Checked Load: Architectural Support for JavaScript Type-Checking on Mobile Processors

Owen Anderson Emily Fortuna Luis Ceze Susan Eggers
Computer Science and Engineering, University of Washington
http://sampa.cs.washington.edu

Abstract

Dynamic languages such as Javascript are the de-facto standard for web applications. However, generating efficient code for dynamically-typed languages is a challenge, because it requires frequent dynamic type checks. Our analysis has shown that some programs spend upwards of 20% of dynamic instructions doing type checks, and 12.9% on average.

In this paper we propose Checked Load, a low-complexity architectural extension that replaces software-based, dynamic type checking. Checked Load is comprised of four new ISA instructions that provide flexible and automatic type checks for memory operations, and whose implementation requires minimal hardware changes. We also propose hardware support for dynamic type prediction to reduce the cost of failed type checks. We show how to use Checked Load in the Nitro JavaScript just-in-time compiler (used in the Safari 5 browser). Speedups on a typical mobile processor range up to 44.6% (with a mean of 11.2%) in popular JavaScript benchmarks. While we have focused our work on JavaScript, Checked Load is sufficiently general to support other dynamically-typed languages, such as Python or Ruby.

1 Introduction

Dynamically-typed languages, including JavaScript [9], Python [18], and Ruby [7], have exploded in popularity, thanks largely to their lower barriers to entry for both development and deployment, afforded, in part, by their high-level abstractions. These language features allow for more expressive programs, while simultaneously making universal portability significantly more tenable.

JavaScript, in particular, has seen massive growth, allowing rich client-side interactions in web applications, enabled by the availability of high-performance JavaScript virtual machines in all major web browsers. Popular web applications, including Google Maps, Twitter, and Facebook would not be feasible without both high-throughput and low-latency JavaScript virtual machines on the client.

At the same time, innovations in mobile device programmability have opened up embedded targets to the same class of programmers. Today's smart mobile devices are expected to provide a developer API that is usable by normal application developers, as opposed to the specialized embedded developers of the past. One such platform, HP/Palm's WebOS [17], uses JavaScript as its primary application development language. Others encourage JavaScript-heavy web applications in addition to their native development environments, as a means of providing feature-rich, portable applications with minimal development costs.

Because of their power and space constraints, embedded processors for mobile devices typically do not employ traditional heavyweight architectural techniques, such as wide-issue, out-of-order execution, to hide instruction latencies. Instead, lightweight, minimal techniques must be used. For lightweight architectures which were originally designed for executing languages such as C, dynamically-typed languages pose special performance problems. While these architectures have successfully weathered the transition from their procedural-language roots to modern object-oriented languages like C++ and Java, progressing to dynamically-typed languages is considerably more challenging to implement efficiently. In contrast to object-oriented languages, where the majority of the language closely resembles a procedural language (with the addition of dynamic method dispatch), dynamically-typed languages are built on higher-level primitives.

In JavaScript this translates into an instruction stream dominated by dynamic type checks for "primitive" types and hash-table accesses for user-defined types. In addition, because of a combination of a language that is not conducive to static analysis and the demands responsiveness places on code-generation latency, one cannot rely on the JavaScript virtual machine's just-in-time compiler to per-

form the aggressive optimizations that would reduce the costs of these operations. For example, the JavaScript VM Nitro (previously known as SquirrelFish Extreme [26]) that is used in Safari 5 performs only local optimization, such as allocating registers within a single macro-opcode, spilling and restoring them at opcode boundaries.

Contributions. The contributions of this paper are two-fold. First, we quantify the costs of type checks on mobile processors, in effect, identifying a performance gap to be closed. Second, we propose an instruction set extension to address this gap, design its implementation, and demonstrate its use in code generation and its impact on performance.

More specifically, we investigate the special costs of dynamic typing in JavaScript and propose and evaluate architecture-level instructions that reduce them. Our study shows that, while type checks are individually small, on the order of two x86 instructions per check, they occur with such frequency as to account for an average 10.9% (and a high of 46.8%) of executed instructions generated by JavaScript's virtual machine and 12.9% (62.0% at the maximum) of the execution time of the generated code.

To reduce the cost of these type checks, we propose a simple instruction set extension, Checked Load. Checked Load adds four instructions to an ISA and requires only a handful of basic hardware components the size of a comparator or MUX for its implementation. It brings performance benefits on generated code that average 11.2% and run as high as 44.6%, without slowing down microarchitectural critical paths. In conjunction with Checked Load, we also propose dynamic type prediction, which utilizes standard branch prediction hardware and is responsible for a large fraction of the performance improvements.

Outline. The remainder of this paper is organized as follows: Section 2 analyzes the costs of type checking on a modern mobile processor. Section 3 presents Checked Load, our proposed instruction set extension for efficiently implementing type checks. Section 4 demonstrates the performance benefits of Checked Load, including several type-prediction mechanisms. Section 5 examines related work in the scope of optimizing the execution of dynamic languages, from historical work on LISP to modern JavaScript implementations. Finally, Section 6 concludes.

2 Quantifying Type Check Costs

While it is accepted wisdom that dynamic type checking is a source of performance loss in dynamically typed languages, any investment in customized hardware must be driven by strong empirical evidence. For that reason, we first demonstrate the costs of dynamic typing on mobile processors by instrumenting the execution of a modern JavaScript VM, and running it on a simulator to reflect the performance characteristics of a mobile processor.

Type guards, as type checks are known in modern, dynamic language implementations, follow the prototype laid out by Steenkiste [23]. Under this approach, all values, regardless of type, are represented as machine-word-sized (virtual) registers, the high bits of which are used to implement a type tag. For primitive integers, the remaining bits are the value of the (reduced-width) integer; for other types they are a pointer into a memory allocation pool, thus providing the address of the object.

