Vector Parallelism in JavaScript: Language and compiler support for SIMD

Ivan Jibaja
University of Texas at Austin
ivan@cs.utexas.edu

Peter Jensen, Ningxin Hu, Mohammad R. Haghighat
Intel Corporation
{*peter.jensen, ningxin.hu, mohammad.r.haghighat*}*@intel.com*

John McCutchan
Google Inc.
johnmccutchan@google.com

Dan Gohman
Mozilla
dgohman@mozilla.com

Stephen M. Blackburn
Australian National University
steve.blackburn@anu.edu.au

Kathryn S. McKinley
Microsoft Research
mckinley@microsoft.com

***Abstract*—JavaScript is the most widely used web programming language and it increasingly implements sophisticated and demanding applications such as graphics, games, video, and cryptography. The performance and energy usage of these applications benefit from hardware parallelism, including SIMD (Single Instruction, Multiple Data) vector parallel instructions. JavaScript's current support for parallelism is limited and does not directly exploit SIMD capabilities. This paper presents the design and implementation of SIMD language extensions and compiler support that together add fine-grain vector parallelism to JavaScript. The specification for this language extension is in final stages of adoption by the JavaScript standardization committee and our compiler support is available in two open-source production browsers. The design principles seek portability, SIMD performance portability on various SIMD architectures, and compiler simplicity to ease adoption. The design does not require automatic vectorization compiler technology, but does not preclude it either. The SIMD extensions define immutable fixed-length SIMD data types and operations that are common to both ARM and x86 ISAs. The contributions of this work include *type speculation* and optimizations that generate minimal numbers of SIMD native instructions from high-level JavaScript SIMD instructions. We implement type speculation, optimizations, and code generation in two open-source JavaScript VMs and measure performance improvements between a factor of 1.7× to 8.9× with an average of 3.3× and average energy improvements of 2.9× on micro benchmarks and key graphics kernels on various hardware, browsers, and operating systems. These portable SIMD language extensions significantly improve compute-intensive interactive applications in the browser, such as games and media processing, by exploiting vector parallelism without relying on automatic vectorizing compiler technology, non-portable native code, or plugins.**

I. Introduction

Increasingly more computing is performed in web browsers. Since JavaScript is the dominant web language, sophisticated and demanding applications, such as games, multimedia, finance, and cryptography, are increasingly implemented in JavaScript. Many of these applications benefit from hardware parallelism, both at a coarse and fine grain. Because of the complexities and potential for concurrency errors in coarse grain (task level) parallel programing, JavaScript has limited its parallelism to asynchronous activities that do not communicate through shared memory [21]. However, fine-grain vector parallel instructions — Single-Instruction, Multiple-Data (SIMD) — do not manifest these correctness issues and yet they still offer significant performance advantages by exploiting parallelism. This paper describes motivation, design goals, language specification, compiler support, and two open-source VM implementations for the SIMD language extensions to JavaScript that are in the final stages of adoption by the JavaScript Standards committee [20].

SIMD instructions are now standard on modern ARM and x86 hardware from mobile to servers because they are both high performance and energy efficient. SIMD extensions include SSE4 for x86 and NEON for ARM. These extensions are already widely implemented in x86 processors since 2007 and in ARM processors since 2009. Both extensions implement 128 bits and x86 processors now include larger widths in the AVX instruction set. For example, this year Intel has released the first CPUs with AVX 512 bit vector support. However, ARMs largest width is currently 128 bits. These instruction set architectures include vector parallelism because it is very effective at improving performance and energy efficiency in many application domains, for example, image, audio, and video processing, perceptual computing, physics engines, fluid dynamics, rendering, finance, and cryptography. Such applications also increasingly dominate client-side and server-side web applications. Exploiting vector parallelism in JavaScript should therefore improve performance and energy efficiency on mobile, desktop, and server, as well as hybrid HTML5 mobile JavaScript applications.

Design: This paper presents the design, implementation, and evaluation of SIMD language extensions for JavaScript. We have two design goals for these extensions. (1) Portable vector performance on vector hardware. (2) A compiler implementation that does not require automatic vectorization technology to attain vector performance. The first goal helps developers improve the performance of their applications without unpleasant and unexplainable performance surprises on different vector hardware. The second goal simplifies the job of realizing vector performance in existing and new JavaScript Virtual Machines (VMs) and compilers. Adding dependence testing and loop transformation vectorizing technology is possible, but our design and implementation do not require it to deliver vector performance.

This paper defines SIMD language extensions and new compiler support for JavaScript. The language extensions consist of fixed-size immutable vectors and vector operators, which correspond to hardware instructions and vector sizes common to ARM and x86. The largest size common to both is 128 bits. Although an API with variable or larger sizes (e.g., Intel's AVX 512-bit vectors) may seem appealing, correctly generating code that targets shorter vector instructions from longer ones violates our vector performance portability design goal. For example, correctly shortening non-streaming vector instructions, such as shuffle/swizzle, requires generating scalar code that reads all values out and then stores them back, resulting in scalar performance instead of vector performance on vector hardware.

We define new SIMD JavaScript data types (e.g. Int32x4, Float32x4), constructors, lane accessors, operators (arithmetic, bitwise operations, comparisons, and swizzle/shuffle), and typed array accessors and mutators for these types. To ease the compiler implementation, most of these SIMD operations correspond directly to SIMD instructions common to the SSE4 and NEON extensions. We choose a subset that improve a wide selection of sophisticated JavaScript applications, but this set could be expanded in the future. This JavaScript language specification was developed in collaboration with Google for the Dart programming language, reported in a workshop paper [14]. The Dart and JavaScript SIMD language specifications are similar in spirit. The language extensions we present are in the final stages of approval by the ECMAScript standardization committee (Ecma TC39) [20].

Type Speculation: We introduce *type speculation*, a modest twist on type inference and specialization for implementing these SIMD language extensions. For every method containing SIMD operations, the Virtual Machine's Just-in-Time (JIT) compiler immediately produces SIMD instructions for those operations. The JIT compiler speculates that every high-level SIMD instruction operates on the specified SIMD type. It translates the code into an intermediate form, optimizes it, and generates SIMD assembly. In most cases, it produces optimal code with one or two SIMD assembly instructions for each high-level SIMD JavaScript instruction. The code includes guards that check for non-conforming types that reverts to unoptimized code when needed. In contrast, modern JIT compilers for dynamic languages typically perform some mix of *type inference*, which uses static analysis to *prove* type values and eliminate any dynamic checks, and *type feedback*, which *observes* common types over multiple executions of a method and then optimizes for these cases, generating guards for non-conforming types [1, 11]. Our JIT compilers instead will use the assumed types of the high-level SIMD instructions as hints and generate code accordingly.

