Vectorization Technology To Improve Interpreter Performance

ERVEN ROHOU, KEVIN WILLIAMS, and DAVID YUSTE, Inria Rennes – Bretagne Atlantique

In the present computing landscape, interpreters are in use in a wide range of systems. Recent trends in consumer electronics have created a new category of portable, lightweight software applications. Typically, these applications have fast development cycles and short life spans. They run on a wide range of systems and are deployed in a target independent bytecode format over Internet and cellular networks. Their authors are untrusted third-party vendors, and they are executed in secure managed runtimes or virtual machines. Furthermore, due to security policies or development time constraints, these virtual machines often lack just-in-time compilers and rely on interpreted execution. At the other end of the spectrum, interpreters are also a reality in the field of high-performance computations because of the flexibility they provide.

The main performance penalty in interpreters arises from instruction dispatch. Each bytecode requires a minimum number of machine instructions to be executed. In this work, we introduce a novel approach for interpreter optimization that reduces instruction dispatch thanks to vectorization technology. We extend the split compilation paradigm to interpreters, thus guaranteeing that our approach exhibits almost no overhead at runtime. We take advantage of the vast research in vectorization and its presence in modern compilers. Complex analyses are performed ahead of time, and their results are conveyed to the executable bytecode. At runtime, the interpreter retrieves this additional information to build the SIMD IR (intermediate representation) instructions that carry the vector semantics. The bytecode language remains unmodified, making this representation compatible with legacy interpreters and previously proposed JIT compilers.

We show that this approach drastically reduces the number of instructions to interpret and decreases execution time of vectorizable applications. Moreover, we map SIMD IR instructions to hardware SIMD instructions when available, with a substantial additional improvement. Finally, we finely analyze the impact of our extension on the behavior of the caches and branch predictors.

Categories and Subject Descriptors: D.3.4.4 [**Interpreters**]

General Terms: Performance

Additional Key Words and Phrases: Interpreters, SIMD, bytecode dispatch

ACM Reference Format:
Rohou, E., Williams, K., and Yuste, D. 2013. Vectorization technology to improve interpreter performance ACM Trans. Architec. Code Optim. 9, 4, Article 26 (January 2013), 22 pages.
DOI = 10.1145/2400682.2400685 http://doi.acm.org/10.1145/2400682.2400685

1. INTRODUCTION

In the present computing landscape, interpreters are in use in a wide range of systems. Recent advances in portable consumer electronics have created a new category of computing applications. These applications are lightweight, mobile, and have reduced development costs and time. It results in a culture of very fast deployment times and often short application life-times. These applications run on many devices from portable electronics to home entertainment systems. They are deployed over networks, to many different hardware targets. Their authors are typically untrusted third-party

Authors' addresses: E. Rohou, K. Williams, and D. Yuste (corresponding author), Inria Rennes – Bretagne Atlantique; email: david.yuste@gmail.com.

DOI 10.1145/2400682.2400685 http://doi.acm.org/10.1145/2400682.2400685

vendors. For these reasons, mobile executables must be distributed in a secure, target-independent format, namely bytecode. These bytecodes run in secure managed environments called virtual machines. Virtual machines ensure the integrity of the system and control the access of the deployed application to privileged resources. Traditional virtual machines for languages such as Java and .NET use interpreters for fast startup time and augment them with JIT (Just-In-Time) compilers for increased performance [Paleczny et al. 2001; Oracle 2010]. Developing a JIT compiler, however, is much more complex and time consuming than an interpreter, and vendors may decide to content themselves with interpretation. Some vendors of modern consumer electronics require code to be digitally signed and secured before deployment, and their policies prohibit dynamic code generation on their targets [Xamarin 2011], including JIT compilers.

At the other end of the spectrum, interpreters are also in use in fields where high-performance computations are necessary, such as physics. The added flexibility and the faster development cycles favor such environments. Scientists from both CERN and Fermilab report [Naumann and Canal 2008] that “many of LHC experiments’ algorithms are both designed and used in interpreters”. As another example, the need for an interpreter is also one of the three reasons motivating the choice of Jython for the data analysis software of the Herschel Space Observatory [Wieprecht et al. 2004].

Finally, many domain specialists turn to languages such as Python, Ruby, or Matlab simply because they are easier and faster to learn[1]. These languages happen to be mostly interpreted.

Unfortunately, the performance of interpreters is lower than what static and JIT compilers achieve. The causes are inherent to the nature of interpreters. First, a program is translated into an IR used internally by the interpreter. At this point, only fast optimizations can be applied. Then, the IR is “executed”: each instruction is dispatched, and the code chunk matching its semantics is executed. Instruction dispatch is an important source of inefficiency due to the overhead introduced. Our experiments with our own interpreter for CLI on x86 show that, on average, the execution of a single bytecode takes from 20 to 30 machine instructions. They account for loading a bytecode from memory, decoding it, and transferring the flow of control to the code that performs the appropriate function. Parameters, if any, are then read from the evaluation stack, the computation is performed, and the result is written back.

A common way for improving performance of interpreters is to reduce the number of instructions to dispatch. To this end, common sequences of instructions are gathered into single superinstructions [Ertl and Gregg 2003, 2004]. The interpreter identifies these repeated patterns and replaces them with the corresponding superinstruction. However, the space of search for superinstructions is limited due to time constraints. Aggressive analysis of the code and powerful transformations are out of their reach.

The *spirit* of our approach is similar to superinstructions, in the sense that we improve the performance of the interpreter by reducing the number of dispatches. The major differences are as follows. First, the patterns are much coarser grain than traditional superinstructions: they can encompass up to hundreds of dynamic bytecodes. Second, they are not computed by the interpreter, but generated by an offline compiler, and exposed to the interpreter as candidate superinstructions, thanks to predefined builtins. We preserve backward compatibility with legacy interpreters, while enabling very significant speedups on systems based on our proposal.

[1]See for example http://www.ohloh.net/languages/compare for trends on programming languages usage for over half a million open source projects.

To this end, we use a split compilation strategy to generate new IR instructions based on SIMD technology. The amount of instructions to dispatch is strongly reduced thanks to automatic vectorization of the code. The transformation is performed during the offline compilation (source code to bytecode), without penalty in the interpreter side. The program is speculatively vectorized thanks to the techniques developed in previous work [Nuzman et al. 2011] and SIMD builtins are embedded in the bytecode binary. We modified our interpreter to make it aware of the new vector semantics. Operations over vectors identified by the offline compiler become SIMD IR instructions at almost no penalty.

In Section 2, we first summarize previous research on vectorized bytecode for JIT compilers, on which we base this work. Section 3 then motivates the current work, and presents, at high level, how an interpreter takes advantage of the vectorization opportunities exposed in the bytecode. We develop the implementation details in Section 4, in particular we introduce our target virtual machine, and we describe the changes performed in order to benefit from the proposed extensions. Our experimental framework as well as the results are detailed in Section 5. We review related work in Section 6.

2. BACKGROUND

This section describes in detail prior art [Rohou et al. 2011; Nuzman et al. 2011] related to split vectorization. With our co-authors, we have been previously studying how vectorization can benefit JIT compilers, despite the cost of the transformation: we proposed a split compilation approach in which the bytecode is speculatively vectorized by an offline compiler. At runtime, a JIT compiler only has to materialize a few parameters and instantiate code templates to produce highly optimized native code, at little cost.

The current work focuses only on *interpreters* (as opposed to JIT compilers). We consider the *same* speculatively vectorized bytecode as produced by the mentioned approach, and we investigate how it can be efficiently *interpreted*. We show that vectorization is a powerful tool to improve the performance of interpreters as well.

By reusing the framework unmodified, we guarantee compatibility with the previous approach, in particular the deployed bytecode would benefit for the same acceleration, should a JIT compiler be present on the target system. We aim here at making the expected acceleration possible on an interpreter-based virtual machine.