Given this structure, before a value can be accessed, a sequence of mask instructions first extracts the value of the tag. This is then followed by a branch on the tag value, either to a fast-path block (if the comparison succeeds), or to an error or recovery block (if it fails). In the fast case, the value is then extended and used in the type-appropriate computation. A clever tag encoding [23] is employed, so that two's-complement arithmetic operations may be issued on values known to be integers without the need to mask out the tag bits beforehand. In the slow case, fully general type-conversion routines (typically written in C and provided by the VM) are called to perform the necessary conversions before the actual computation can be performed.

Figure 1 shows the x86 assembly produced by the Nitro JavaScript VM for the fast-path block of an indexed array store operation, written as `a[i] = c;` in both JavaScript and C. In C this would typically be a single store instruction with a base-plus-offset addressing mode.

In contrast to the simplicity of that implementation, the x86 generated by the JavaScript VM contains five branches, three of which are directly attributable to type checking. To minimize the number of instructions generated, this particular implementation uses x86's rich branching constructs to avoid complex tag masking operations for the guards. Despite this, more than a third of the instructions are expended on type guards. On a RISC ISA, these guards may require more instructions, raising the type-guard-to-total-instruction ratio and leading to an even worse performance picture.

2.1 Experimental Methodology

JavaScript VM. To measure the cost of type checks on a real JavaScript implementation, we instrumented a copy of the Nitro JavaScript VM [26], that is used in Apple's Safari and MobileSafari browsers for desktop computers and the iPhone, respectively. Nitro is a method-oriented JIT compiler, with a macro-expansion code generator that is designed for low-latency code generation. Compilation occurs in two phases: the first phase lowers JavaScript source code

```
FastPath:
  // Load virtual registers
    mov    %rax → %rdx
    mov    0x10(%r13) → %rax

  // Guard for integer
    cmp    %r14, %rdx
    jb     SlowCase
    mov    %edx → %edx

  // Guard for JSCell
    test   %rax, %r15
    jne    SlowCase
    mov    $0x797d10 → %r11

  // Guard for JSArray
    cmp    %r11, (%rax)
    jne    SlowCase
    mov    0x38(%rax) → %rcx

  // Guard for array length
    cmp    0x30(%rax), %edx
    jae    SlowCase
    mov    0x20(%rcx,%rdx,8) → %rax

  // Guard null array pointer
    test   %rax, %rax
    je     SlowCase

  // Store to array
    mov    %rax → 0x18(%r13)
    ...

SlowCase:
        ...
```

Figure 1. Code generated by Nitro for the `op_put_by_val` (indexed array store) macro-operation. Note the five guards that it performs: that the index is an integer, that the array is non-primitive, that the array is an array, that the array length is not zero, and that the data pointer is not null.

to a linear bytecode representation composed of "macro-ops" that represent the operations of a program at a relatively high level; the second phase compiles each bytecode to machine code, performing no optimization across macro-op boundaries.

An alternative software technique for reducing the cost of dynamic type checks is trace-based compilation [10, 11], which is heavier-weight due to expensive trace collection and aggressive optimization, but generates higher-quality code. Nevertheless, due to various overheads and limitations, light-weight, non-trace-based compilation remains dominant in practice on both mobile and desktop JavaScript implementations. (We revisit the idea of trace-based compilation in Section 5.3.)

In the context of the analysis in this paper, Nitro's design is comparable to that of the V8 JavaScript VM [12] used in Google's Chrome browser and on the Android and WebOS mobile platforms. V8 is also a traditional, method-oriented compiler. While V8 and Nitro differ in important respects (primarily in their approaches to dynamic inference and caching of user-defined types), these differences do not affect a study of type checking.

Instrumentation. Our instrumentation utilized unused x86 opcodes that were inserted into the dynamically generated code at the entrance and exit of type-check code. We modified Nitro to insert these opcodes automatically during code generation. As they are intercepted by the simulation platform (described below), these special opcodes generate temporary event markers that allow us to break the execution into guard and non-guard regions for separate analysis. The markers themselves are not counted in our measurements.

We also inserted similar markers at the beginning and end of the execution of dynamically-generated code, as opposed to the compiler itself. This allows us to define the primary region of interest within the execution, *i.e.*, the section spent executing dynamically-generated code (as opposed to sections spent in the compiler and the interpreter). In many cases, we will demonstrate results both for the main region of interest and for the entire execution.

Simulation Platform. Because the Nitro VM generates code dynamically, it can only be executed on architectures for which it has a code generator, currently x86 and ARM. For this work we instrumented the Nitro VM running natively on an x86 processor, using the PIN binary instrumentation infrastructure [15]. We constructed a timing model that simulates a microarchitecture similar to the Qualcomm Snapdragon [19] (an ARM variant used in devices such as the Google Nexus One smartphone), including its memory hierarchy and branch prediction hardware. Note that, while Snapdragon's ISA is ARM and our simulation infrastructure utilizes x86, the first-order affects of our measurements are unlikely to be altered by the choice of ISA.

For our cache model, we simulate a two-level cache hierarchy. The L1 cache is 32KB in size, 4-way associative, with 32-byte cache lines. The L2 cache is 512KB and 8-way associative, with 64-byte cache lines. Memory accesses, in cycles, are 1 and 5 for L1 hits and misses, and 10 and 100 for L2 hits and misses, respectively.

For branch prediction, we implemented a two-level, history-correlated predictor [24], with a 12-bit branch history register (BHR) and a 4KB branch history table (BHT) composed of 2-bit saturating counters. We model a branch penalty of one cycle on a correct prediction, and 10 on a misprediction.

Benchmarks. We performed measurements on both major JavaScript benchmark suites, SunSpider [27] and V8 [13]. SunSpider is the older of the two, and is composed of short-running (250ms at most) kernels that are represen-

tative of common JavaScript tasks.

The V8 suite, in contrast, comprises longer running benchmarks that are scaled-down implementations of common algorithms, for instance, a constraint solver machine-translated from Scheme, with a scaled-down input size. The V8 benchmark harness is designed to count the number of iterations of each test that can be performed within a constant amount of time. Because of the slowdown introduced by our binary instrumentation methodology, this approach is not appropriate for our experiments. Therefore, we modified each V8 benchmark to execute a fixed number of iterations, and have scaled these iteration counts to produce a baseline of 5 to 10 seconds of native, uninstrumented execution. The SunSpider suite was executed unmodified.