Implementation: We implement and evaluate this compiler support in two JavaScript Virtual Machines (V8 and SpiderMonkey) and generate JIT-optimized SIMD instructions for x86 and ARM. Initially, V8 uses a simple JIT compiler (full codegen) to directly emit executable code [7], whereas SpiderMonkey uses an interpreter [6, 18]. Both will detect hot sections and later JIT compilation stages will perform additional optimizations. We add JIT type speculation and SIMD optimizations to both Virtual Machines (VMs). Our JIT compiler implementations include type speculation, SIMD method inlining, SIMD type unboxing, and SIMD code generation to directly invoke the SIMD assembly instructions. When the target architecture does not contain SIMD instructions or the dynamic type changes from the SIMD class to some other type, SpiderMonkey currently falls back on interpretation and V8 generates deoptimized (boxed) code.

Benchmarks: For any new language features, we must create benchmarks for evaluation. We create microbenchmarks by extracting ten kernels from common application domains. These kernels are hot code in these algorithms that benefit from vector parallelism. In addition, we report results for one application, skinning, a key graphics algorithm that associates the skin over the skeleton of characters from a very popular game engine. We measure the benchmarks on five different Intel CPUs (ranging from an Atom to an i7), and four operating systems (Windows, Unix, OS X, Android). The results show that SIMD instructions improve performance by a factor of 3.4× on average and improve energy by 2.9× on average. SIMD achieves super-linear speed ups in some benchmarks because the vector versions of the code eliminate intermediate operations, values, and copies. On the skinning graphics kernel, we obtain a speedup of 1.8×.

Artifact: The implementations described in this paper are in Mozilla Firefox Nightly builds and in submission to Chromium. We plan to generate optimized scalar code for machines that lack SIMD instructions in future work. This submission shows that V8 and SpiderMonkey can support SIMD language extensions without performing sophisticated dependence testing or other parallelism analysis or transformations, i.e., they do not *require* automatic vectorization compiler technology. However, our choice does not *preclude* such sophisticated compiler support, or preprocessor/developer-side vectorization support in tools such as Emscripten [17], or higher level software abstractions that target larger or variable size vectors, as applicable, to further improve performance. By adding portable SIMD language features to JavaScript, developers can exploit vector parallelism to make demanding applications accessible from the browser. We expect that these substantial performance and energy benefits will inspire a next generation of JIT compiler technology to further exploit vector parallelism.

Contributions: This paper presents the design and implementation of SIMD language extensions and compiler support for JavaScript. No other high level language has provided direct access to SIMD performance in an architecture-independent manner. The contributions of this paper are as follows:

1) a language design justified on the basis of portability and performance;
2) compiler type speculation without profiling in a dynamic language; and
3) the first dynamic language with SIMD instructions that deliver their performance and energy benefits.

II. Background and Related Work

Westinghouse was the first to investigate vector parallelism in the early 1960s, envisioning a co-processor for mathematics, but cancelled the effort. The principal investigator, Daniel Slotnick, then left and joined University of Illinois, where he lead the design of the ILLIAC IV, the first supercomputer and vector machine [3]. In 1972, it was 13 times faster than any other machine at 250 MFLOPS and cost $31 million to build. CRAY Research went on to build commercial vector machines [4] and researchers at Illinois, CRAY, IBM, and Rice University pioneered compiler technologies that correctly transformed scalar programs into vector form to exploit vector parallelism.

Today, Intel, AMD, and ARM processors for servers, desktop, and mobile offer coarse-grain multicore and fine-grain vector parallelism with Single Instruction Multiple Data (SIMD) instructions. For instance, Intel introduced MMX instructions in 1997, the original SSE (Streaming SIMD Extensions) instructions in 1999, and its latest extension, SSE4, in 2007 [9, 10]. All of the latest AMD and Intel machines implement SSE4.2.ARM implements vector parallelism with its NEON SIMD instructions, which are optional in Cortex-A9 processors, but standard in all Cortex-A8 processors.

The biggest difference between vector instruction sets in x86 and ARM is vector length. The AVX-512 instruction set in Intel processors defines vector lengths up to 512 bits. However, NEON defines vector lengths from 64-bit up to 128-bit. To be compatible with both x86 and ARM vector architectures and thus attain vector-performance portability, we choose one fixed-size 128-bit vector length, since it is the largest size that both platforms support. Choosing a larger or variable size than all platforms support is problematic when executing on machines that only implement a shorter vector size because some long SIMD instructions can only be correctly implemented with scalar instructions on shorter vector machines. See our discussion below for additional details. By choosing the largest size all platforms support, we avoid exposing developers to unpleasant and hard to debug performance degradations on vector hardware. We choose a fixed size that all architectures support to deliver performance portability on all vector hardware.

Future compiler analysis could generate code for wider vector instructions to further improve performance, although the dynamically typed JavaScript setting makes this task more challenging than, for example, in Fortran compilers. We avoided a design choice that would require significant compiler support because of the diversity in JavaScript compiler targets, from embedded devices to servers. Our choice of a fixed-size vector simplifies the programming interface, compiler implementation, and guarantees vector performance on vector hardware. Supporting larger vector sizes could be done in a library with machine-specific hooks to attain machine-specific benefits on streaming SIMD operations, but applications would suffer machine-specific performance degradations for non-streaming operations, such as shuffle, because the compiler must generate scalar code when an architecture supports only the smaller vector sizes.

Both SSE4 and NEON define a plethora of SIMD instructions, many more than we currently propose to include in JavaScript. We choose a subset for simplicity, selecting operations based on an examination of demanding JavaScript applications, such as games. Most of our proposed JavaScript SIMD operations map directly to a hardware SIMD instruction. We include a few operations, such as shuffleMix (also known as swizzle), that are not directly implemented in either architecture, but are important for the JavaScript applications we studied. For these operations, the compiler generates two or three SIMD instructions, rather than just one. The current set is easily extended.

Intel and ARM provide header files which define SIMD intrinsics for use in C/C++ programs (`xmmintrin.h` and `arm_neon.h`, respectively). These intrinsics directly map to each SIMD instruction in the hardware, thus there are currently over 400 intrinsic functions [10]. Similarly, ARM implements NEON intrinsics for C and C++ [2]. These platform-specific intrinsics result in architecture-dependent code, thus using either one directly in JavaScript is not desirable nor an option for portable JavaScript code.

Managed languages, such as Java and C#, historically only provide access to SIMD instructions through their native interfaces, JNI (Java Native Interface) and C library in the case of Java and C# respectively, which use the SIMD intrinsics. However, recently Microsoft and the C# Mono project announced a preliminary API for SIMD programming for .NET [15, 16]. This API is directly correlated to the set of SSE intrinsics, which limits portability across different SIMD instruction sets. In C#, the application can query the hardware to learn the maximum vector length and version of the SSE standard on the architecture. This API results in architecture-specific code embedded in the application, is not portable, and is thus not appropriate for JavaScript.