While based on the same offline compilation technology, the approach is very different because interpreters do not have any dynamic code generation capability, which was key to materializing the vectors and delivering performance in the JIT compilers.

For the sake of clarity, we dedicate this section to a brief description of vectorization and a summary of the main results of our previous research: split vectorization and the production of speculatively vectorized bytecode by an offline compiler.

2.1. Vectorization

Vectorization is compiler optimization that transforms loops. It detects patterns where an instruction is repeatedly applied to different data. Autovectorization consists in automatically rewriting the loop so that a Single Instruction applies to Multiple Data (SIMD).

In the search for instructions comprising many operations in a single dispatch, SIMD instruction sets are good candidates. Their semantics are well understood and widespread. Vectorization techniques have been studied for decades, resulting in a mature and well known field. Since processors started to extend their instruction sets with SIMD capabilities, automatic vectorization has become a common feature of compilers.

In a static compiler targeting native executables, automatic vectorization starts by analyzing loop dependences and selecting vectorizable loops, always taking in consideration a cost model based on the target. Then, candidate loops are strip-mined by a factor of the vector length. Finally, scalar instructions within the loop body are replaced with the corresponding SIMD instructions available in the target architecture.

2.2. Split Vectorization for JIT Compilers

However, vectorization algorithms require aggressive, time-consuming loop analysis and transformations (data dependence analysis, which is quadratic in the number of memory references, control flow analysis, loop normalization, construction of SSA form, etc.) This complexity prevents them from being deployed as part of JIT compilers or interpreters, or in very limited forms. Still, vectorization can be performed ahead of time. In this scenario, expensive transformations are moved up to the offline compilation step, and their results are encoded into the deployed executable in bytecode format. The development flow of interpreted software fits in this scheme.

The difficulty of targeting a bytecode is that no information about the target machine is known. And the performance of vectorizers has traditionally been very dependent on the detailed description of the instruction set. We previously showed [Nuzman et al. 2011] how loops can still be speculatively vectorized. Vector length, actual target SIMD instructions, and other target-specific parameters are abstracted away. Actual values of these parameters are instantiated by the final JIT compiler at runtime.

Let us take as running example the small loop of Figure 1(a). An offline compiler without autovectorization produces code similar to the pseudo-bytecode of Figure 1(b). In order to reduce the amount of instructions, the loop is speculatively vectorized following the transformation proposed before. Figure 2(a) illustrates the outcome in pseudo-bytecode. The iterator variable i is increased with the value *vec_size* (lines 31–34). Thus, the total number of instructions to dispatch is reduced by a factor of *vec_size*. This value, as well as every vector operation, is abstracted away, and is materialized only on the target.

The abstraction of target-specific parameters takes the form of builtins. Builtins are a standard way used by compilers to carry semantic information down the compilation flow. They consist of function calls whose names have a special semantics. Looking at the example of Figure 2(a), the builtin *get_vec_size* (line 1) provides the vector size in the target machine or interpreter. Similarly, *load_vector*, *store_vector* (lines 20, 25, and 29) abstract the operation of loading a vector whose size and layout are unknown at offline compilation time. The arithmetic of the example relies in the *op_Multiply* (line 27) that implements the product of two vectors.

2.3. Offline Compilation Framework

The chosen bytecode is CLI (Common Language Infrastructure). CLI is a standardized [ECMA International 2006] framework that defines a bytecode format and a runtime environment suitable for the deployment of portable applications. It forms the core of the Microsoft .NET framework as well as open-source implementations [Mono 2012; Campanoni et al. 2010]. It has also been adopted for the development of rich Internet applications by the Silverlight [Papa and Kinney 2010] and Moonlight projects [Moonlight 2009].

Gcc4cli [Costa et al. 2007] is used to compile C source-code into CLI bytecode. Gcc4cli is a port of the GCC compiler that targets the CLI format. GCC features a multiplatform autovectorizer [Nuzman and Henderson 2006; Nuzman and Zaks 2006] that can perform inner-loop, outer-loop, intra-loop, and straight-line code vectorization. Gcc4cli has been recently adapted [Nuzman et al. 2011] to generate CLI bytecode with builtins facilitating the adaptation to various instruction sets.

There are no standard defining vector extensions in the CLI format. The Mono project proposed an extension in the form of the `Mono.Simd` library [de Icaza 2008]

```
for(int i=0; i<N; i++)
{
  sz[i] = sx[i] * sy[i];
}
```

(a) Source code of target loop

```
          ldc    0
          stloc  i

loop:
          ldloc  i
          ldc    4          // sizeof(float)
          mul
          stloc  offset

          ldloc  sz         // compute @sz[i]
          ldloc  offset
          add

          ldloc  sx         // compute @sx[i]
          ldloc  offset
          add
          ldind             // load sx[i]

          ldloc  sy         // compute @sy[i]
          ldloc  offset
          add
          ldind             // load sy[i]

          mul

          stind             // store to sz[i]

          ldc    1          // i++
          ldloc  i
          add
          stloc  i

          ldloc  i          // loop exit?
          ldc    N
          blt    loop
```

(b) Scalar CLI bytecode generated by gcc4cli (simplified for readability)

Fig. 1. Running example: simple vector product.

to let C# programmers exploit SIMD instruction set extensions. The library defines vector types, and operators on these types. The semantics of the operators is entirely defined in the library with scalar code, and code relying on this extension can run on any CLI-compliant virtual machine.

The `Mono.Simd` library is currently biased towards the x86 architecture. The proposed extensions [Nuzman et al. 2011] provide the abstraction of the actual vector size, a wider set of vectorization idioms appropriate to various instruction sets, as well as the realignment mechanisms.

```
        call   veclib.VSF::get_vec_size
        stloc  vec_size

        ldc    0
        stloc  i

loop:
        ldloc  i
        ldc    4           // sizeof(float)
        mul
        stloc  offset

        ldloc  sz          // compute @sz[i]
        ldloc  offset
        add

        ldloc  sx          // compute @sx[i]
        ldloc  offset
        add
        call   veclib.VSF::load_vector

        ldloc  sy          // compute @sy[i]
        ldloc  offset
        add
        call   veclib.VSF::load_vector

        call   veclib.VSF::op_Multiply

        call   veclib.VSF::store_vector

        ldloc  vec_size   // i += vec_size
        ldloc  i
        add
        stloc  i

        ldloc  i           // loop exit?
        ldc    N
        blt    loop
```

(a) vectorized pseudo-bytecode generated by gcc4cli, input of interpreter

```
int veclib.VSF::get_vec_size (void) {
  return 4;
}

Vector4f veclib.VSF::load_vector (Vector4f* addr) {
  return *addr;
}

void veclib.VSF::store_vector (Vector4f* addr, Vector4f v) {
  *addr = v;
}

Vector4f veclib::op_Multiply (Vector4f a, Vector4f b) {
  Vector4f c = { a.v0 * b.v0, a.v1 * b.v1, a.v2 * b.v2, a.v3 * b.v3 };
  return c;
}
```

(b) implementation of the builtins in the support library

Fig. 2. Running example: vectorized bytecode.

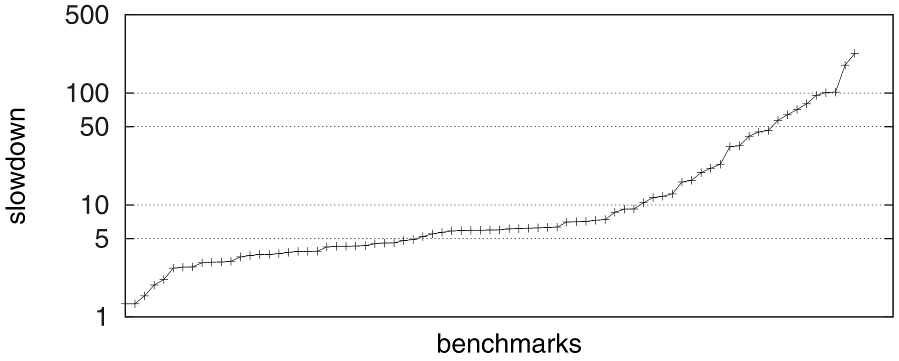

Fig. 3. Penalty incurred by the legacy interpreter on 78 kernels. Each point represents a kernel. Kernels are ordered by slowdown for readability.