While SunSpider and V8 are the standard JavaScript benchmark suites, several of their benchmarks are somewhat ill-suited for evaluating type checks.

- `regexp` and `regexp-dna` measure the performance of dynamically-generated, but *imperative*, regular expression code, rather than JavaScript, and thus evaluate very few type guards.
- `date-format-xparb` and `date-format-tofte` perform very little work, but make extensive use of `eval` to force frequent recompilation, thus forcing most of the time spent on the benchmark to be in the compiler rather than the generated code.
- `string-unpack-code` is a benchmark of the speed of compiling very large functions, which are then never called. Because the program itself does no work, very few type guards are evaluated.
- `bitops-bitwise-and` is a microbenchmark of a case where a combination of the tag encoding and the precise choice of tested operation, *i.e.,* bit-wise and, removes all but a single type guard on each loop iteration. Specifically, it allows a transformation that eliminates the guards on both operands in favor of a single guard on the resultant value.

We produce results for these benchmarks, but caution the reader to consider their relevance to real-world JavaScript code. With respect to real-world relevance in a more general sense, a recent study [20] found that SunSpider is more representative of today's JavaScript applications than V8, particularly with respect to memory allocation behavior and the use of language features, such as `eval`.

2.2 Instruction Counts

We first consider dynamic instruction costs, a simple proxy for energy, which is of critical importance on mobile processors. We simulated the benchmarks and collected dynamic instruction counts, using the compiler-inserted markers to break the execution into three regions: instructions spent in the interpreter and compiler, instructions spent in dynamically-generated non-guard code, and instructions spent in dynamically-generated type guards.

Figure 2(a) depicts the proportional breakdown of the latter two regions on both benchmark suites. The data illustrates that a significant component of the execution consists of instructions for implementing type guards. In the SunSpider benchmarks (the upper section of the Figure 2(a)), guard instructions consume more than 25% of the dynamic instruction count of generated code in a number of benchmarks, but there is significant variation, from a minimum of essentially zero in `regexp-dna` to 46.8% in `date-format-xparb`. The mean for these benchmarks is 10.9%.

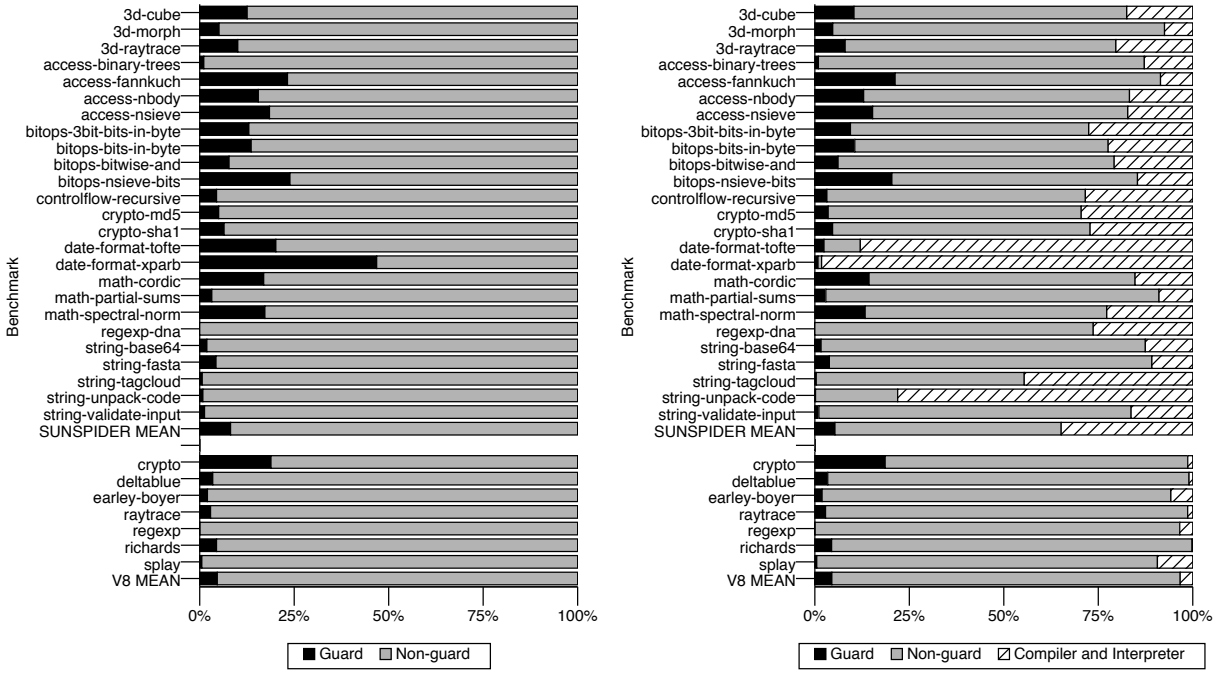
The overhead of dynamic type checking is less in the V8 suite (shown in the lower section of Figure 2(a)), with an average of 4.1%, because these benchmarks tend to be dominated by hash-table-based, user-defined types, which significantly outweigh type guards in per-operation cost. While accessing a user-defined type *does* require a type guard to check that the value is a pointer to an object rather than an integer, the cost of that guard is amortized over the entire hash table access, reducing its impact on overall performance.

Figure 2(b) puts this data in context with the overheads imposed by other parts of the virtual machine, particularly the compiler and the interpreter. In this context the average type-checking overhead drops to 6.3% of total dynamic instructions. However, as software techniques and JavaScript VMs improve, the number of instructions spent in interpretation and code generation will trend towards zero in steady-state, reflecting the higher values in Figure 2(a).