Until now, dynamic scripting languages, such as PHP, Python, Ruby, Dart, and JavaScript, have not included SIMD support in their language specification. We analyzed the application space and chose the operations based on their popularity in the applications and their portability across the SSE3, SSE4.2, AVX, and NEON SIMD instruction sets. We observed a few additional SIMD patterns that we standardize as methods, which the JIT compiler translates into multiple SIMD instructions.

III. Design Rationale

Our design is based on fixed-width 128-bit vectors. A number of considerations influenced this decision, including the programmer and the compiler writer.

A fixed vector width offers simplicity in the form of consistent performance and consistent semantics across vector architectures. For example, the number of times a loop iterates is not affected by a change in the underlying hardware. A variable-width vector or a vector width larger than the hardware supports places significant requirements on the JIT compiler. Given the variety of JavaScript JIT VMs and the diverse platforms they target, requiring support for variable-width vectors was considered unviable. Additionally, variable width vectors cannot efficiently implement some important algorithms (e.g. matrix multiplication, matrix inversion, vector transform). On the other hand, developers are free to add more aggressive JIT compiler functionality that exploits wider vectors if the hardware provides them. Another consideration is that JavaScript is heavily used as a compiler target. For example, Emscripten compiles from C/C++ [**?**], and compatibility with `mmintrin.h` offered by our fixed width vectors is a bonus.

Finally, given the decision to support fixed width vectors, we selected 128 bits because it is the widest vector supported by all major architectures today. Not all instructions can be decomposed to run on a narrower vector instruction. For example, non-streaming operations, such as the shuffle instruction, in general cannot utilize the SIMD hardware at all when the hardware is narrower than the software. For this reason, we chose the largest common denominator. Furthermore, 128 bits is a good match to many important algorithms, such as single-precision transformations over homogeneous coordinates in computer graphics (XYZW) and algorithms that manipulate the RGBA color space.

IV. Language Specification

This section presents the SIMD data types, operations, and JavaScript code samples. The SIMD language extensions give direct control to the programmer and require very simple compiler support, but still guarantees vector performance when the hardware supports SIMD instructions. Consequently, most of the JavaScript SIMD operations have a one-to-one mapping to common hardware SIMD instructions. This section includes code samples for the most common data types. The full specification is available on line [20].

A. Data Types

We add the following three new fixed-width 128-bit numeric value types to JavaScript.

Float32x4	Vector with four 32-bit single-precision floats
Int32x4	Vector with four 32-bit signed integers
Int16x8	Vector with 8 16-bit signed integers
Int8x16	Vector with 16 8-bit signed integers
Uint32x4	Vector with 4 32-bit unsigned integers
Uint16x8	Vector with 8 16-bit unsigned integers
Uint8x16	Vector with 16 8-bit unsigned integers
Bool32x4	Vector with 4 boolean values
Bool16x8	Vector with 8 boolean values
Bool8x16	Vector with 16 boolean values

Figure 1 shows the simple SIMD type hierarchy. Each SIMD types has four to sixteen *lanes*, which correspond to degrees of SIMD parallelism. Each element of a SIMD vector is a lane. Indices are required to access the lanes of vectors. For instance, the following code declares and initializes a SIMD single-precision float and assigns `3.0` to `a`.

```
var V1 = SIMD.Float32x4 (1.0, 2.0, 3.0, 4.0);
var a  = SIMD.Float32x4.extractLan(V1,3);
```

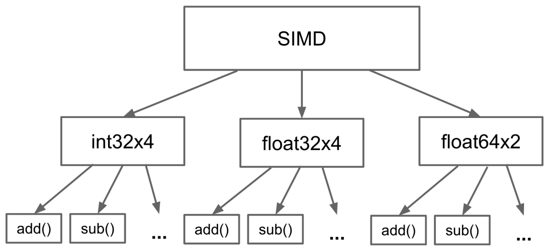

Figure 1: SIMD Type Hierarchy

B. Operations

Constructors: The type defines the following constructors for all of the SIMD types. The default constructor initializes each of the two or four lanes to the specified values, the `splat` constructor creates a constant-initialized SIMD vector, as follows.

```
var c = SIMD.Float32x4(1.1, 2.2, 3.3, 4.4);
// Float32x4(1.1, 2.2, 3.3, 4.4)
var b = SIMD.Float32x4.splat(5.0);
// Float32x4(5.0, 5.0, 5.0, 5.0)
```

Accesssors and Mutators: The proposed SIMD standard provides operations for accessing and mutating SIMD values, and for creating new SIMD values from variations on existing values.

extractLane	Access one of the lanes of a SIMD value.
replaceLane	Create a new instance with the value change for the specified lane.
select	Create a new instance with selected lanes from two SIMD values.
swizzle	Create a new instance from another SIMD value, shuffling lanes.
shuffle	Create a new instance by shuffling from the first SIMD value into the XY lanes and from the second SIMD value into the ZW lanes.

These operations are straightforward and below we show a few examples.

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = a.x; // 1.0
var c = SIMD.Float32x4.replaceLane(1, 5.0);
// Float32x4(5.0, 2.0, 3.0, 4.0)
var d = SIMD.Float32x4.swizzle(a, 3, 2, 1, 0);
// Float32x4(4.0, 3.0, 2.0, 1.0)
var f = SIMD.Float32x4(5.0, 6.0, 7.0, 8.0);
var g = SIMD.Float32x4.shuffle(a, f, 1, 0, 6, 7);
// Float32x4(2.0, 1.0, 7.0, 8.0)
```

Arithmetic: The language extension supports the following thirteen arithmetic operations over SIMD values: **add, sub, mul, div, abs, max, min, sqrt, reciprocalApproximation, reciprocalSqrtApproximation, neg, clamp, scale, minNum, maxNum**
We show a few examples below.

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.Float32x4(4.0, 8.0, 12.0, 16.0);
var c = SIMD.Float32x4.add(a,b);
// Float32x4(5.0, 10.0, 15.0, 20.0)
var e = SIMD.reciprocalSqrtApproximation(d);
// Float32x4(0.5, 0.5, 0.5, 0.5);
var f = SIMD.scale(a, 2);
// Float32x4(2.0, 4.0, 6.0, 8.0);
var lower = SIMD.Float32x4(-2.0, 5.0, 1.0, -4.0);
var upper = SIMD.Float32x4(-1.0, 10.0, 8.0, 4.0);
var g = SIMD.Float32x4.clamp(a, lower, upper);
// Float32x4(-1.0, 5.0, 3.0, 4.0)
```

Bitwise Operators: The language supports the following four SIMD bitwise operators: **and, or, xor, not**