As result of the offline compilation, a CLI-compliant binary is obtained. Vector semantics are embedded in the form of builtins. The contribution of this work is to make an interpreter aware of these builtins in order to increase performance.

3. VECTORIZATION AND INTERPRETERS

A requirement of this work is to guarantee functional compatibility with legacy interpreters, unaware of the semantics of the SIMD builtins, in order to provide a smooth path to developers in charge of supporting the new SIMD features. As in our previous work [Nuzman et al. 2011], a companion library with the implementation of each builtin is deployed in conjunction with the vectorized program. Figure 2(b) shows the library code corresponding to the builtins of our example specialized on 4-float vectors. This library is ignored by interpreters aware of the special semantics of the builtins.

The main goal is to deliver performance, and to show that split vectorization benefits interpreters as well. The efficiency of our proposal relies in the capability of the interpreter to deal with the new builtins. In a JIT compiler, the key feature that delivers performance from vector builtins is the dynamic code generation. Optimized code sequences are produced for each builtin. Vector sizes and alignment constraints are materialized, and standard optimizations (such as constant propagation, dead code elimination, function inlining) remove inefficiencies.

Interpreters lack this code generation capability. In order to quantify the performance penalty, and to motivate our work, we ran some introductory experiments. We executed[2] both a vectorized version and a scalar version of each kernel presented in Section 5.1, using a legacy interpreter unaware of vector builtins. The calls to the companion library, and the additional bytecodes to dispatch, offset the benefits of the vectorization. The results are plotted on Figure 3 as follows: for each of the 78 kernels, we computed the slowdown, and we ordered them in increasing order for readability (this graph is sometimes referred to as an S-curve). The results show that a legacy interpreter executing the builtins as library calls may experience slowdowns ranging from 30% up to 225× compared with the interpretation of the scalar version of the program. This is consistent with the dramatic increase of the number of dispatched instructions as consequence of the library calls, whose bytecodes must be interpreted as any other in the program.

[2]The experiments were performed on an Intel Core 2 Duo E6850 at 3 GHz, with 4 MB of cache memory. The interpreter used in these experiments is an unmodified version of PVM, described in Section 4.

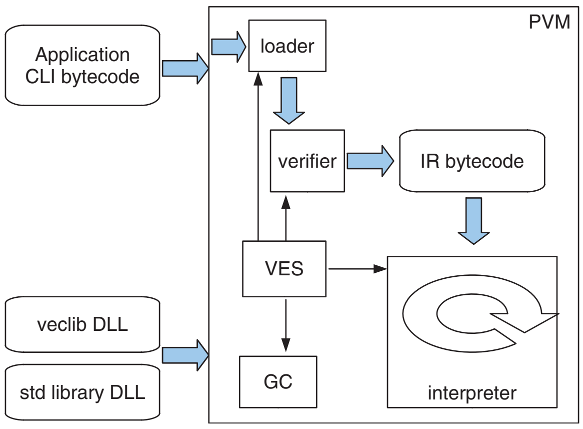

Fig. 4. Structure of PVM.

Incidentally, the offline compiler abstracts vector size but also target realignment capabilities. This was key to delivering performance with a JIT compiler [Nuzman et al. 2011]. The bytecode contains all the information needed by a JIT compiler to produce efficient native code. The realignment mechanism is not necessary for the interpreter and constitutes an overhead. However, for compatibility reasons as well as fairness, we left the offline compiler unmodified (including realignment), and we added appropriate IR instructions, and library support.

The goal of this work is to address how an interpreter can be made aware of the SIMD builtins to overcome the overhead and deliver higher peformance. The idea is to replace the builtins with the corresponding IR instructions. Under these conditions, the number of dispatched bytecodes is effectively reduced by a factor of the vector size, resulting in significant speedups. Moreover, extra benefits are obtained when the interpretation of the new IR instructions takes advantage of the host native SIMD features. The next section details the modifications needed in the interpreter in order to reach the announced improvements.

4. IMPLEMENTATION DETAILS

The performance improvement of the interpreter requires two key components: (1) the offline compilation of the source program, which generates vectorized bytecode; and (2) the online interpretation that leverages the vector builtins to build superinstructions. As discussed in Section 2, the former reuses unmodified previous work in autovectorization [Nuzman et al. 2011; Rohou et al. 2011] in the context of portable bytecodes. The latter is the contribution of this article: it consists in extensions of a virtual machine and interpreter to significantly improve the efficiency of the interpreter.

We base our work on PVM (Portable Virtual Machine), a proprietary virtual machine that executes applications written in the CLI format. Figure 4 illustrates its general architecture. PVM is complete with loader, bytecode verifier, internal representation, interpreter, garbage collector, and runtime support.

The operations of the virtual machine are under the control of the VES (Virtual Execution System). When an application is to be executed, the VES first requests the loader to locate the executable and to put it in memory. The verifier is then invoked to check the legality of the bytecode, as mandated by the CLI standard [ECMA International 2006]. In particular, it checks that local variables are only assigned values of the proper type, and that no stack overflow or underflow can occur. While processing the CLI input stream, it emits an internal representation (IR bytecode) that is used by

Table I. SIMD IR Instructions Summary

Arithmetic	*Constructors*
IR_VEC_ADD	IR_VEC_INIT
IR_VEC_SUB	IR_VEC_DUP
IR_VEC_MUL	IR_VEC_UNIFORM
IR_VEC_DIV	IR_VEC_AFFINE
IR_VEC_DOT_PRODUCT	IR_VEC_INT2FLOAT
IR_VEC_REDUCE_PLUS	IR_VEC_FLOAT2INT
IR_VEC_WIDEN_MULT_HI	
IR_VEC_WIDEN_MULT_LO	

Data manipulation	*Memory management*
IR_VEC_INTERLEAVE_HIGH	IR_VEC_LDARG
IR_VEC_INTERLEAVE_LOW	IR_VEC_LDLOC
IR_VEC_PACK	IR_VEC_LDOBJ
IR_VEC_UNPACK_HIGH	IR_VEC_ALOAD
IR_VEC_UNPACK_LOW	IR_VEC_LOADALIGNED
IR_VEC_SHR	IR_VEC_REALIGNLOAD
IR_VEC_BIT_FIELD_REF	IR_VEC_MASK4LOAD
IR_VEC_GET_X	IR_VEC_STLOC
IR_VEC_GET_Y	IR_VEC_STOREALIGNED
	IR_VEC_REALIGNOFFSET

the rest of virtual machine. IR resembles CLI, but it is more efficiently processed by the interpreter. For example, polymorphic CLI operators like `add` are replaced by typed operators like `add.i4` (32-bit integer addition) or `add.r8` (double precision floating point).

Frequent bytecode patterns are combined into superinstructions [Ertl and Gregg 2003], in order to reduce the number of dispatched bytecodes and improve performance. Superinstructions are statically prebuilt in PVM. The unmodified reference version of PVM contains 478 regular IR bytecodes and 68 superinstructions. Superinstructions are made of two or three original instructions.

4.1. Fewer Dispatches thanks to SIMD IR Instructions

At this point, we introduce the first extension of the interpreter. The IR instruction set is extended with 121 new SIMD IR instructions, matching vector operations over different data types, and some primitives required for the target abstraction. They are derived from the work of Nuzman et al. [2011] and summarized in Table I, without distinction among data types. During the construction of the IR representation, the verifier recognizes the calls to the companion library in the input bytecode stream and constructs SIMD IR instructions instead. The result of this transformation is shown in Figure 5, where the CIL bytecode has been translated into the IR representation.

