Hybrid Type Legalization for a Sparse SIMD Instruction Set

YOSI BEN ASHER and NADAV ROTEM, Haifa University

SIMD vector units implement only a subset of the operations used by vectorizing compilers, and there are multiple conflicting techniques to legalize arbitrary vector types into register-sized data types. Traditionally, type legalization is performed using a set of predefined rules, regardless of the operations used in the program. This method is not suitable to sparse SIMD instruction sets and often prevents the vectorization of programs. In this work we introduce a new technique for type legalization, namely vector element promotion, as well as a hybrid method for combining multiple techniques of type legalization. Our hybrid type legalization method makes decisions based on the knowledge of the available instruction set as well as the operations used in the program. Our experimental results demonstrate that program-dependent hybrid type legalization improves the execution time of vector programs, outperforms the existing legalization method, and allows the vectorization of workloads which were not vectorized before.

Categories and Subject Descriptors: D.34 [**Processors**]: Compilers

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Compiler, SIMD, vector

ACM Reference Format:
Ben Asher, Y. and Rotem, N. 2013. Hybrid type legalization for a sparse SIMD instruction set. ACM Trans. Architec. Code Optim. 10, 3, Article 11 (September 2013), 14 pages.
DOI: http://dx.doi.org/10.1145/2509420.2509422

1. INTRODUCTION

Modern processors contain Single Instruction Multiple Data (SIMD) vector execution units which enable data-level parallelism. These execution units are known to accelerate the execution of many workloads. SIMD instructions were first used by supercomputers [Russell 1978] in the 1970s, but have since made their way to PCs and even hand-held and mobile device processors.

SIMD instruction sets are known to be high throughput and power efficient compared to scalar instructions. In fact, SIMD instruction sets have become the dominant architecture for graphic processors. However, implementing these instruction sets has a cost (in area and power), and only the most useful subset of instructions is implemented. For example, the Intel AVX [Intel 2011b] (Advanced Vector Extensions) instruction set implements vector operations on 256-bit vectors, which extends the previous generation instruction set that operated on 128-bit vectors. However, not all of the operations that were supported by the 128-bit vector unit (SSE) are supported by AVX. AVX only implements the floating point operations and not integer operations. Similarly, previous generations of the Intel vector instruction set were also sparse. For example, in SSE4, the vector shift-left operation was implemented, while the vector shift-right

Authors' addresses: Y. Ben Asher and N. Rotem (corresponding author), Haifa University, Haifa, Israel; email: nadav256@gmail.com.

DOI: http://dx.doi.org/10.1145/2509420.2509422

operation was not. The 32-bit and 16-bit shift operations were implemented, but the 8-bit and 64-bit shift operations were not.

Sparse instruction sets are ubiquitous, and they raise a serious challenge for compilers targeting SIMD processors.

Traditionally, compilers had to translate high-level language types that were not implemented by the processor, for example, the emulation of 64-bit integers on 32-bit processors, or the implementation of 8-bit operations on RISC processors for which the minimal word size is 32 bits. This process is called *type legalization* [Lattner and Adve 2004]. The *type* legalization phase transforms the program so that all of the calculations in the program operate on legal types, meaning that they can fit into a machine register. After type legalization, the *operation* legalization phase ensures that all of the operations in the program can be implemented by a single machine instruction.

Unlike type legalization of scalars, vector type legalization for hardware with a sparse instruction set adds another dimension of complexity. The quality of the type legalization phase is critical, and in many cases poor type legalization prevents the vectorization of programs.

The *type* legalization phase can handle vector types in several different ways. This flexibility can affect the *operation* legalization phase greatly, especially in cases of a "hole" in the instruction set. The *operation* legalizer, which is aware of the available instruction set and the operations used in the program, has no way to influence the decision of the type legalizer.

There are inherent trade-offs between the different type legalization techniques which affect the code quality dramatically. In Section 1.2, we discuss these trade-offs and demonstrate the importance of selecting the optimal type legalization technique.

Our work addresses this problem and makes the following contributions.