Bit Shifts: We define the following logical and arithmetic shift operations and then show some examples: **shiftLeftByScalar, shiftRightLogicalByScalar, shiftRightArithmeticByScalar**

```
var a = SIMD.Int32x4(6, 8, 16, 1);
var b = SIMD.Int32x4.shiftLeftByScalar(a,1);
// Int32x4(12, 16, 32, 2)
var c = SIMD.Int32x4.shiftRightLogicalByScalar(a, 1);
// Int32x4(3, 4, 8, 0)
```

Comparison: We define three SIMD comparison operators that yield SIMD boolean values where 0xF and 0x0 represent true and false respectively: **equal, notEqual, greaterThan, lessThan, lessThanOrEqual, greaterThanOrEqual**

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.Float32x4(0.0, 3.0, 5.0, 2.0);
var gT = SIMD.Float32x4.greaterThan(a, b);
// Float32x4(0xF, 0x0, 0x0, 0xF);
```

Type Conversion: We define type conversion from floating point to integer and bit-wise conversion (i.e., producing an integer value from the floating point bit representation): **fromInt32x4, fromFloat32x4, fromFloat64x2, fromInt32x4Bits, fromFloat32x4Bits, fromFloat64x2Bits**

```
var a = SIMD.Float32x4(1.1, 2.2, 3.3, 4.4)
var b = SIMD.Int32x4.fromFloat32x4(a)
// Int32x4(1, 2, 3, 4)
var c = SIMD.Int32x4.fromFloat32x4Bits(a)
// Int32x4(1066192077, 1074580685, 1079194419,
   1082969293)
```

Arrays: We introduce load and store operations for JavaScript typed arrays for each base SIMD data type that operates with the expected semantics. An example is to pass in a `Uint8Array` regardless of SIMD type, which is useful because it allows the compiler to eliminate the shift in going from the index to the pointer offset. The extracted SIMD type is determined by the type of the load operation.

```
var a   = new Float32Array(100);
for (var i = 0, l = a.length; ++i) {
  a[i] = i;
}
for (var j = 0; j < a.length; j += 4) {
  sum4 = SIMD.Float32x4.add(sum4,
              SIMD.Float32x4.load(a, j));
}
var result = SIMD.Float32x4.extractLane(sum4, 0) +
             SIMD.Float32x4.extractLane(sum4, 1) +
             SIMD.Float32x4.extractLane(sum4, 2) +
             SIMD.Float32x4.extractLane(sum4, 3);
```

Figure 2 depicts how summing in parallel reduces the number of sum instructions by a factor of the width of the SIMD vector, in this case four, plus the instructions needed to sum the resulting vector. Given a sufficiently long array and appropriate JIT compiler technology, the SIMD version reduces the number of loads and stores by about 75%. This reduction in instructions has the potential to improve performance significantly in many applications.

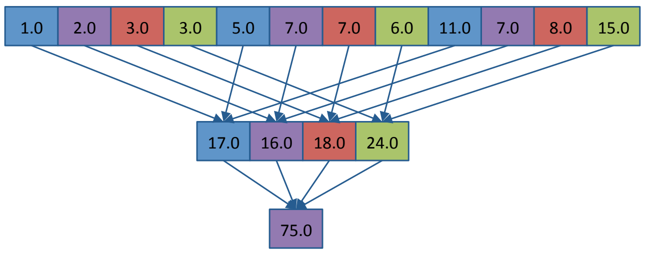

Figure 2: Visualization of `averagef32x4` summing in parallel.

V. Compiler Implementations

We add compiler optimizations for SIMD instructions to Firefox's SpiderMonkey VM [6, 18] and Chromium's V8

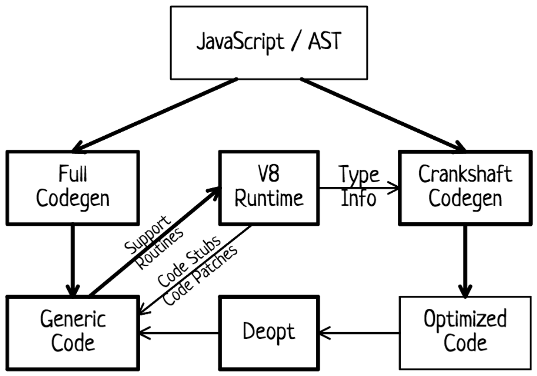

Figure 3: V8 Engine Architecture.

VM [7]. We first briefly describe both VM implementations and then describe our type speculation, followed by unboxing, inlining, and code generation that produce SIMD assembly instructions.

SpiderMonkey: We modified the open-source SpiderMonkey VM, used by the Mozilla Firefox browser. SpiderMonkey contains an interpreter that executes unoptimized JavaScript bytecodes and a Baseline compiler that generates machine code. The interpreter collects execution profiles and type information [18]. Frequently executed JS functions, as determined by the profiles collected by the interpreter, are compiled into executable instructions by the Baseline compiler. The Baseline compiler mostly generates calls into the runtime and relies on inline caches and hidden classes for operations such as property access, function calls, array indexing operations, etc. The Baseline compiler also inserts code to collect execution profiles for function invocations and to collect type information. If a function is found to be hot, the second compiler is invoked. This second compiler, called IonMonkey, is an SSA-based optimizing compiler, which uses the type information collected from the inline cache mechanism to inline the expected operations, thereby avoiding the call overhead into the runtime. IonMonkey then emits fast native code translations of JavaScript. We added SIMD support in the runtime. Instead of waiting for the runtime to determine whether the method is hot, we perform type speculation and optimizations for all methods that contain SIMD operations. We modify the interpreter, the Baseline compiler code generator, and the IonMonkey JIT compiler. We add inlining of SIMD operations to the IonMonkey compiler. We modify the register allocator and bailout mechanism to support 128-bit values.

V8: We also modified the open-source V8 VM, used by the Google Chromium browser. Figure 3 shows the V8 Engine Architecture. V8 does not interpret. It translates the JavaScript AST (abstract syntax tree) into executable instructions and calls into the runtime when code is first executed, using the Full Codegen compiler. Figure 4 shows an example of the non-optimized code from this compiler. This compiler also inserts profiling and calls to runtime support routines. When V8 detects a hot method, the Crankshaft compiler translates into an SSA form. It uses the frequency and type profile information to perform global optimizations across basic blocks such as type specialization, inlining, global value numbering, code hoisting, and register allocation. We modify the runtime to perform type speculation when it detects methods with SIMD instructions, which invokes the SIMD compiler support. We added SIMD support to both the Full Codegen compiler and the Crankshaft compiler. For the Full Codegen compiler, the SIMD support is provided via calls to runtime functions implemented in C++, as depicted in Figure 4. We modified the Crankshaft compiler, adding support for inlining, SIMD register allocation, and code generation which produces optimal SIMD code sequence and vector performance in many cases.

Both compilers use frequency and type profiling to inline and then perform type specialization on other types, other optimizations, register allocation, and finally generate code.

A. Type Speculation

Optimizing dynamic languages requires type specialization [8, 13], which emits code for the common type. This code must include an initial type tests and a branch to deoptimized generic code or jumps back to the interpreter or deoptimized code when the types do not match [12]. Both optimizing compilers perform similar forms of type specialization. In some cases, the compiler can use type inference to prove that the type will never change and can eliminate this fallback [1, 5]. For example, unboxing an object and its fields generates code that operates directly on floats, integers, or doubles, rather than generating code that looks up the type of every field on each access, loads the value from the heap, operates on them, and then stores them back into the heap. While a value is unboxed, the compiler assigns them to registers and local variables, rather than emitting code that operates on them in the heap, to improve performance.

The particular context of a SIMD library allows us to be more agressive than typical type specialization. We *speculate* types based on a number of simple assumptions. We consider it a design bug on the part of the programmer to override methods of the SIMD API, and thus we produce code speculatively for the common case of SIMD methods operating on SIMD types specified by our language extension. If programs override the SIMD methods, type guards that the compiler inserts in the program correctly detects this case, but performance suffers significantly. Likewise, we speculate that SIMD API arguments are of the correct type and optimize accordingly. If they are not, the compiler correctly detects this case, but performance will suffer.

Original JavaScript Code