Once the IR for a function is built, control is passed to the interpreter proper. It essentially consists in a infinite loop that contains a large `switch` statement. At each iteration, a bytecode is read, and dispatched to the appropriate `case` entry where the semantics of the bytecode are implemented: input operands are read from the evaluation stack, some computation is performed, and output operands are stored to the stack. For most IR bytecodes, the implementation is entirely contained in the `case`

```
        ldc    16           // vector size defined as 16
        stloc  vec_size

        ldc    0
        stloc  i

    loop:
        mul.loc.const i 4   // i * sizeof(float)
        stloc  offset

        ldloc  sz           // compute @sz[i]
        ldloc  offset
        add

        ldloc  sx           // compute @sx[i]
        ldloc  offset
        add
        ldloc.vec

        ldloc  sy           // compute @sy[i]
        ldloc  offset
        add
        ldloc.vec

        mul.vsf

        stloc.vec

        ldloc  vec_size     // i += vec_size
        ldloc  i
        add
        stldloc i           // store and load

        ldc    N            // loop exit?
        blt    loop
```

Fig. 5. IR representation of running example.

entry, only the more complex ones require calls to functions outside the main loop (for example, function calls or object construction).

Also, we provide the implementation of our SIMD IR instructions. Most of the arithmetic and data manipulation instructions are implemented as loops performing the requested operations over all elements of the input vectors. The improvement relies in the fact that these loops are statically compiled within the interpreter. Interpretation of entire vector idioms is skipped, and the number of dispatched bytecodes is reduced by a factor proportional to the vectorization factor.

Figure 6 shows a simplified version of the interpreter loop. The first `case` entry describes the implementation of a superinstruction in the original interpreter: `mul.loc.const` combines in a single IR instruction the common sequence of instructions `ld.loc` (push a local variable on the stack), `ldc` (push a constant on the stack), and `mul` (multiply the two values at the top of the stack). The second `case` entry (`mul.vsf`) is introduced to support vector semantics. The size of the vectors is determined by the predefined vector size (16 in this example), and the type of each element (`vsf`: single precision floating point). Figure 6(a) corresponds to a scalar implementation of vector instructions. The vector is multiplied within a loop iterating over all elements.

One parameter is left for the interpreter to choose: the size of the vector slices on which it operates. The offline compiler only vectorizes a loop when the iteration

```
while (1) {

  op_code = read_instruction (&arguments);

  switch (op_code) {
    ...
    case IR_MUL_LOC_CONST: {
      int local    = load_local (arguments[0].as_local);
      int constant = arguments[1].as_constant;
      STACK_PUSH (sizeof(int), local * constant);
      break;
    }
    ...
    case IR_VEC_MUL_VSF: {
      float a[4] = (float[4]) STACK_POP (sizeof(char[16]));
      float b[4] = (float[4]) STACK_POP (sizeof(char[16]));
      float c[4];

      for (int i = 0; i < 4; i++)
        c[i] = a[i] * b[i];

      STACK_PUSH (sizeof(char[16]), (char[16])c);
      break;
    }
    ...
  }
}
```

(a) interpreter with a scalar implementation of vector IR (pseudo-C)

```
while (1) {

  op_code = read_instruction (&arguments);

  switch (op_code) {
    ...
    case IR_MUL_LOC_CONST: {
      int local    = load_local (arguments[0].as_local);
      int constant = arguments[1].as_constant;
      STACK_PUSH (sizeof(int), local * constant);
      break;
    }
    ...
    case IR_VEC_MUL_VSF: {
      vsf a[4] = (vsf) STACK_POP (sizeof(char[16]));
      vsf b[4] = (vsf) STACK_POP (sizeof(char[16]));
      vsf c[4];

      c = __builtin_ia32_mulps(a, b); // GCC extension for SSE product

      STACK_PUSH (sizeof(char[16]), (char[16])c);
      break;
    }
    ...
  }
}
```

(b) interpreter with an accelerated implementation of vector IR (pseudo-C)

Fig. 6. Implementation of interpreter (excerpt).

distance is infinite, making it legal to increase the vector size at will. The online phase is free to choose any value[3]. The trade-off is the following: a larger value means a coarser superinstruction and hence a better speedup; the risk is to entirely fall back to scalar code when the actual array is smaller, or to increase the number of remaining iterations to be processed in the epilogue after the vector loop. A smaller value reduces this probability, but also decreases the performance advantage of the superinstruction. In any case, the vector size is a parameter of the interpreter, which can be adjusted by the final user. In this work, we experimented with 16 bytes since it turns out that this is the size of most current hardware vector registers.

Two other approaches were possible.

(1) The offline compiler could pass the actual size of the array when it can be statically determined, for example, by adding a parameter to the *get_vec_size* builtin. The online compiler could process the entire vector at once, effectively maximizing the effect of the superinstruction.

(2) The offline compiler could generate several identical loops that successively iterate on the array with vectors of decreasing sizes. This multilevel approach would process vectors with the largest (and most effective) size, and remaining iterations with vectors of smaller size.

However, these two variants require changes to the offline compiler. This would break the compatibility with previous work, which we did not want to address in this work.

4.2. Accelerated SIMD IR Instructions

On top of reducing the number of dispatches, we also analyzed the impact of providing an implementation of SIMD IR instructions accelerated with the native SIMD instruction set. Many of the SIMD IR instructions have a straightforward translation on typical vector instructions, and many more can be written by combining few of them. We targeted two instruction sets: the various versions of x86's SSE and the ARM NEON.

We implemented this optimization of the *interpreter code* thanks to a GCC GNU extension that provides both vector types and operators. Simple arithmetic is automatically detected by GCC: a simple `a+b` where `a` and `b` are of type `v4sf` (i.e., a vector of four floats) generates a single `addps` SSE machine instruction. More advanced computations, such as dot product, interleaving, shuffling, and so on, require extensions, also provided by GCC. Figure 6(b) illustrates the code of the interpreter when SIMD IR instructions are accelerated (line 19).

When a processor family supports several levels of SIMD extensions, such as x86 with SSE, SSE2, SSE3, SSSE3, and SSE4.x, we used conditional compilation directives to take advantage of the highest available level. For example, the `unpack_high` builtin returns a vector of four 32-bit integers made of the four least significant elements of a vector of eight 16-bit values. It can be efficiently implemented with the SSE4.1 instruction `pmovsxwd` alone. However, the combination of the two SSE2 instructions `punpcklwd` and `psradi` achieves the same result. When no SIMD instructions of a given target help, we fall back to scalar implementation.

The performance gain of this optimization is second order compared to the effect of dispatch reduction as consequence of the SIMD IR instructions. The reason is the following: the percentage of optimized code in the whole interpreter is relatively small in comparison with the overhead of the dispatch mechanism. Only the implementation of IR instructions in the `switch` entries are accelerated, and only when hardware support is available. Still, for loop-intensive applications, the speedup can be significant.

[3]As long as it is a power of 2—an assumption made offline because it greatly simplifies address generation.

Table II. Description of Kernels

Name	Description & vectorization features
2mm	linear algebra kernel (arithmetic, shift, reduction)
alvinn	weight-update for neural-nets training (outer-loop)
atax	linear algebra kernel (arithmetic, reduction)
bicg	linear algebra kernel (arithmetic, reduction)
convolve	2D convolution (reduction)
correlation	datamining (arithmetic, reduction)
covariance	datamining (arithmetic, reduction)
dct	8x8 discrete cosine transform (outer-loop)
dissolve	video image dissolve (widening multiplication)
doitgen	linear algebra kernel (arithmetic,reduction,casts)
dscal	scale elements by constant (arithmetic)
gemm	linear algebra kernel (arithmetic, reduction)
gemver	linear algebra kernel (arithmetic, reduction)
gemsummv	linear algebra kernel (arithmetic, reduction)
gramschmidt	linear algebra solver (arithmetic, reduction)
interpolator	rate 2 interpolation (strided access, dot-product)
jacobi	stencil computation (arithmetic, reduction)
mixstreams	mix four audio channels (SLP vectorization)
saxpy	constant times a vector plus a vector (arithmetic)
sfir	single sample finite impulse response (reduction)

5. EXPERIMENTAL EVALUATION

In this section, we present our experimental setup and evaluate our proposal. We show that reducing the overhead of the bytecode dispatch yields excellent speedups, and that exploiting native SIMD instructions gives an additional benefit. We then analyze the behavior of our modified interpreter with respect to architectural features such as the branch predictor and the cache hierarchy. Finally, we briefly discuss code size issues.