2.3 Timing

While instruction counts provide us with a basic impression of the costs of type checks, they do not give an accurate accounting of the performance of a modern mobile processor. We also collected timing information using our simulator to assess the performance impact of type guards on execution time (Figure 3(a)). Overall, a number of benchmarks exhibited greater than 10% of executed cycles spent in guard code, though again there was significant variation. The results range from a minimum of almost 0% on `regexp-dna` to a maximum of 62% on `date-format-xparb`, with a mean of 12.9% over all benchmarks. We observe a similar difference between the SunSpider and V8 benchmarks as above, with a mean cost of 14.7% on SunSpider and 6.5% on V8.

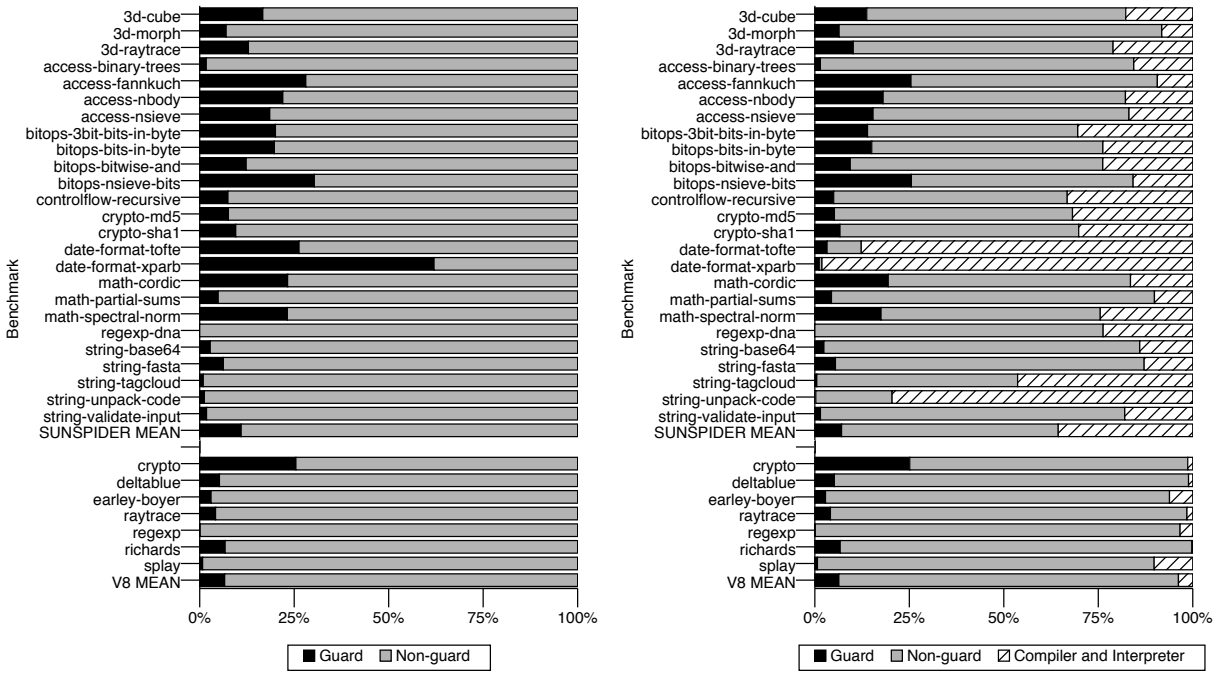
We can also view this data in its larger execution con-



(a) Region-of-interest (ROI) breakdown

(b) Whole-program breakdown

Figure 2. The dynamic instruction count cost of dynamic type checks in JavaScript benchmarks.



(a) Region-of-interest (ROI) breakdown

(b) Whole-program breakdown

Figure 3. The cycle cost of dynamic type checks in JavaScript benchmarks.

text, with the overheads of the compiler and the interpreter included (Figure 3(b)). Here the mean cost of type checks becomes 8.5% overall, and 8.8% and 6.4% for the SunSpider and V8 benchmarks, respectively. These numbers represent an upper limit on the performance improvement we can expect to see on these benchmarks from any attempts to reduce or remove the costs of type guards.

These results reflect the microarchitectures of today's mobile devices. Because *every* guard includes a branch, our data (not shown) indicate that branch mispredictions disproportionately impact the cost of guards. Consequently, as mobile pipelines become longer (as evidenced by the lengthening from 5 stages in ARM9 [2], to 8 in ARM11 [1], to 13 in the current ARM Cortex-A8 [3]), the relative cost of guard code should increase, as the balance between branch costs and other types of latencies shifts towards the former.

3 An ISA Extension for Type Checking

In order to reduce the performance impact of dynamic type checking, we propose an instruction set extension known as Checked Load. This extension comprises one new architected register, `chklp`, which holds the pointer to the type-error handler, and four new instructions for tag checking. Before discussing the implementation of these instructions, we first specify their complete semantics.

`chklb` – Takes as operands a memory location of the data to be loaded, a word-sized register for the destination, and a byte-sized immediate for the type tag. `chklb` is executed as a load, except that, when the value is accessed from the cache, the *most significant byte* is checked against the tag immediate. For implementation efficiency, the target address must be cache-block-aligned. On failure, control transfers to the error handler address stored in `chklp`. Note that the cache-line alignment restriction is not likely to be costly. In fact, both Nitro and TraceMonkey already force heap-allocated values to be cache-line aligned. Adopting Checked Load, then, simply requires reordering the bytes of these values to coincide with the required layout — no increased memory traffic and thus no loss of performance.

`chklbn` – Functions as `chklb`, except that the result of the tag comparison is negated before determining whether the control transfer occurs.

`chklw` – Takes as operands a memory location of the data to be loaded, a word-sized register for the destination, and a word-sized register for the type tag. It is executed as a load, except that, when the value is accessed from the cache, the *first word of the line* is checked against the register-resident tag. On failure, control transfers to the error handler address stored in `chklp`.

`chklwn` – Functions as `chklw`, except that the result of the tag comparison is negated before determining whether the control transfer occurs.