```
    sum = SIMD.float32x4.add(sum, f32x4list.getAt(i));
```

V8 Full Codegen code (not optimized)

```
3DB5BCBE   222  ff3424          push [esp]
3DB5BCC1   225  89442404        mov [esp+0x4],eax
3DB5BCC5   229  ff75e8          push [ebp-0x18]
3DB5BCC8   232  ba02000000      mov edx,00000002
3DB5BCCD   237  8b7c2408        mov edi,[esp+0x8]
;; call .getAt()
3DB5BCD1   241  e84a24fcff      call 3DB1E120
3DB5BCD6   246  8b75fc          mov esi,[ebp-0x4]
3DB5BCD9   249  890424          mov [esp],eax
3DB5BCDC   252  ba04000000      mov edx,00000004
3DB5BCE1   257  8b7c240c        mov edi,[esp+0xc]
;; call .add()
3DB5BCE5   261  e876fdffff      call 3DB5BA60
3DB5BCEA   266  8b75fc          mov esi,[ebp-0x4]
3DB5BCED   269  83c404          add esp,0x4
3DB5BCF0   272  8945ec          mov [ebp-0x14],eax
```

V8 CrankShaft Code (optimized)

```
3DB5E306   358  0f101cc6        movups xmm3,[esi+eax*8]
3DB5E30A   362  0f58d3          addps xmm2,xmm3
```

Figure 4: Example V8 compiler generated code

These assumptions are predicated on the fact that the SIMD instructions have well established types and semantics, and that developers who use the API are expected to write their code accordingly. Because we expect developers to use SIMD instructions in performance-sensitive settings, we have the opportunity to aggressively optimize methods that contain them more eagerly, rather than waiting for these methods to become hot. We expect the net result to be a performance win in a dynamic optimization setting. Each of these simple optimizations is a modest twist on conventional JIT optimization of dynamic code that we tailor for the performance critical SIMD setting.

Inlining: To achieve high performance with the proposed SIMD language extension, we modified the optimizing compilers to always replace method calls to SIMD operations on SIMD types with inlined lower level instructions (IR or machine-level) that operate on unboxed values. The compilers thus eliminates the need for unboxing by keeping the values in registers. The compiler identifies all the SIMD methods and inlines them. These methods are always invoked on the same SIMD type and with the same parameters with the appropriate SIMD or other type. Thus, inlining is performed when the system observes the dynamic types of each SIMD method call and predicts they are monomorphic.

Value Boxing and Unboxing: Both baseline compilers will box SIMD objects, arguments, and return values like any regular JavaScript object. Both of the JavaScript VM JIT compilers optimize by converting boxed types to unboxed values [13]. As discussed above, boxed values are allocated on the heap, garbage collected, and must be loaded and stored to the heap on each use and definition, respectively. To improve performance, the optimizing compilers put unboxed values in registers, operate on them directly, and then stores modified values back to the heap as necessary for correctness and deoptimization paths. We modified this mechanism in both compilers to operate over SIMD methods.

Example: Consider again the `averagef32x4()` method from Section IV. In V8, the Full Codegen compiler generates the code in Figure 4, which is a straight forward sequence of calls into the V8 runtime. The parameters reside in heap objects. Note below that the parameters reside in heap objects and pointers to those heap objects are passed on the stack. The two calls invoke the .add() operator and the .getAt operator, respectively. The runtime has its own established calling convention using registers. However, all user visible values are passed on the stack.

The V8 Crankshaft compiler generates an SSA IR, directly represents SIMD values in registers, and uses SIMD instructions directly instead of runtime calls. The final code produced by the V8 Crankshaft optimizing compiler; after inlining, unboxing, and register allocation is the optimal sequence of just two instructions, as illustrated at the bottom of Figure 4. SpiderMonkey generates the same code.

The code has just two instructions; one for fetching the value out of the array and one for adding the two float32x4 values. The compiler puts `sum` variable in the xmm2 register for the entire loop execution!

VI. Methodology

This section describes our hardware and software, measurements, and workload configurations. All of our code, workloads, and performance measurement methodologies are publicly available.

A. Virtual Machines and Measurements

We use the M37, branch 3.27.34.6, version of V8 and version JavaScript-C34.0a1 of SpiderMonkey for these experiments.

B. Hardware Platforms

We measure the performance and energy of SIMD language extension on multiple different architectures and operating system combinations. Table I lists characteristics of our experimental platforms. We report results on x86 hardware. Among the hardware systems we measure are an in-order Atom processor, a recent 22 nm low-power dual-core Haswell processor (i5-4200U) and a high-performance quad-core counterpart (i7-4770K).

C. Benchmarks

To evaluate our language extensions, we developed a set of benchmarks from various popular JavaScript application domains including 3D graphic code, cryptography, arithmetic, higher order mathematical operations, and visualization. Table II lists their names and the number of lines of code in the original, the SIMD version, and the number of

Processor	Architecture	Frequency	Operating System
i5-4200U	Haswell	1.60 GHz	Ubuntu Linux 12.04.4x64
i7-3720QM	Sandy Bridge	2.60 GHz	Mac OS X 10.9.4
i5-2520M	Sandy Bridge	2.50 GHz	Windows 7 64-bit
i7-4770K	Haswell	3.50 GHz	Ubuntu Linux 12.04.4
Atom Z2580	Cloverview	2.00 GHz	Android 4.2.2

Table I: Experimental platforms

SIMD operations. For this initial SIMD benchmark suite, we select benchmarks that reflect operations typical of SIMD programming in other languages such as C++, and that are sufficiently self-contained to allow JavaScript VM implementers to use them as a guide for testing the correctness and performance of their system.