Our goal is *not* to evaluate the merit of vectorization itself. Vectorization has been studied for decades, and it is widely discussed in the literature. We focus on conveying the benefits of vectorization to interpreters, and we study the correlation between the vectorization factor and the speedup of interpreters. For this reason, we purposely select vectorization-friendly kernels. They are not meant to be representative applications, rather to illustrate specific behaviors on simple and well-understood cases.

For completeness, we also present the performance on a number of full applications selected to illustrate the various possible behaviors (Section 5.2).

5.1. Experimental Setup

We experimented with the Polybench 1.0 suite [Pouchet et al. 2010]. We also added some kernels from our previous work on vectorization for JIT compilers [Nuzman et al. 2011]. These benchmarks are loop kernels performing computations on arrays. The data type of the array elements is parametric. For each benchmark, we varied the type of the main arrays among `char`, `short`, `int`, `float`, and `double`. Note that not all benchmarks can have their data type changed in a straightforward manner. Reasons include a shift-right operator which cannot be applied to a floating point type, or chained divisions which produce an intermediate zero when integer types are used due to rounding, and later yield to a divide-by-zero error. In some other cases, the cost model of the vectorizer decides that vectorization is not profitable for some data types. The missing combinations are simply not reported in the results. They result in neither speedups nor slowdowns since vectorization did not occur. Kernels are listed and briefly described in Table II. Across all data types, we have a total of 78 combinations.

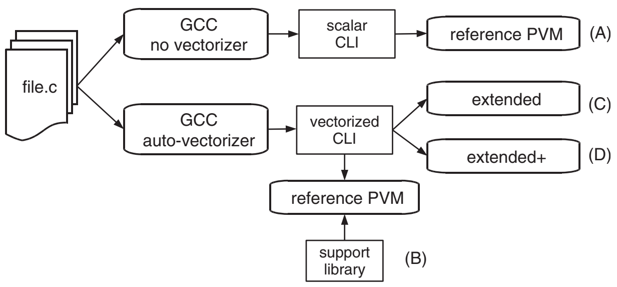

Fig. 7. Experimental execution flows.

Our target platforms aim at representing both high-end and embedded mobile systems. We experimented with:

—an Intel Xeon W3550 workstation, 3.07 GHz, running Linux and supporting up to SSE4.1; SSE has eight 128-bit vectors (in 32-bit mode);
—an ARMv7 board, 720 MHz, running Linux, supporting the NEON SIMD extension. NEON has 32 64-bit SIMD registers. These registers can also be addressed as 16 128-bit registers, but in this work we intentionally restrict ourselves to the 64-bit view to illustrate vector lengths different from SSE.

Each benchmark is run in four different configurations, as illustrated in Figure 7.

—*reference (A)*. In this configuration, the bytecode is not vectorized, it is compiled by the gcc4cli compiler at the `-O2` optimization level and run with the unmodified *reference* PVM. This configuration serves as a baseline.
—*legacy (B)*. The vectorized bytecode is run by an unmodified interpreter, unaware of the special semantics of vector builtins. The builtins are executed as regular function calls to the companion library. The purpose of this configuration is twofold: (1) to validate that the vectorized bytecode is legal and can be run by an unmodified VM; (2) to estimate the penalty incurred by these legacy interpreters. The experimental results serve as a motivation of this work. They are shown on Figure 3 and discussed in Section 3.
—*extended (C)*. The vectorized bytecode is run by the interpreter extended with SIMD IR instructions: all vector builtins are recognized by the bytecode verifier and expanded into an intermediate representation. No function call to the library occurs.
—*extended+ (D)*. This is similar to the previous configuration, but in addition SIMD IR instructions are executed by the interpreter using target SIMD instructions to the extent possible.

5.2. Detailed Performance Analysis

Our main objective is performance, i.e. the total runtime of applications. We compare the execution time of our proposal (configurations *extended* (C) and *extended+* (D)) to the reference interpreter (configuration A).

5.2.1. Extended. Figure 8 illustrates the speedup of the *extended* configuration with respect to *reference*, computed as the ratio t_A/t_C of their execution times, for both ARM (left bar, white) and x86 (central bar, gray). In addition, the reduction in the number of IR bytecode dispatches is shown in the rightmost bar (black). Note that this reduction is independent of the underlying architecture as long as the same interpreter runs the bytecode. The kernels are organized by data type: `double`, `float`, `int`, `short`, and `char`.

Average speedups range from 9% to 8× across all the data type configurations, in both evaluated architectures. Our experiments show that the performance increase

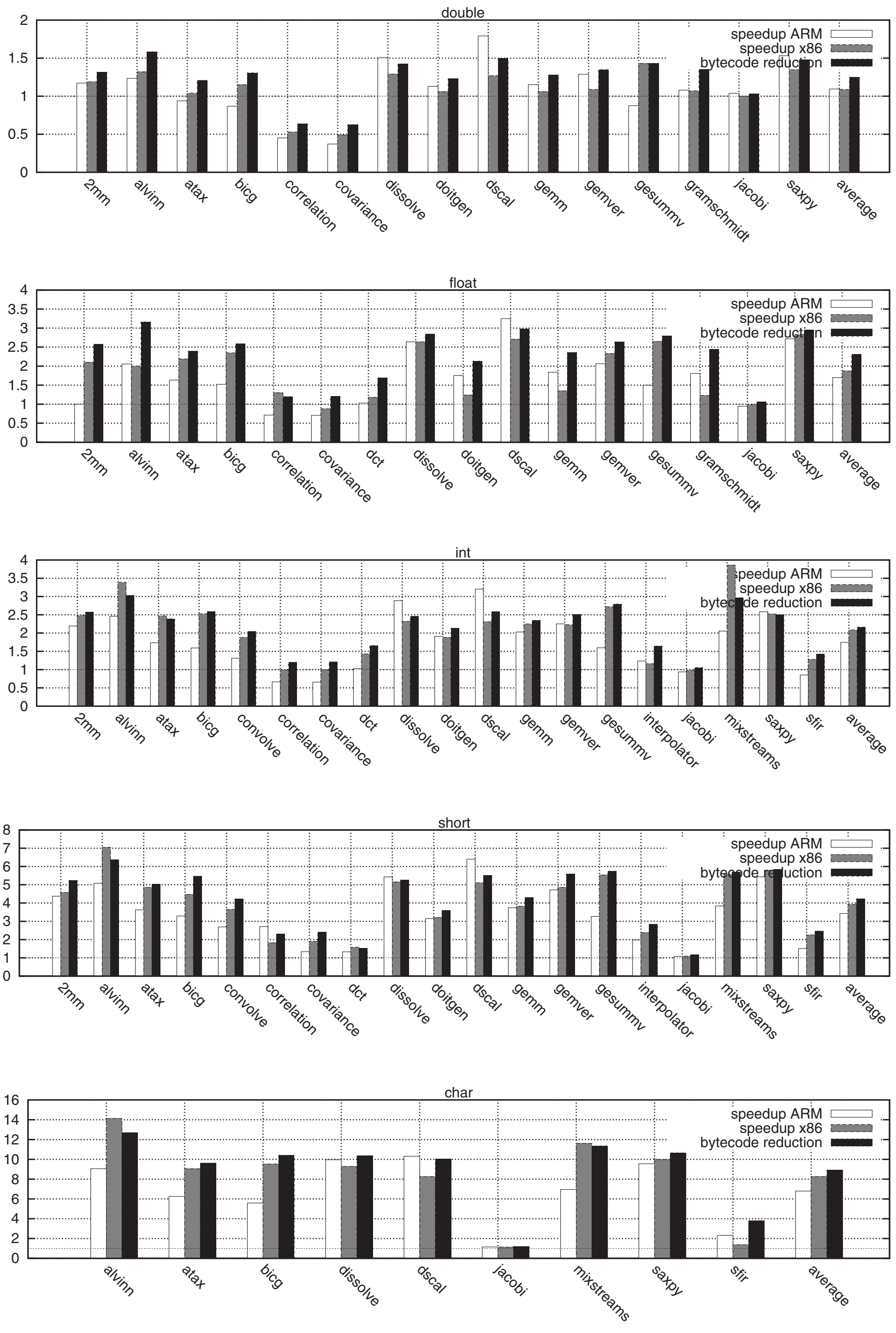

Fig. 8. Speedup and reduction in number of executed bytecodes (*extended* wrt. *reference*).