With respect to instruction encoding constraints, note that the size of the type tag in `chklb` and `chklbn` (a byte in our description) is flexible. While a wider tag eases VM implementation, narrower tags are feasible as well. LISP machines commonly used only two bits for tags, and we are confident that a modern JavaScript VM could reasonably be implemented with four-bit type tags. Alternatively, the tag could be stored in a second special register or a general purpose register (rather than an immediate field) to relieve pressure on the instruction encoding of the Checked Load instructions.

The benefits of using these instructions comes from several sources. From an instruction counting perspective, each Checked Load instruction is the fusion of two loads (the loaded field, and the tag), a comparison, and a branch, effectively replacing a common four instruction sequence with a single instruction. As we will demonstrate in Section 3.2, this fusion also allows the execution of the tag load and the comparison in parallel with the primary load, removing them from the critical path.

3.1 Code Generation

At a high level, code generation with Checked Load follows an optimistic policy: code for a "fast path" is generated assuming that the type guard will pass, using `chklw` or `chklb`, as appropriate, in place of the first load from the guarded object. The transfer of control to a failure handler is done by the hardware, without the need for special code generation beyond the initial loading of `chklp` to configure the error-handler register.

Because of the way modern JavaScript VMs generate code, a single such handler can be reused for an entire VM-level macro-op worth of instructions, approximately a single line of JavaScript source code. This choice of failure granularity is arbitrary from a correctness perspective, and is selected to achieve a balance between the complexity of code generation and exposing more optimization opportunities to the purely local code generator. If the cost of setting `chklp` for each macro-op proved prohibitive, the granularity could be coarsened to amortize the cost over a larger region, at the cost of slightly generalizing the error handler generated by the VM.

Given this context of how Checked Load can be used at a high level, we now detail the specific instruction sequences that generate code for type guards. In order to test if a value is an integer (as illustrated in Figure 4), the address of the error handler is first loaded into `chklp`. The initial load (from a virtual register, for example) in the guard is then replaced with a `chklb`. Because the most significant byte is used for the tag comparison, if the load completes successfully, the value can be used as is for integer arithmetic

(because of the tag encoding described in Section 2). A failure would transfer control to the registered handler.

```
TypeGuard:
  mov &ErrorHandler → %ChklReg
  chklb 0(%MemLoc), 0xFF → %DestReg
  ...

ErrorHandler:
  ...
```

Figure 4. Sample generated code for an integer guard.

In the case of a guard on a larger, object-like value, a combination of `chklbn` and `chklw` may be more appropriate. In this situation, there are two levels of indirection in accessing the value. First, the pointer to the value is loaded from a virtual register, and that value is guarded *against* being an integer; then an offset from that pointer is dereferenced to access the actual data.

To implement this using Checked Load, the first load is replaced with a `chklbn`, with the type *integer* for the immediate operand. Thus, when the load from the virtual register occurs, it is implicitly checked to ensure that it is *not* an integer. Then `chklw` loads through that pointer, checking the first word of the loaded cache line against the tag register. This case is illustrated in Figure 5. While a word-sized tag may initially appear excessive, it is necessary to support cases (observed in the Nitro VM) where the tag fulfills a secondary role as a virtual method table pointer, and thus must be pointer-sized.

```
IntGuard:
  mov &ErrorHandler → %ChklReg
  chklbn 0(%MemLoc), 0xFF → %DestReg

TypeGuard:
  mov 0xFF..FF → %TagReg
  chklw 0(%DestReg), %TagReg → %DestReg2

ErrorHandler:
  ...
```

Figure 5. Sample generated code for an object guard.

3.2 Microarchitecture

For efficiency, we implemented Checked Load type-tag checking in the cache itself. This takes the form of comparators attached to the most-significant-byte (for `chklb`), and to the first word of the cache line (for `chklw`). In addition, a single multiplexer selects the result from among the multiple cache ways, and an XOR gate accommodates the negation flag (*i.e.*, `chklb` vs. `chklbn`).

Such an implementation for `chklb` is diagrammed in Figure 6. On a cache hit, in parallel with the cache tag comparison, the type tag is compared against the relevant portion of the cache line in the set. Just as the cache tag comparison selects which way contains the target address, it also selects which result of the type tag comparisons to use. Finally, the XOR gate is used to implement the negation option. Similar hardware implements `chklw`.

We required earlier that `chklb` only be used to load values that are cache-line aligned. This forces these values to coincide with the first word of a cache line, as do all tag checks originating from `chklw`. Consequently, the Checked Load logic can be hardwired to a fixed word of each way, ensuring that the logic for evaluating a type guard failure is no deeper than that for checking the cache tag. The implementation allows a processor to add Checked Load to its ISA without lengthening its cache critical path. This in turn ensures that using Checked Load does not impact the cycle counts of other instructions or the cycle time of the processor.

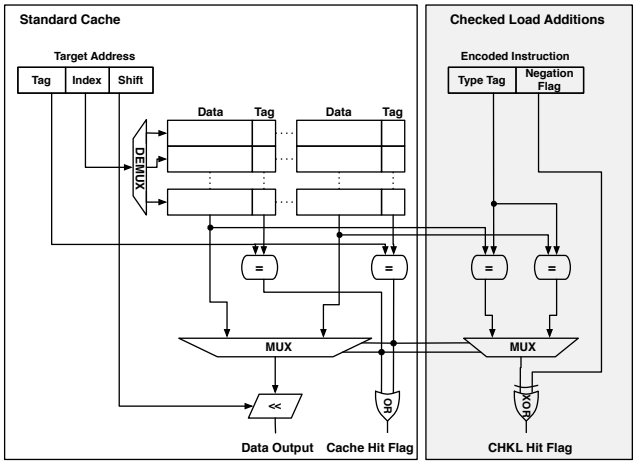


Figure 6. Implementation of `chklb` tag checking in parallel with cache tag checking

The implementation of the failure case of a Checked Load instruction simply copies `chklp` to the program counter.