(1) We propose a new type legalization mechanism, *vector element promotion*, which enables efficient code generation for many programs. This new legalization kind was implemented by the authors of this article and has been integrated into LLVM in version 3.1.
(2) We present the trade-offs between the two possible vector type legalization methods and the dependence of type legalization on the operations used in the program. This is done by studying a variety of kernels from multiple domains.
(3) We design a polynomial-time hybrid type legalization algorithm for combining the type legalization techniques using the analysis of operations in the input program and the available instruction set. We demonstrate the efficiency of this approach using the LLVM compiler.
(4) Our work contributes to the approach of VaporSIMD, AnySL, and the Intel OpenCL SDK [Karrenberg and Hack 2011; Nuzman et al. 2011; Intel 2011a] which vectorize target independent IR and rely on the code generation to produce optimal code. Our work demonstrates that much of the configuration work regarding different vectorization parameters that was previously done in the vectorizer can be done by the code generator.

The remainder of this article is organized as follows.

We start by presenting a new type legalization technique in Section 1.1 and discuss the trade-offs between the possible type legalization techniques in Section 1.2. In Section 2, we discuss the need to combine multiple legalization kinds and present our hybrid type legalization mechanism.

In Section 3, we evaluate our hybrid type legalization solution on a number of programs. In Section 4, we survey related works. Lastly, we present our conclusions and planned future research in Section 5.

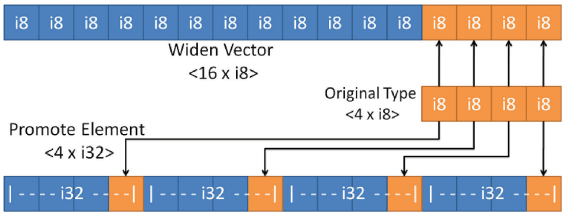

Fig. 1. Possible type legalization techniques.

1.1. Vector Element Promotion

In this article we use the notation $<N \times T>$ to describe vector types. The N variable denotes the number of vector elements and T denotes the element type. The element type iXX indicates an integer which is XX bits wide.

Integer promotion of scalars [Redwine and Ramsey 2004] is a well-known and very important compiler transformation which allows the use of narrow types on wide machine instructions. In our research, we have extended this method to operate on vectors. Much like scalar transformations, each element in the vector is widened to a type which is represented by more bits. For example, using this method on an X86 machine, the type $<4 \times i16>$ is promoted to the type $<4 \times i32>$, which is natively supported by the processor. Figure 1 depicts two possible vector type legalizations for the $<4 \times i8>$ type on X86 (excluding scalarization).

Saving "garbage" data in the upper part of the word does not damage the correctness of the program, as long as the higher-bit data does not find its way into the lower bits that are used. The integer promotion transformation carefully ensures that this is always the case. For example, if a program uses a multiplication of two 8-bit integers then the program can be changed to a multiplication of 32-bit integers, since the lower 8 bits of the 32-bit result only depend on the lower 8 bits of the multiplicands. On the other hand, if a program uses a division of two 8-bit integers, then extra code will be generated on the corresponding 32-bit values to ensure that the upper bits are correctly extended from the sign bits, since otherwise the result of the division will be incorrect in the lower 8 bits.

With the implementation of the vector element promotion type legalization technique, there are three ways to legalize vector types.

(1) *Scalarization*. Split the incoming vector into multiple scalars.
(2) *Vector Widening*. Add unused vector elements to widen the vector to fit in a physical register. With this legalization method, the type of the vector elements does not change, only the number of elements. For example, a vector of four 8-bit integers would be widened to a vector of sixteen 8-bit integers.
(3) *Vector Element Promotion*. Promote (increases in size) the type of each element to make a wider register. In this method, the number of vector elements does not change, only the size of each element. For example, a vector of four 8-bit integers would be promoted to a vector of four 32-bit integers.

Scalarizing vectors is generally a poor strategy for type legalizing code. When comparing the other two type legalization techniques, it is not clear which method is better and should be used.