]

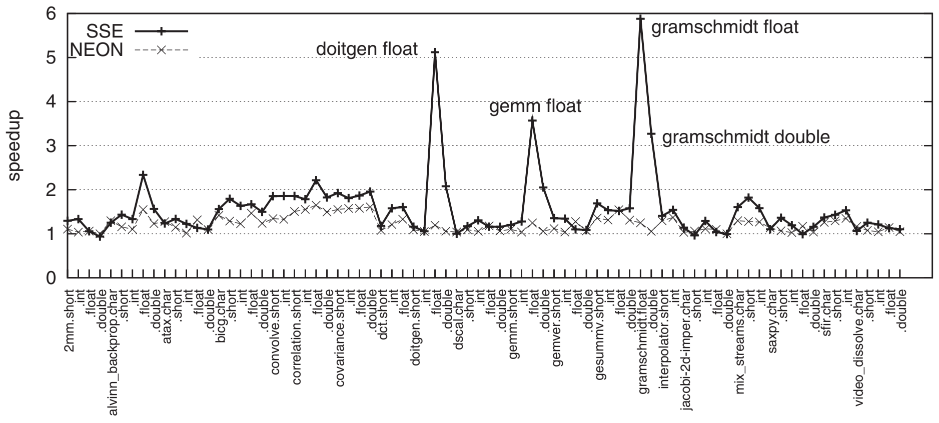

Fig. 9. Speedup using native SIMD instruction set (*extended*+ wrt. *extended*).

is directly correlated with the reduction in the number of dispatched IR bytecodes. Also, the results show that data types are influential in the observed speedups. On x86, kernels operating on `char`, `short` and `integer/float` benefit from an average 8×, 4× and 2× speedup respectively. The `double` data type results in 9% speedup. Similar numbers are observed for ARM. The reason is as follows: the amount of overhead removed with the introduction of SIMD IR instructions is mostly proportional to the vectorization factor. Some unavoidable overhead remains, inherent to the vectorization technique, explaining why the speedup is not strictly equal to the vectorization factor. This overhead arises from additional code to handle alignment on iteration spaces that are not multiple of the vectorization factor.

Across all data types, the performance of `jacobi` is almost unchanged. The reason is that the only vectorized loop is an inner loop that only copies an array into another one. The benefits of this optimization, while theoretically nonnull, is hidden by the weight of the rest of the computation. Similarly, the kernels `correlation.double` and `covariance.double` experience slowdowns. In these two cases, the small benefit of the new IR instructions is insufficient to recoup the overhead of vectorization.

5.2.2. Extended+. This configuration provides an additional contribution. It shows that combining the dispatch benefits of SIMD IR instructions with the exploitation of the native SIMD instruction set yields extra performance gains. In this scenario, SIMD IR instructions are executed by means of one or few native instructions (see the example on Figure 6(b)). Figure 9 illustrates, for each kernel, the speedups obtained by SSE4.1 and NEON (*extended*+ configuration) over *extended*, computed as the ratio of execution times t_C/t_D. The kernels are distributed along the horizontal axis. We first observe that the trends are similar for the two instruction sets: SSE and NEON can accelerate the same parts of the interpreter. SSE achieves a better speedup: 58% on average (42% when ignoring the four outstanding values), compared to NEON's 22%. Two reasons explain the lower performance of NEON.

(1) Some SIMD primitives could not easily be expressed with the NEON instruction set, hence are scalarized.
(2) NEON operates on smaller vectors (64 bits), hence compared to SSE we need twice as many instructions to implement the 128-bit vector semantics.

Four benchmarks have an outstanding performance with SSE, with a speedup above 3×: `gramschmidt` for data type `double` and `float`, `doitgen float`, and `gemm float`.

Table III. Summary of the Benchmarks

Name	Origin	Overall speedup	Comment
tiffdither	MiBench	none	no loop vectorized
IS	NAS Parallel Benchmarks 3.3.1	0.25%	some loops vectorized, no speedup
DC	NAS Parallel Benchmarks 3.3.1	none	
scimark	scimark 2.1c	1%	some speedup
tiff2bw	MiBench	2.2×	
dither-thr	adapted from MiBench	27%	significant overall speedup
alpha	inspired from MiBench	1.7×	

These numbers are due to a particularly underperforming *extended* configuration. This phenomenon is also visible in Figure 8 where the x86 target performs poorly respective to ARM and to the reduction in executed instructions. We confirmed this by measuring the IPC of the interpreter when executing these kernels (defined as the average number of native instructions executed per clock cycle). We observed that the IPC is particularly low, between 0.23 and 0.43, and the number of stalls due to the branch prediction unit is high[4].

5.3. Performance of Applications

As mentioned, the purpose of this article is by no means to evaluate the merit of vectorization as a compiler optimization, but to study how a feasible acceleration can be conveyed to interpreters.

Amdahl's law [Amdahl 1967] states that programs cannot run faster than their sequential part. Assuming that a single loop can be vectorized in an application, and that this loop accounts for a proportion p of the total runtime, and s is the acceleration of the vectorized loop, then the maximum speedup is $S = \frac{1}{(1-p)+p/s}$. The smaller the value of p, the smaller the speedup, regardless of the efficiency of the vectorizer (i.e., the value of s).

We selected a set of benchmarks that illustrate the various situations. See Table III for a summary. We considered the two benchmarks of the NAS Parallel Benchmark written in C: `IS` [Bailey et al. 1994] and `DC` [Frumkin and Shabanov 2003], the implementation of `scimark` [Pozo and Miller 2000] in C, and some image processing benchmarks from the MiBench suite [Guthaus et al. 2001]: Floyd-Steinberg dithering (`tiffdither`), conversion to black and white (`tiff2bw`). We also added two benchmarks in the spirit of the MiBench: `dither-thr` (a different algorithm of dithering, see the following), and `alpha` (alpha blending of two images).

`Scimark`, `IS`, and `DC` are unmodified. We modified parts of the MiBench for two reasons: (1) the gcc4cli compiler is not able to handle all aspects of C and POSIX; (2) we outlined compute-intensive loops to make measurements easier at function level.

We compiled all benchmarks with and without vectorization. We ran each benchmark to completion on the x86 workstation, and we collected the total number of executed bytecodes, and the execution time. Vectorized versions are run with the *extended/extended+* PVM, scalar versions are run by the reference PVM. We also collect this data for the vectorized loops when appropriate.

5.3.1. tiffdither. `tiffdither`, from MiBench, implements the Floyd-Steinberg dithering algorithm. The critical loop contains a cross-iteration dependence that prevents vectorization. No loop is vectorized, hence no speedup is achieved. The additional complexity

[4]We confirmed with hardware performance counters that the increased branch misprediction ratio is not due to indirect branches, but we could not identify the source of the degradation of these pathological cases.

of the interpreter could have led to a slowdown; however, we measured that the impact is minimal. This aspect is further discussed in Section 5.4.