Dynamic Type Prediction. Type check failures are expensive. In order to reduce their cost, we developed support for dynamic type prediction. The key idea is to predict whether a Checked Load is likely to fail, and if so, execute the slow path directly. We use the same dynamic prediction

mechanism used by branches: the instruction address and branch history are fed to a predictor; the output is a prediction bit that indicates whether the Checked Load is likely to fail or not. Fortunately, we can reuse the same structure used for actual branch prediction, saving on hardware complexity, but with little cost in performance. In the next section we evaluate the performance impact of type prediction, including the effect of sharing a predictor between Checked Load instructions and branches.

4 Evaluation

In order to evaluate the performance impact of Checked Load, we again employ the performance model developed in Section 2 to assess the improvements in execution time from replacing type guards with Checked Load instructions. We evaluate a Checked Load implementation in which the data type, *i.e.*, the implicit branch of `chklb` and `chklw`, is predicted with a standard correlated predictor (*gshare*).

In our first evaluation, the type-guard predictor is distinct from the hardware for general branch prediction but uses its design, namely, a 4096-entry history table (analogous to the branch history table) and a 12-bit history register (analogous to the branch history register). The Checked Load predictor dynamically maps each combination of the Checked Load PC address and the history register, using a simple XOR hash function, to a 2-bit saturating counter in the branch history table. The value of the counter is used to predict the direction of the branch, and is updated once the correct direction is resolved [24].

As can be seen in Figure 7(a), this implementation provides demonstrable performance benefits over software type checking, with a mean decrease in the cycle count of the region of interest of 11.9%. The benchmarks in the SunSpider suite tend to benefit more from Checked Load than those in the V8 suite, as evidenced by their respective means of 14.2% and 6.3%. This is likely due to the prevalence of numerical type checking (rather than hash-table-based type checking) in the SunSpider suite; in particular, note that `crypto`, the only arithmetically intensive benchmark in the V8 suite, shows a significantly above-average benefit.

However, since we are targeting resource-constrained mobile processors, expending chip area and energy on a separate BHT for Checked Load prediction may not be feasible. Two alternatives eliminate this cost: we can either use one set of prediction hardware for both standard branches and Checked Load, or we can forego dynamic Checked Load prediction altogether, utilizing static prediction. To implement the former, we merge the prediction tables for branches and Checked Load instructions into a single 4096-entry BHT, while preserving distinct 12-bit branch history registers for each. The static prediction implementation assumes that type guards rarely fail, predicting that the fall-through code fast-path will execute.

Figure 7(b) shows that merging Checked Load and general branch prediction into the same branch history table has little negative impact on performance. The mean improvement in the region of interest drops marginally to 11.2% (from 11.9%) and the relative application-specific behavior remains the same between the separate- and joined-table implementations.

However, results for the static alternative indicate that static prediction performs no better for Checked Load than it does for general branches. Figure 7(c) depicts the performance impact of statically predicted Checked Load for both benchmark suites. The mean improvement of static prediction is only 1.5%, a modest improvement over executing the type-checking code. For some benchmarks, such as `access-nbody`, static prediction results in a performance *loss* due to frequent type-check failures. Such occurrences are particularly common in functions that are deliberately type-generic, such as container libraries. Only programs in which type failures are rare are there significant benefits, but only up to 17.4%.

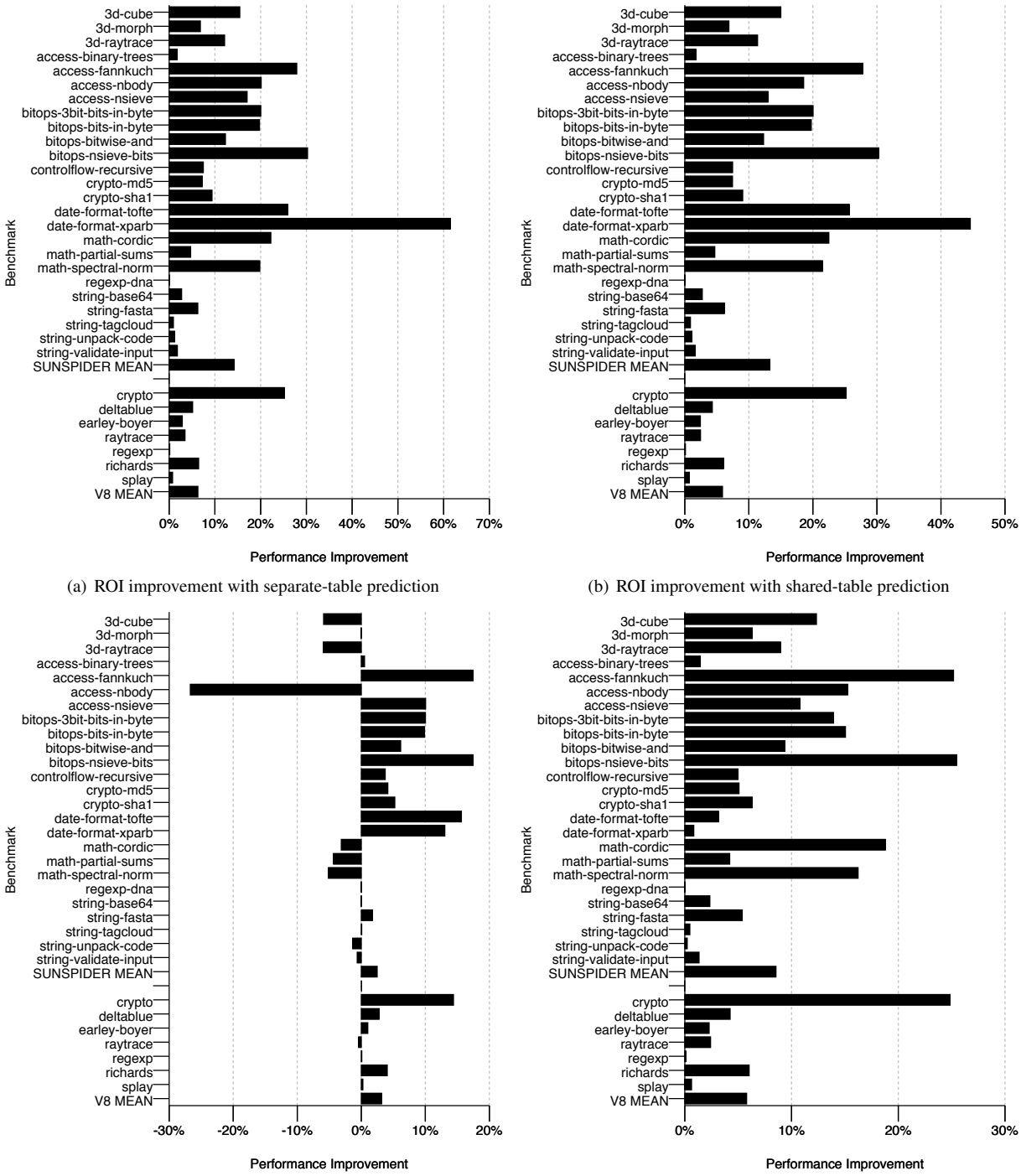
A sufficiently powerful JavaScript VM might improve these results through the use of feedback-directed code generation, generating code after collecting profile information and/or regenerating code over the course of program execution to adapt to program phase changes. The only JavaScript VM we are aware of that uses this technique is TraceMonkey.

