Na, Seon Wook Kim, Youngsun Han. "JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications". ACM Transactions on Architecture and Code Optimization (TACO), Volume 12 Issue 4, Article No.64, January 2016.

Appendix A: New x86-64 Instructions of chapter 6.

name	Mnemonic	Description
xehcmp	xehcmp reg32, imm32, flag, neg	HW exception comparison of reg32 with imm32
	xehcmp reg32, reg32, flag, neg	HW exception comparison of reg32 with reg32
	xehcmp reg64, reg64, flag, neg	HW exception comparison of reg64 with reg64
	xehcmp reg64, mem64, flag, neg	HW exception comparison of reg64 with mem64
	xehcmp mem64, reg, flag, neg	HW exception comparison of mem64 with reg64
xehtest	xehtest reg64, reg64, flag, neg	HW exception test of reg64 with reg64
	xehtest reg32, reg32, flag, neg	HW exception test of reg32 with reg32
	xehtest reg32, imm32, flag, neg	HW exception test of reg32 with imm32
	xehtest mem8, imm8, flag, neg	HW exception test of mem8 with imm8
	xehtest reg8, imm8, flag, neg	HW exception test of reg8 with imm8
xehadd	xehadd reg32, reg32, flag, neg	HW exception addition of reg32 with reg32
	xehadd reg32, imm32, flag, neg	HW exception addition of reg32 with imm32
	xehadd reg32, mem32, flag, neg	HW exception addition of reg32 with mem32
	xehadd reg64, reg64, flag, neg	HW exception addition of reg64 with reg64
	xehadd reg64, mem64, flag, neg	HW exception addition of reg64 with mem64
xehsub	xehsub reg32, reg32, flag, neg	HW exception subtraction of reg32 less reg32
	xehsub reg32, imm32, flag, neg	HW exception subtraction of reg32 less imm32
	xehsub reg32, mem32, flag, neg	HW exception subtraction of reg32 less mem32
	xehsub reg64, reg64, flag, neg	HW exception subtraction of reg64 less reg64
	xehsub reg64, mem64, flag, neg	HW exception subtraction of reg64 less mem64

name	Mnemonic	Description
xehimull	xehimul reg32, reg32, flag, neg	HW exception integer multiplication of reg32 with reg32
	xehimul reg32, imm32, flag, neg	HW exception integer multiplication of reg32 with imm32
	xehimul reg32, mem32, flag, neg	HW exception integer multiplication of reg32 with mem32
	xehimul reg64, reg64, flag, neg	HW exception integer multiplication of reg64 with reg64
	xehadd reg64, mem64, flag, neg	HW exception integer multiplication of reg64 with mem32
xehor	xehor reg32, reg32, flag, neg	HW exception binary or of reg32 with reg32
	xehor reg32, mem32, flag, neg	HW exception binary or of reg32 with mem32
	xehor reg64, reg64, flag, neg	HW exception binary or of reg64 with reg64
	xehsub reg64, mem64, flag, neg	HW exception binary or of reg64 with mem64
xehand	xehand reg32, imm32, flag, neg	HW exception binary and of reg32 with imm32
xehneg	xehneg reg32, flag, neg	HW exception binary neg of reg32
	xehneg reg64, flag, neg	HW exception binary neg of reg64
xehucomis	xehucomis reg128, reg128 flag, neg	HW exception ucomis operation of reg128 with reg128
xehtestshr	xehtestshr reg8	HW exception test + shift rigth operations with reg8
	xehtestshr reg32	HW exception test + shift rigth operations with reg32
	xehtestshr reg64	HW exception test + shift rigth operations with reg64
xehtestcmp	xehtestcmp reg64, reg64	HW exception test + comparison of reg64 and reg64
	xehtestcmp rax, reg64	HW exception test + comparison of rax and reg64

Appendix B: New x86-64 Instructions of chapter 7

name	**Mnemonic**	**Description**
movStoreClassCache	movStoreClassCache mem64, reg64	Class Cache Request for property access scenario plus *mov* instruction of reg64 to mem64
movStoreClassCacheArray	movStoreClassCacheArray mem64, reg64, regArray	Class Cache Request for *elements array* access scenario plus *mov* instruction of reg64 to mem64
movClassID	movClassID mem64	Special *mov* instruction of ClassID field from an object to the regObjectClassId register
movClassIDArray	movClassIDArray regArray, mem64	Special *mov* instruction of ClassID field from an object to a regArrayObjectClassId0-3 register

Appendix C: New x86-64 Instructions of chapter 8

name	**Mnemonic**	**Description**
specialMovMap	specialMovMap mem64	Special *mov* instruction to load the *Hidden Class identifier* of an object (along with the whole cache line) to the special *hiddenClassReg* register.
specialMovOffset	specialMovOffset reg64, mem64, propertyID	Property Cache and Prototype Cache request, which is indexed by both the *propertyID* operand and the *Hidden Class identifier* stored in the *hiddenClassReg* register. Then a memory request is performed with the obtained offset. At the end of the instruction, the obtained value from memory is stored to reg64 destination register.