Although JavaScript only supports double-precision numeric types, we take advantage of recent optimizations in JavaScript JIT compilers that optimize the scalar code to use single-precision instructions when using variables that are obtained from Float32x4Arrays. All of our scalar benchmarks, with the exception of Average64x2, perform float32 operations (single precision). The scalar codes thus have the advantage of single precision data sizes and optimizations, which makes the comparison to their vector counterparts an apples-to-apples comparison, where the only change is the addition of SIMD vector instructions.

3D graphics: As noted in Section III, float32x4 operations are particularly useful for graphics code. Because most of the compute intensive work on the CPU side (versus the GPU) involves computing projection and views of matrices that feed into WebGL, we collected the most common 4x4 matrix operations and a vertex transformation of a 4 element vector for four of our benchmarks:

MatrixMultiplication	4x4 Matrix Multiplication
Transpose4x4	4x4 Matrix Transpose
Matrix4x4Inverse	4x4 Matrix Inversion
VertexTransform	4 element vector transform

Cryptography: While cryptography is not a common domain for SIMD, we find that the hottest function in Rijnadel cipher should benefit from SIMD instructions. We extracted this function into the following kernel.

ShiftRows	Rotation of row values in 4x4 matrix

Higher Level Math Operations: Mathematical operations such as trigonometric functions, logarithm, exponential, and power, typically involve complicated use of SIMD instructions. We hand-coded a representative implementation of the sinx4() function. We believe such operations will become important in emerging JavaScript applications that implement physics engines and shading. For example, the AOBench shading (Ambient Occlusion benchmark) benefits from 4-wide cosine and sine functions.

Sine	Four element vector sine function.

	LOC		SIMD
Benchmark	**Scalar**	**SIMD**	**calls**
Transpose4x4	17	26	8
Matrix4x4Inverse	83	122	86
VertexTransform	26	12	13
MatrixMultiplication	54	41	45
ShiftRows	12	18	3
AverageFloat32x4	9	9	2
AverageFloat64x2	9	9	2
Sinex4†	14	5	1
Mandelbrot	25	36	13
Aobench	120	201	119
Skinning	77	90	66

Table II: Benchmark characteristics. We measure the lines of code (LOC) for the kernel of each benchmark in both scalar and SIMD variants. †In the case of Sinex4, the table reports the LOC for the simple sine kernel, which makes calls to the sine function in the Math libary and the equivalent SIMD implementation respectively. The full first-principles implementation of SIMD sine takes 113 LOC and makes 74 SIMD calls.

Math, Mandelbrot, and more graphics: In addition, we modified the following JavaScript codes to use SIMD optimizations.

Average32x4	Basic math arithmetic (addition) on arrays of float32 items.
Mandelbrot	Visualization of the calculation of the Mandelbrot set. It has a static number of iterations per pixel.
AOBench	Ambient Occlusion Renderer. Calculates how exposed each point in a scene is to ambient lighting.
Skinning	Graphics kernel from a game engine to attach a renderable skin to an underlying articulated skeleton.

Developers are porting many other domains to JavaScript and they are likely to benefit from SIMD operations, for example, physics engines; 2D graphics, e.g., filters and rendering; computational fluid dynamics; audio and video processing; and finance, e.g., Black-Scholes.

D. Measurement Methodology

We first measure each non-SIMD version of each benchmark and configure the number of iterations such that it executes for about 1 second in steady state. This step ensures the code is hot and the JIT compilers will be invoked on it. We measure the SIMD and non-SIMD benchmark configurations executing multiple iterations 10 times. We invoke each JavaScript VM on the benchmark using their command line JavaScript shells. Our benchmark harness wraps each benchmark, measuring the time and energy using performance counters. This methodology results in statistically significant results comparing SIMD to non-SIMD results.

Time: We measure the execution time using the low overhead real time clock. We perform twenty measurements, interleaving SIMD and scalar systems, and report the mean.

Energy: We use the Running Average Limit Power (RAPL) Machine Specific Registers (MSRs) [19] to obtain the energy measurements for the JavaScript Virtual Machine running the benchmark. We perform event-based sampling through CPU performance counters. We sample PACKAGE_ENERGY_STATUS which is the energy consumed by the entire package, which for the single-die packages we use, means the entire die. Two platforms, the Android Atom and the Windows Sandy Bridge, do not support RAPL and thus we report energy results only for the other three systems.

The performance measurement overheads are very low (less than 2% for both time and energy). We execute both version of the benchmarks using the above iteration counts.

VII. Results

This section reports our evaluation of the impact of SIMD extensions on time and energy.

A. Time

Figure 5 shows the speedup due to the SIMD extensions. The graphs show scalar time divided by SIMD time, so any value higher than one reflects a speedup due to SIMD. All benchmarks show substantial speedups and unsurprisingly, the micro benchmarks (left five) see greater improvement than the more complex kernels (right five).

It may seem surprising that many of the benchmarks improve by more than 4×, yet our SIMD vectors are only four-wide. Indeed, the matrix shift rows benchmark improves by as much as 7× over the scalar version. This super-linear speed up is due to the use of our SIMD operations in an optimized manner that changes the algorithm. For example, the code below shows the SIMD and nonSIMD implementations of the shift row hot methods. Note how we eliminate the need to have temporary variables because we do the shifting of the rows by using SIMD swizzle operations. Eliminating temporary variables and intermediate operations deliver the super-linear speed ups. In summary, the kernels improved by 2× to 9× due to the SIMD extension.