5.3.2. NAS Parallel Benchmarks. We used the latest version of the NAS Parallel Benchmarks at the time of writing, namely NPB-3.3.1. Only `IS` and `DC` are written in C. To keep running times reasonable, we used a class A for `IS` and class S for `DC`. Two loops are vectorized in `IS`, and four loops in `DC`. Neither shows a significant speedup.

In `IS`, the two vectorized loops are in the initialization section of the benchmark, and iterate respectively only 2048 and 512 times. The vectorized benchmark executes 0.4% fewer bytecodes in 0.25% less time (79.29 s instead of 79.47 s).

Out of the four vectorized loops in `DC`, one is never executed, the other three iterate respectively 1, 5, and 59 times. Both versions report the same runtime (1.16 s). The vectorized version executes an additional 80 bytecodes (out of 159 million) to account for the vectorization overhead.

5.3.3. scimark. We used `scimark` version 2.1c, in C [Pozo and Miller 2000], and we ran the "large" dataset. Four loops are vectorized, only one is relevant for performance.

`Scimark` reports performance in Mflops for each of its subparts (FFT, SOR, Monte-Carlo, Sparse matmult, LU). We observe a 12% speedup in FFT (from 3.31 Mflops to 3.72 Mflops). FFT operates on double precision, the vectorization factor is 2 for 16-byte containers. The speedup, partially offset by the vectorization overhead, is consistent with what we observe with the kernels. Because the composite score of `scimark` is the arithmetic average of all subparts, `scimark` achieves only a 1% speedup overall.

5.3.4. Image Processing. Finally, we considered three image processing benchmarks taken or inpired from the MiBench suite [Guthaus et al. 2001]. All of them use as input 3648x2736 images.

—`tiff2bw` converts a color image to black and white. The picture is internally represented as three arrays of 16-bit short int (red, green, and blue components). Processing consists in multiplying corresponding elements by a compile-time constant, and then adding them. The scalar version requires 329 million bytecodes to execute the loop in 5.14 s. The vectorized version takes 44.9 million bytecodes in 0.75 s. The vectorization factor is 8, the reduction in the number of executed bytecodes is 7.3×, and the speedup of the loop is 6.9×. In this particular case, the compiler was able to prove that the arrays were already properly aligned and kept the overhead to a minimum, hence the speedups. The speedup of the whole benchmark is 2.2×, from 8.15 s down to 3.74 s. SSE instructions contribute to 7.7% of the speedup (configuration *extended*+).

—The data dependency that prevents vectorization in `tiffdither` derives from the algorithm that diffuses quantization error to the neighboring pixels. The `dither-thr` benchmark is inspired by `tiffdither`, but we implemented threshold dithering instead of Floyd-Steinberg. The benchmark internally represents the grey-level image as an array of 32-bit integers. Arithmetic operators consist in max and subtraction. The reduction in executed bytecodes in the loop is 2.9× (from 180 million down to 62.5 million), and the vectorized loop executes 2.4× faster (1.54 s versus 0.64 s). This is in line with the results of the kernels for a vectorization factor of 4. The whole application is 27% faster: from 3.9 s down to 3.07 s. SSE instructions contribute 6.3% of the speedup.

—In order to vary the arithmetic operator, we implemented a third image processing benchmark in the spirit of the first two. Alpha blending is a way produce translucency: two images, typically a foreground and a background, are combined, the alpha channel characterizing the level of translucency of the foreground. This benchmark represents each grey-level image as an array of 32-bit integers. The arithmetic operators include multiplication by a parameter, addition, and division by a runtime computed value. The vectorization factor is 4, the reduction is the number of bytecodes is 2.6× (from 240 million down to 92.5 million), and the loop speedup is 2.7× (from 4.55 s down to 1.69 s). Again, these speedups match the results of the kernels. The speedup of the benchmark is 1.7×: from 6.77 s down to 3.94 s. SSE instructions contribute to 20% of the speedup.

5.4. Architectural Behavior

We analyzed the architectural behavior thanks to hardware performance counters. We focused our analysis on the x86 platform because it provides a large spectrum of countable events and many performance counters. These counters can be easily accessed through the PAPI library [Browne et al. 2000].

5.4.1. Branch Prediction. The accuracy of the indirect branch predictor can be key for an interpreter implemented with a `switch` statement: the main control consists in a tight loop that contains an indirect jump at each iteration, the number of possible targets being rather large. We observed that the number of executed indirect jumps is on average equal to the number of executed IR bytecodes. In practice, only the bytecode dispatch accounts for indirect jumps.

We measured that the branch penalty on our x86 target is about 20 cycles, and the average interpreter loop takes on average 22 cycles for the *reference* configuration and 35 cycles for the *extended* configuration (which implements the more complex SIMD IR instructions).

In the *reference* configuration, on x86, 18% of cycles are wasted in mispredicted indirect jumps. The *extended* configuration has 21% of the cycles lost. This increase, however, is more than compensated by the reduction in the number of executed bytecodes (hence indirect jumps). In the *extended+* configuration, misprediction of indirect jumps accounts for 25% of the total cycles. Part of this latter increase is simply due to a faster implementation of the vector bytecodes, hence a reduction of the *total* number of cycles, which implies a higher ratio.

We also observed that the absolute number of mispredictions varies significantly between configurations, but also across kernels. The only difference between configurations *extended* (C) and *extended+* (D) in the interpreter source code is the implementation of `switch` entries for the new IR instructions. The addresses of the corresponding labels vary accordingly. We verified on hand-written microbenchmarks—both with hardware counters and a simulator—that this can be enough to generate large variations in the behavior of the indirect-jump branch predictor.

Finally, we note that modern branch predictors are increasingly good at predicting indirect jumps. We ran our extended interpreter with all kernels on a x86 simulator and measured the performance of a recently proposed predictor, the ITTAGE [Seznec and Michaud 2006] (Indirect Target TAgged GEometric history length), configured with 18kbits of storage. The average misprediction rate of indirect branches is 0.5% for configuration C (*extended*) and 0.6% for configuration D (*extended+*), showing that the impact of indirect branch mispredictions diminishes with the introduction of more recent and accurate predictors.

5.4.2. Caches. The main loop of the interpreter is a large `switch` statement, consisting of 546 entries (478 regular bytecodes, 68 superinstructions). We added 121 new IR instructions. Many original entries, and all new ones are fully implemented inside the `switch` statement. Table IV reports the code size (in bytes) of this loop. The difference between *reference* and *extended* directly represents the additional IR instructions. The L1 cache of both the ARM and Intel machines are 32 KB, meaning that most of the interpreter still fits in the first-level cache. Performance counters also show that the L1 instruction cache miss rate is below 0.1% in all cases.

Table IV. Code Size (bytes) of Main Interpreter Loop

	reference	extended	extended+
x86/SSE	20251	24900	23806
ARM/NEON	25692	30900	31236

New SIMD IR instructions do not modify how data is accessed beyond what loop vectorization does. In our kernels, we measured that data cache miss rate is not impacted by our proposal and remains insignificant.

5.5. Code Size

Vectorization is known for increasing code size. This can be an issue for JIT compilers because compilation time tends to increase linearly with code size. Interpreters are much less impacted, because only the verification phase and the generation of the IR are slightly slower. Thanks to the SIMD IR recognition, the average increase in the *static* number of IR bytecodes due to vectorization is only 6%.

6. RELATED WORK

Ertl and Gregg [2003] improve the performance of interpreters thanks to replicated instructions and superinstructions. Our approach is similar to their static superinstructions, but it differs in several aspects: we handle much coarser-grain patterns, including whole loops; we rely on builtins to recognize the patterns; these patterns are produced by aggressive optimizations (vectorization) in the offline compiler as opposed to profiling.

In a more recent article, Ertl and Gregg study the combination of stack caching with dynamic superinstructions [Ertl and Gregg 2004]. Our work also extends the IR instruction set. Stack caching is not implemented. However, these two techniques are shown independent and mutually beneficial. Stack caching could benefit our approach as well, and improve all our configurations.