To more specifically quantify the impact of the three branch prediction strategies for Checked Load instructions, we examine the correct prediction rates for for all three (Table 1). Overall, the data explain the level of performance of the three schemes – static prediction shows a significantly worse prediction rate than either dynamic approach. Additionally, we observe that the accuracy loss from moving to a joined-table from a separate-table implementation is minimal, despite halving the prediction hardware.

These measurements quantify performance for the region of interest, which excludes time spent in the interpreter and the compiler. Figure 7(d) shows the impact of these overheads, illustrating a mean overall improvement of 7.8% using Checked Load and the shared prediction hardware. As argued in Section 2.2, JavaScript VMs will likely increase in sophistication and software optimization techniques. To the extent that they do, the impact of these overheads on the overall performance of the system will trend downward, making the region of dynamically generated code the dominant factor in performance.

5 Related Work

Previous research on improving the performance of dynamically-typed languages can be divided into three



(a) ROI improvement with separate-table prediction

(b) ROI improvement with shared-table prediction

(c) ROI improvement with static prediction

(d) Whole-program improvement with shared-table prediction

Figure 7. Performance impacts of Checked Load

	Type of Prediction		
	Static	Dynamic Joined	Dynamic Separate
3d-cube	76.5%	97.7%	98.4%
3d-morph	86.6%	99.7%	99.3%
3d-raytrace	68.3%	96.9%	99.1%
access-binary-trees	92.3%	99.9%	99.9%
access-fannkuch	100%	99.7%	99.9%
access-nbody	58.0%	96.6%	98.2%
access-nsieve	100%	91.9%	99.9%
bitops-3bit-bits-in-byte	100%	100%	100%
bitops-bits-in-byte	100%	100%	100%
bitops-bitwise-and	100%	100%	100%
bitops-nsieve-bits	99.9%	99.9%	99.9%
controlflow-recursive	100%	100%	100%
crypto-md5	100%	99.6%	99.9%
crypto-sha1	100%	98.9%	99.9%
date-format-tofte	99.8%	99.4%	99.9%
date-format-xparb	85.2%	91.9%	99.9%
math-cordic	83.0%	99.3%	99.3%
math-partial-sums	69.0%	99.1%	99.1%
math-spectral-norm	76.3%	98.1%	99.9%
regexp-dna	81.4%	74.0%	57.1%
string-base64	86.4%	99.6%	100%
string-fasta	93.5%	99.7%	99.9%
string-tagcloud	87.0%	98.5%	99.9%
string-unpack-code	55.9%	97.8%	99.8%
string-validate-input	76.7%	97.9%	99.9%
SunSpider Mean	87%	97.4%	98.0%
crypto	100%	99.8%	99.9%
deltablue	99.6%	96.1%	99.6%
earley-boyer	94.2%	95.6%	99.4%
raytrace	76.6%	88.2%	97.3%
regexp	100%	94.3%	99.7%
richards	100%	97.4%	100%
splay	93.6%	98.0%	100%
V8 Mean	94.8%	95.6%	99.5%

Table 1. The prediction rates for Checked Load with static prediction, dynamic prediction with a separate branch history table, and dynamic prediction with a joined branch history table.

major areas, beginning with LISP machines, continuing through software techniques for accelerating early dynamically-typed, object-oriented languages like SmallTalk [8] and Self [14] and culminating in current work on trace-based, dynamic compilation.

Over time, the emphasis of research has shifted from hardware to software; we have proposed a re-examination of that trend, and have illustrated how hardware may be used profitably to accelerate dynamically-typed programs in ways that software cannot.

5.1 LISP Machines

The question of the performance cost of dynamic typing was first addressed in the context of LISP. In contrast to most later work, the LISP research focused on implementation concerns for primitive types (chiefly numerical types), while later work has focused more heavily on user-defined types. Steenkiste [23] provides the most comprehensive overview of the culmination of the LISP-derived research.

Steenkiste [22] observed that, even with dynamic type checking disabled, LISP programs spend 25% of their runtime manipulating type tags, while White [28] noted that these operations are dominated by integer types. In reaction, static type inference systems were developed [4, 21], based on recognizing type-specific operators and standard function calls, in particular, for statically detecting and efficiently compiling numerical operations.

On the hardware side, most LISP machine implementations [6, 16, 25] included instruction set support for operations on tagged pointers. Some also supported, in conjunction with static code multi-versioning, dynamic, type-based function dispatch.