```
// Typical implementation of the shiftRows function
function shiftRows(state, Nc) {
  for (var r = 1; r < 4; ++r) {
    var ri = r*Nc; // get the starting index of row 'r'
    var c;
    for (c = 0; c < Nc; ++c)
        temp[c] = state[ri + ((c + r) % Nc)];
    for (c = 0; c < Nc; ++c)
        state[ri + c] = temp[c];
} }
// The SIMD optimized version of the shiftRows function
// Function special cased for 4 column setting (Nc==4)
// This is the value used for AES blocks
function simdShiftRows(state, Nc) {
  if (Nc !== 4) {
    shiftRows(state, Nc);
  }
  for (var r = 1; r < 4; ++r) {
    var rx4 = SIMD.Int32x4.load(state, r << 2);
    if (r == 1) {
      SIMD.Int32x4.store(state, 4,
            SIMD.Int32x4.swizzle(rx4, 1, 2, 3, 0));
    } else if (r == 2) {
      SIMD.Int32x4.store(state, 8,
            SIMD.Int32x4.swizzle(rx4, 2, 3, 0, 1));
    } else { // r == 3
      SIMD.Int32x4.store(state, 12,
            SIMD.Int32x4.swizzle(rx4, 3, 0, 1, 2));
} } }
```

Of course the impact of the SIMD instructions is dampened in the richer workloads for which SIMD instructions are only one part of the instruction mix. Nonetheless, it is encouraging to see that the skinning benchmark, which is based on an important real-world commercial JavaScript workload, enjoys a 1.8× performance improvement due to the addition of SIMD instructions.

B. Energy

Figure 6 shows the energy improvement due to the SIMD extensions. The graph shows scalar energy divided by SIMD energy using the hardware performance counters. Any value higher than one reflects an energy improvement due to SIMD. The results are consistent with those in Figure 5, with the improvements dampened slightly. The dampening is a result of measurement methodology. Whereas performance is measured on one CPU, package energy is measured for the entire chip. The energy draw is affected both by 'uncore' demands such as the last level cache and memory controller, as well other elements of the core such as L1 and L2 caches, the branch predictor, etc., each of which are not be directly affected by the SIMD extensions.

Nonetheless, the energy improvements are substantial. For the real-world skinning workload the improvements are between 25% and 55%, which is significant, particularly in the power-sensitive context of a mobile device.

VIII. Future Work Discussion

Our design goal of portability is intended to be consistent with the existing JavaScript language specification. However, this constraint precludes platform-specific optimizations which are not currently accessible from JS that would benefit performance and energy efficiency.

First, the opportunity to use wider vector lengths, such as the AVX 512 vector instruction set that Intel is already shipping, will deliver additional performance improvements, particularly on the server side. Stream processors (in the form of an API) can and will be built in software on top of the current SIMD.js specification to utilize this hardware.

Second, a large number of operations are currently not supported, including platform-specific ones. One approach is to provide an extended API for SIMD.js that accesses platform-specific instructions and optimizations. This API would sit on top of and complement the base API described in this paper. The extension API could offer opportunities for performance tuning, specialized code sequences, and support porting of code from other platforms.

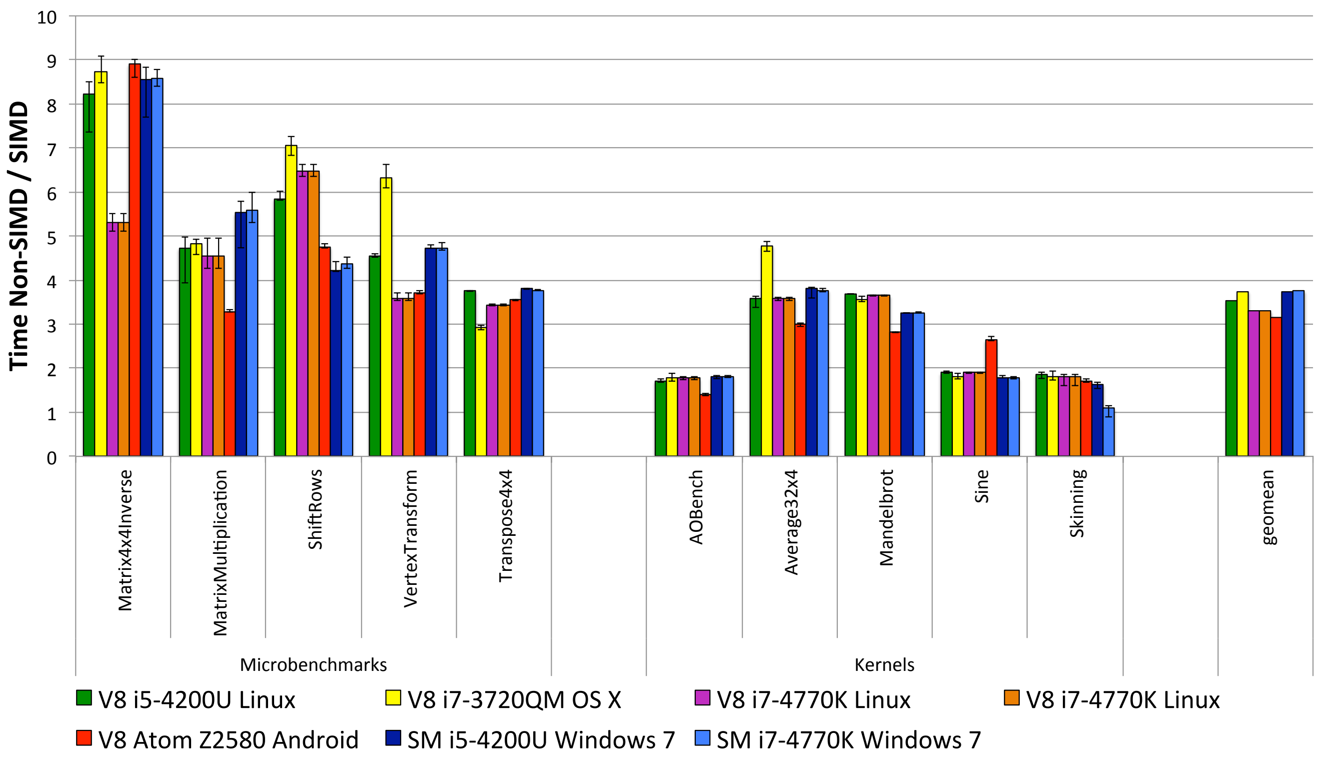

Figure 5: SIMD performance with V8 and SpiderMonkey (SM). Normalized to scalar versions. Higher is better.

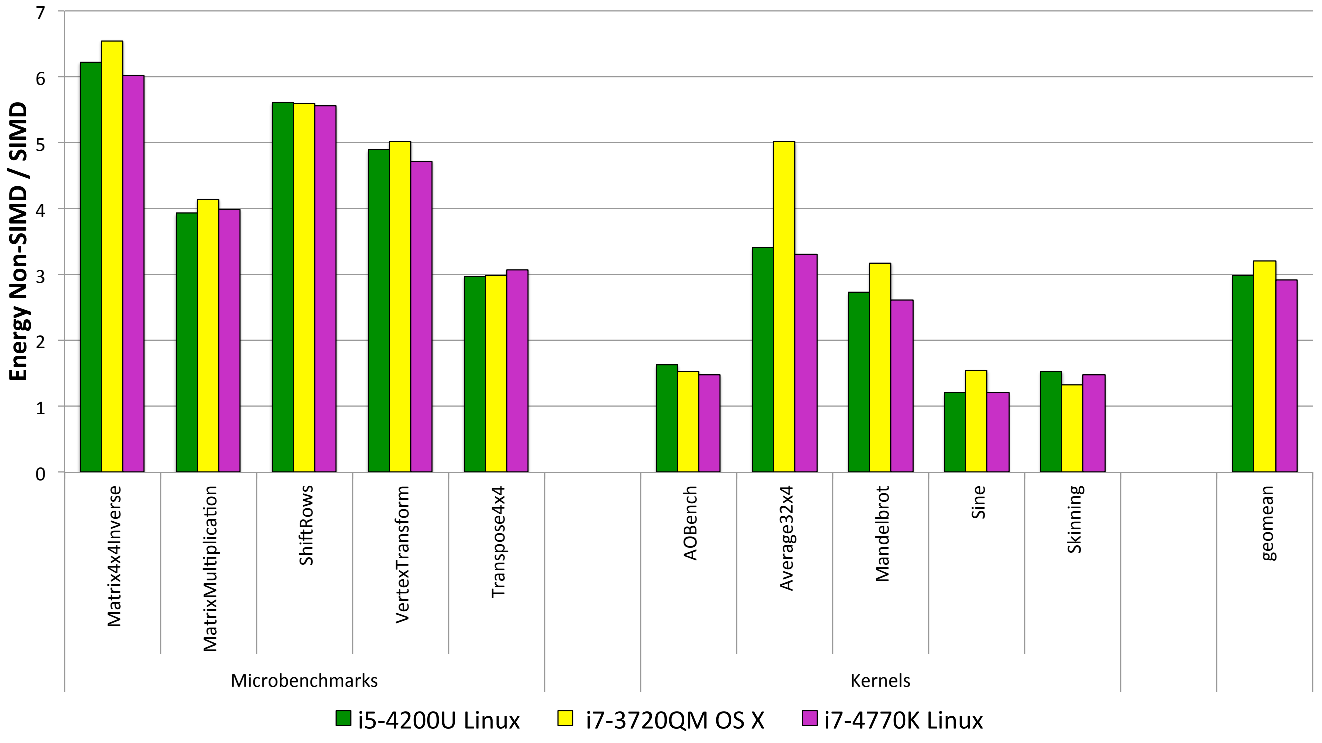

Figure 6: SIMD energy on select platforms with V8. Normalized to the scalar versions. Higher is better.

One can classify the remaining SIMD operations in two groups: those that are portable but have semantic differences (SIMD.Relaxed) and those that are only available on some platforms (SIMD.Universe). Functions in SIMD.Relaxed would mimic functions in the base API with corresponding names, and provide weaker portability with greater potential for performance (e.g., with unspecified results if NaN were to appear in any part of the computation, treating zero as interchangeable with negative zero, and unspecified results if an overflow occurs). Functions in the SIMD.Universe namespace could adhere to well defined semantics but their availability would result in various code paths depending on the architecture. For example, SIMD.isFast would need to check whether the JIT compiler can generate a fast code sequence from each operation for the current hardware.

IX. Conclusion