Jump threading [Bell 1973] has been initially proposed as an alternative to native code. Some interpreters exploit the idea to improve the branch prediction of the bytecode dispatch. Virtual machines without threaded code have been shown [Ertl and Gregg 2001] to have bad indirect branch prediction rates. We observe that with more modern predictors, the dispatch with nonthreaded code is no longer a performance bottleneck. In addition to making the code simpler, it has the following advantage: threaded code can typically not be written in standard C and requires either assembly code or GNU extensions (*labels as values*).

Very recent work integrates initial automatic vectorization capabilities in virtual machines for Java [Nie et al. 2010]. The technique targets desktop and server machines, while we also consider constrained mobile devices. Their Java Vectorization Interface is similar in spirit to the builtin mechanism we use, but ours is more generic in the sense that our bytecode does not assume any specific ISA or even vector length.

Several languages provide extensions or libraries that let programmers take advantage of SIMD instruction set extensions. We discuss `Mono.Simd` [de Icaza 2008] in Section 2. The GNU Compiler Collection (GCC) proposes attributes and compiler builtins (see Section 6.49 of the manual [Stallman and the GCC Developer Community 2011]). With Java3D, Oracle provides the `vecmath` package [Clingman et al. 2004] that defines types (tuples, vectors, and matrices) appropriate for direct mapping on SIMD instruction sets. In contrast to our approach, these types are meant to be directly used by the programmer. Our solution leverages the wealth of research in autovectorization.

With other colleagues, we previously [Rohou et al. 2011; Nuzman et al. 2011] proposed the split vectorization technique that serves as a base for our SIMD IR instructions. It is presented in detail in Section 2. While both approaches build on the same offline compilation steps, the notable difference is in the type of execution environment: previous work improves the native code generated by a JIT compiler, while our current focus is to improve an interpreter by building SIMD IR instructions. By taking the same bytecode as input, we ensure compatibility with the previous technique.

7. CONCLUSION

Interpreters are in use in many systems. They are significantly slower than JIT compilers, but also much less complex to develop and maintain. Vendors of embedded systems may content themselves with interpretation for their lower-end products, or even prohibit dynamic code generation (hence JIT compilers) on their platforms. Some users of high-performance computing also favor interpreters because of the faster development cycles of their applications.

Instruction dispatch is the main overhead of interpreters. This work proposes to build coarse-grain IR instructions by leveraging the well-established and powerful autovectorization techniques available in an offline compiler. Our experiments on loop kernels obtain significant speedups that are proportional to the vectorization factor, up to 8× on average when arrays of bytes are involved. We also map SIMD IR instructions to actual vector instructions to obtain further speedup, on average 22% and up to 65% on ARM NEON, and 42% on average and up to 133% on x86 SSE. We also analyze the behavior of the architecture when running our interpreter and we show that cache and branch predictors behave well in the presence of SIMD IR instructions.

REFERENCES

AMDAHL, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (AFIPS'67)*. ACM, New York, 483–485.

BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, L., FATOOHI, R., FINEBERG, S., FREDERICKSON, P., LASINKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. 1994. The NAS parallel benchmarks. Tech. rep. RNR-94-007, NASA.

BELL, J. R. 1973. Threaded code. *Comm. ACM 16*, 370–372.

BROWNE, S., DONGARRA, J., GARNER, N., LONDON, K., AND MUCCI, P. 2000. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the Conference on Supercomputing*.

CAMPANONI, S., AGOSTA, G., CRESPI REGHIZZI, S., AND DI BIAGIO, A. 2010. A highly flexible, parallel virtual machine: Design and experience of ILDJIT. *Softw. Pract. Experi. 7,* 2, 177–207.

CLINGMAN, D., KENDALL, S., AND MESDAGHI, S. 2004. *Practical Java Game Programming*. Jenifer Niles.

COSTA, R., ORNSTEIN, A. C., AND ROHOU, E. 2007. CLI back-end in GCC. In *Proceedings of the GCC Developers' Summit*. 111–116.

DE ICAZA, M. 2008. Mono's SIMD support: Making mono safe for gaming. http://tirania.org/blog/archive/2008/Nov-03.html.

ECMA International 2006. *Common Language Infrastructure (CLI) Partitions I to IV,* 4th ed. ECMA International.

ERTL, M. AND GREGG, D. 2001. The behavior of efficient virtual machine interpreters on modern architectures. In *Euro-Par Parallel Processing*, Lecture Notes in Computer Science, vol. 2150. 403–413.

ERTL, M. A. AND GREGG, D. 2003. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, 278–288.

ERTL, M. A. AND GREGG, D. 2004. Combining stack caching with dynamic superinstructions. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME'04)*. ACM, New York, USA, 7–14.

FRUMKIN, M. A. AND SHABANOV, L. 2003. Arithmetic data cube as a data intensive benchmark. Tech. rep. NAS-03-005, NASA. February.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*.

MONO. The mono project. http://www.mono-project.com.

MOONLIGHT 2009. Moonlight. http://www.go-mono.com/moonlight.

NAUMANN, A. AND CANAL, P. 2008. The role of interpreters in high performance computing. In *Proceedings of the XII Advanced Computing and Analysis Techniques in Physics Research Conference*. 65.

NIE, J., CHENG, B., LI, S., WANG, L., AND LI, X.-F. 2010. Vectorization for java. In *Network and Parallel Computing*. 3–17.

NUZMAN, D., DYSHEL, S., ROHOU, E., ROSEN, I., WILLIAMS, K., YUSTE, D., COHEN, A., AND ZAKS, A. 2011. Vapor SIMD: Auto-Vectorize once, run everywhere. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

NUZMAN, D. AND HENDERSON, R. 2006. Multi-Platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*.

NUZMAN, D. AND ZAKS, A. 2006. Autovectorization in GCC—Two years later. In *GCC Developer's Summit*.

ORACLE. 2010. Frequently asked questions about the java hotspot vm. http://www.oracle.com/technetwork/java/hotspotfaq-138619.html.

PALECZNY, M., VICK, C., AND CLICK, C. 2001. The java hotspot^{TM} server compiler. In *Proceedings of the Symposium on Java^{TM} Virtual Machine Research and Technology Symposium (JVM'01)*. Vol. 1. USENIX Association, Berkeley.

PAPA, J. AND KINNEY, A. 2010. *Silverlight 4 Overview – Technical Features*. Microsoft.

POUCHET, L.-N., BONDHUGULA, U., BASTOUL, C., COHEN, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

POZO, R. AND MILLER, B. 2000. Scimark. version 2.1c. http://math.nist.gov/scimark2/.

ROHOU, E., DYSHEL, S., NUZMAN, D., ROSEN, I., WILLIAMS, K., COHEN, A., AND ZAKS, A. 2011. Speculatively vectorized bytecode. In *Proceedings of the International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*.

SEZNEC, A. AND MICHAUD, P. 2006. A case for (partially) TAgged GEometric history length branch prediction. *J. Instruc. Level Parall. 8*.

STALLMAN, R. M. AND THE GCC DEVELOPER COMMUNITY. 2011. *Using the GNU Compiler Collection – For GCC 4.6.1*. Free Software Foundation, Inc.

WIEPRECHT, E., BRUMFIT, J., BAKKER, J., DE CANDUSSIO, N., GUEST, S., HUYGEN, R., DE JONGE, A., MATTHIEW, J. J., OSTERHAGE, S., OTT, S., SIDDIQUI, H., VANDENBUSSCHE, B., DE MEESTER, W., WETZSTEIN, M., WIEZORREK, E., AND ZAAL, P. 2004. The HERSCHEL/PACS common software system as data reduction system. In *Astronomical Data Analysis Software and Systems Conference (ADASS) XIII*.

XAMARIN. 2011. MonoTouch limitations. http://ios.xamarin.com/documentation/limitations.

Received July 2011; revised May 2012; accepted August 2012