1.2. Trade-Offs in Type Legalization of Vector Types

In this section, we examine two functions (depicted in Figure 2) and compare the efficiency of the two type legalization kinds operated on each function. We compiled the programs on an X86 processor with the SSE4 instruction set. Both of these programs

```
define <4 x i8>
@function_A(<4 x i8> %x, <4 x i8> %y) {
  %T0 = mul <4 x i8> %x, %y
  %T1 = add <4 x i8> %T0, %x
  ret <4 x i8> %T1
}

define void
@function_B(<4 x i8>* %pA, <4 x i8>* %pB) {
  %T0 = load <4 x i8>* %pA
  %T1 = load <4 x i8>* %pB
  %T2 = add <4 x i8> %T0, %T1
  store <4 x i8> %T2, <4 x i8>* %pA
  ret void
}
```

Fig. 2. Example programs.

```
function_A:
  pextrb $1,%xmm0, %ecx
  pextrb $1,%xmm1, %esi
  pextrb $0,%xmm0, %eax
  pextrb $0,%xmm1, %edx
  ...
  mulb %dl
  movb %al, %dl
  movb %cl, %al
  mulb %sil

... Extremely long sequence ...

  pinsrb $12, %ebx, %xmm2
  pinsrb $13, %r10d, %xmm2
  pinsrb $14, %r9d, %xmm2
  pinsrb $15, %r8d, %xmm2
  paddb  %xmm0, %xmm2
  movdqa %xmm2, %xmm0
  ...
  ret
```

```
function_B:
  movd (%rsi), %xmm1
  movd (%rdi), %xmm0
  paddb %xmm1,  %xmm0
  movd %xmm0, (%rdi)
  ret
```

<4xi8> => <16xi8)

Fig. 3. Assembly output for vector widening.

use the $<4 \times i8>$ vector type. This vector type does not fit into the SSE (128-bit) vector registers and needs to be type legalized in order to fit into a machine register.

Program Function_A performs two operations: Mul and Add. Figure 3 depicts the generated assembly code using the widen-vector method. During the type legalization phase, the vector is widened to a vector of 16 elements, each 8 bits wide ($<16 \times i8>$). The new type fits into an SSE 128-bit register. Next, in the operation legalization phase, the codegen attempts to match a legal SSE instruction to the vector. However, the SSE instruction set does not implement the multiplication of this type, and the code generation scalarizes the multiplication operation into a sequence of 16 element extractions, scalar multiplications, and element insertions. The "pextrb" instructions are used to extract individual elements from the vector.

Alternatively, Figure 4 presents the code generated when legalizing the program using the new promote-element type legalization. In this example, the four 8-bit integer

```
function_A:
  pmulld %xmm0, %xmm1
  paddd  %xmm0, %xmm1
  movdqa %xmm1, %xmm0
  ret
```

<4xi8> => <4xi32>

```
function_B:
  movzbl 2(%rsi), %eax
  movd  1(%rsi), %xmm0
  pinsrd $1, %eax, %xmm0
  movzbl 3(%rsi), %eax
  pinsrd $2, %eax, %xmm0
  movzbl 4(%rsi), %eax
  pinsrd $3, %eax, %xmm0
  movzbl 2(%rdi), %eax
  movd  1(%rdi), %xmm1
  pinsrd $1, %eax, %xmm1
  movzbl 3(%rdi), %eax
  pinsrd $2, %eax, %xmm1
  movzbl 4(%rdi), %eax
  pinsrd $3, %eax, %xmm1
  paddd  %xmm0, %xmm1
  pshufb CP0(%rip), %xmm1
  movd  %xmm1, (%rdi)
  ret
```

Fig. 4. Assembly output for element promotion.

vector is promoted to a four 32-bit integer vector ($<4 \times i32>$)[1]. Each element is widened and the number of vector elements remains the same. The generated code is only three instructions long.

The promote-element type legalization is the preferred legalization for program Function_A, but not for Function_B.