This paper describes the design and implementation of a portable SIMD language extension for JavaScript. This specification is in the final stages of adoption by the JavaScript standards committee and our implementations are available in the V8 and SpiderMonkey open-source JavaScript VMs. The contributions of this paper include a principled design philosophy, a fixed-size SIMD vector language specification, a type speculation optimization, and evaluation on critical kernels. We describe the language extension, its implementation in two current JavaScript JIT compilers, and evaluate the impact in both time and energy. Programability, portability, ease of implementation, and popular use-cases all influenced the design decision to choose a fixed-width 128-bit vector. Our evaluation demonstrates that the SIMD extensions deliver substantial improvements in time and energy for vector workloads. Our design and implementation choices do not preclude adding more SIMD operations in the future, high-level JavaScript libraries to implement larger vector sizes, or adding automatic vectorizing compiler support. Another potential avenue for future work is agressive machine-specific JIT optimizations to utilize wider vectors when available in the underling hardware. Our results indicate that these avenues would likely be fruitful.

X. Acknowledgments

We thank our colleagues from ARM, Google, Intel, Microsoft, and Mozilla that have contributed to the SIMD.js standardization process, the draft specification and documentation for ECMAScript standard, the ongoing implementation effort in production systems, and the members of the ECMAScript standardization committee (Ecma TC39) for their valuable feedback. This work was supported in part by a generous research gift from Intel Corporation to the University of Texas at Austin and NSF grant SHF-0910818.

References

[1] O. Agesen and U. Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *ACM Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–107, 1995.

[2] ARM. NEON and VFP intel (SSE4) programming reference. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Chdehgeh.html, retrieved 2014.

[3] W. Bouknight, S. Denenberg, D. McIntyre, J. Randell, A. Sameh, and D. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[4] CRAY Research, Inc. The CRAY-I Computer System, 1976.

[5] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programmming (ECOOP)*, pages 77–101, 1995.

[6] A. Gal, B. Eich, M. Shaver, D. Anderson, B. K. G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. R. Haghighat, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI 2009)*, pages 465–478, 2009.

[7] Google. V8 JavaScript Engine, retrieved 2014. http://code.google.com/p/v8.

[8] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, 1994.

[9] Intel. Pentium Processor with MMX Technology. http://download.intel.com/support/processors/pentiummmx/sb/24318504.pdf, 1997.

[10] Intel. Intel (SSE4) programming reference, 2007.

[11] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved type specialization for dynamic scripting languages. In *Dynamic Languages Symposium (DLS)*, pages 37–48, 2013.

[12] M. N. Kedlaya, B. Robatmili, C. Caşcaval, and B. Hardekopf. Deoptimization for dynamic language jits on typed, stack-based virtual machines. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 103–114, 2014.

[13] X. Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT*, ACM Symposium on Principles of Programming Languages (POPL), pages 177–188, 1992.

[14] J. McCutchan, H. Feng, N. D. Matsakis, Z. Anderson, and P. Jensen. A SIMD programming model for Dart, JavaScript, and other dynamically typed scripting languages. In *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2014.

[15] Microsoft. The JIT finally proposed. JIT and SIMD are getting married. http://blogs.msdn.com/b/dotnet/archive/2014/04/07/the-jit-finally-proposed-jit-and-simd-are-getting-married.aspx, 2014.

[16] Mono Project. Mono open-source .NET implementation for C#. http://docs.go-mono.com/?link=N%3aMono.Simd, 2007.

[17] Mozilla Corporation. Emscripten. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten, 2012.

[18] Mozilla Foundation. SpiderMonkey, retrieved 2014. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[19] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, Mar. 2012. ISSN 0272-1732. doi: 10.1109/MM.2012.12. URL http://dx.doi.org/10.1109/MM.2012.12.

[20] TC39 - ECMAScript. SIMD.js specification v0.8.4, retrieved 2015. http://tc39.github.io/ecmascript_simd/.

[21] WHATWG. HTML living standard. http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html, retrieved 2015.