Hardware support for LISP superficially resembles our approach, in that it offered ISA-level support for dynamic type checking. However, past support for type guards consisted solely of folding the guards into high-level instructions, and often required complex hardware structures (which likely lengthened critical paths). In contrast, we develop a low-complexity type-checking primitive that is integrated with the cache infrastructure and does not lengthen any critical paths.

5.2 SmallTalk and SELF

The second generation of research in dynamic typing stems from the rise of dynamically-typed, object-oriented languages, led by SmallTalk-80 [8]. For these languages, especially those that did away with the distinction between primitive, hardware-supported types (such as 32-bit integers) and objects (such as a hashtable), dynamic, type-based method dispatch became critical for performance.

The most direct antecedent of modern JavaScript virtual machines (VM) was the implementation of SELF [14], one of the earliest prototype-based, object-oriented languages. Research on efficient implementations of SELF pioneered a number of important JIT-based type specialization techniques, *e.g.*, type-feedback-directed inlining [14]. Other work on SELF introduced polymorphic inline caches [5], a combination of code multiversioning and software branch prediction that allows for high-performance type speculation. Many of the code generation techniques developed for SELF are used by modern JavaScript VMs.

5.3 Trace-based Compilation

In contrast to the traditional, method-oriented compilation techniques used in most virtual machines, trace-based compilation employs a profiler and an interpreter to identify repeated execution traces, and compiles these hot traces to highly-specialized native code. This technique was first

developed in HotPathVM [11], and is currently used in the TraceMonkey JavaScript VM in Mozilla Firefox [10].

Compared to traditional techniques, trace-based compilation offers heavy-weight, high-quality code generation. The collection and aggressive compilation of traces is expensive, but it allows the compiler to optimize away far more redundant operations. While the approach can offer significant performance gains for some benchmarks, its performance on real-world results is mixed, and light-weight, traditional compilers remain dominant in practice. The research in this paper uses the latter, a JavaScript VM called Nitro, previously known as SquirrelFish Extreme [26].

6 Conclusion

In this paper, we have demonstrated that the cost of type checks in dynamic languages such as JavaScript imposes a significant performance penalty on mobile platforms. This overhead is an inhibitor to the development of rich, portable web applications for mobile devices, and will only widen as the penalty of branch mispredictions on mobile processors increases, *i.e.,* as their pipelines deepen.

To address this problem, we have proposed Checked Load, an instruction set extension designed to reduce the type-checking penalty. We have shown how JavaScript applications could use Checked Load instructions to implement type guards, and have designed an efficient implementation that utilizes a mobile processor's branch prediction hardware to predict types. Improvements to JavaScript applications averaged 11.2% and ranged as high as 44.6%.

The performance problems associated with dynamic type checking are not new. Architects first grappled with them in the era of LISP machines, but it has come back to haunt us today. As we are running out of useful ways to use the silicon Moore's Law yields, and as dynamically-typed languages become the primary medium for rich, portable web applications, it is an appropriate time to address this old problem once and for all.

Acknowledgments

We thank the anonymous reviewers for their helpful feedback. We thank the SAMPA group at the University of Washington and Călin Caşcaval for their invaluable feedback on the manuscript and insightful discussions. This work was supported in part by NSF grant CCF-0702225 and a gift from Qualcomm.

References

[1] ARM. ARM11 Processor Family. `http://www.arm.com/products/processors/classic/arm11/index.php`.

[2] ARM. ARM9 Processor Family. `http://www.arm.com/products/processors/classic/arm9/index.php`.

[3] ARM. Cortex-A8 Processor. `http://www.arm.com/products/processors/cortex-a/cortex-a8.php`.

[4] R. A. Brooks, D. B. Posner, J. L. McDonald, J. L. White, E. Benson, and R. P. Gabriel. Design of an Optimizing, Dynamically Retargetable Compiler for Common LISP. In *Conference on LISP and Functional Programming (LFP)*, 1986.

[5] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. *SIGPLAN Notice*, 24(7), 1989.

[6] C. Corley and J. Statz. LISP Workstation Brings AI Power to a User's Desk. *Computer Design 24*, January 1985.

[7] D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Books, 2008.

[8] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the SmallTalk-80 System. In *Symposium on Principles of Programming Languages (POPL)*, 1984.

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

[10] 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. In *Conference on Programming language Design and Implementation (PLDI)*, 2009.

[11] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *International Conference on Virtual Execution Environments (VEE)*, 2006.

[12] Google Inc. V8 JavaScript Virtual Machine. `http://code.google.com/p/v8`, 2008.

[13] Google Inc. V8 Benchmark Suite – version 5. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html`, 2009.

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

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Conference on Programming language Design and Implementation (PLDI)*, 2005.

[16] D. A. Moon. Architecture of the Symbolics 3600. *SIGARCH Computer Architecture News*, 13(3), 1985.

[17] Palm. Palm Development Center. `http://developer.palm.com/`.

[18] Python Software Association. The Python Language Reference. `http://docs.python.org/reference/`.

[19] Qualcomm. The Snapdragon Platform. `http://www.qualcomm.com/products\_services/chipsets/snapdragon.html`.

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

[21] G. L. Steele. Fast Arithmetic in MacLISP. In *MACSYMA Users' Conference*, 1977.

[22] P. Steenkiste and J. Hennessy. LISP on a Reduced-Instruction-Set-Processor. In *Conference on LISP and Functional Programming (LFP)*, 1986.

[23] P. Steenkiste and J. Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Architectural Support for Programming Languages and Operating Systens (ASPLOS)*, 1987.

[24] T. Yeh and Y. Patt. Two-level Adaptive Training Branch Prediction. In *International Symposium on Microarchitecture (MICRO)*, 1991.

[25] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR LISP Architecture. *SIGARCH Computer Architecture News*, 14(2), 1986.

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

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

[28] J. L. White. Reconfigurable, Retargetable BigNums: A Case Study in Efficient, Portable LISP System Building. In *Conference on LISP and Functional Programming (LFP)*, 1986.