Program Function_B performs two load operations, a single add operation, and a store operation. Figure 3 depicts the generated assembly code using the widen-vector method. The generated code is compact and efficient due to the fact that the X86 architecture can efficiently save and load a subregister. Additionally, it supports the add operation on 16 bytes.

Figure 4 presents the code when legalizing the program using the promote-element type legalization. In this example, the generated code is longer because loading and storing the $<4 \times i8>$ has an overhead. The "pinsrd" instructions are used to insert individual elements into the vector.

The preceding example demonstrates that some programs benefit from one kind of type legalization while other programs benefit from another kind. From these examples, we can also see that the type legalization decision for vector types is heavily affected by the available instruction set as well as the operations used by the program. Moreover, we posit that combining the type legalization kinds within the same program may be beneficial.

2. HYBRID TYPE LEGALIZATION

The hybrid type legalization system combines several type legalization techniques. A decision module scans the incoming program and considers the available instruction set and the used instruction before deciding on the type legalization technique that is most likely to benefit for each operation.

2.1. Legalizations Selection

Figure 6 describes the overall design of the proposed hybrid type legalization policy. For each operation in the input program, it is necessary to find the best type legalization

[1]The type $<4 \times i32>$ on X86-SSE is the common vector size for which the instruction set is complete for all operations.

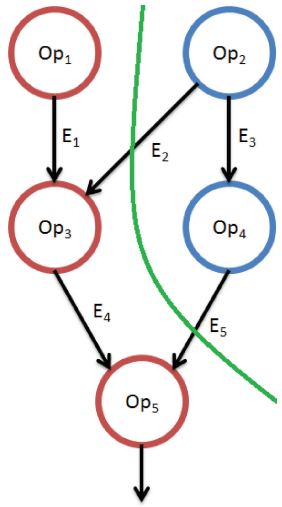

Fig. 5. Colored operation DAG.

method. A cost model is used to estimate the total cost of each of the type legalization techniques for each operation, based on the targeted instruction set.

We limit the hybrid type legalization procedure to a single basic block and process the SSA-based DAG (Directed Acyclic Graph). A type legalization decision is in essence a *graph coloring* problem of a DAG. Each node in the DAG which represents an operation can be legalized in one of several ways; these are the colors of the graph. Each typed operation in the DAG has a different estimated cost for each legalization technique. If an edge in the colored graph connects two nodes of different colors (different legalization techniques) then the compiler needs to convert one type representation to another, which requires an additional cost. An edge which connects two nodes of the same color costs nothing.

The cost of the DAG is calculated by accumulating the selected costs for the operations and edges. An optimal hybrid type legalization is obtained when the graph is colored and the estimated cost of the DAG is minimal. The graph considered in this article is weighted, directed, and acyclic. Let $V(G)$ denote the set of vertices of G and let $E(G)$ denote the set of edges of G. Let $C(G)$ denote the color of vertices in graph G. Each node in LLVM's DAG becomes a single vertex in $V(G)$. Each unweighted edge in LLVM's DAG becomes a weighted edge, using the cost table such that $E = cost[V1, V2]$. Given a graph G, the *coloring* of graph G is a mapping from $V(G)$ to integers. After the coloring of the graph, each color in $C(G)$ represents a different legalization method.

Figure 5 depicts a colored operation DAG. Each color represents a different legalization technique. The cost of the DAG in this example is evaluated by accumulating the cost of the nodes $Red[Op_1] + Red[Op_3] + Red[Op_5] + Blue[Op_2] + Blue[Op_4] + Blue[Op_5]$ and the cost of the edges $RedToBlue[E_2] + RedToBlue[E + 5]$.

The hybrid type legalization method relies on a cost table. The table describes the expected cost for each of the legalization kinds for each typed operation, and for the type conversion between each type representation. For example, the *mul* $<4 \times i8>$ operation would contain the value 1 for the promote-element method, because this operation would be promoted to the legal multiplication operation on $<4 \times i32>$. However, the cost of *mul* $<4 \times i8>$ for the widen-vector legalization would be 16, because this legalization type would result in an illegal operation which would require scalarization. In essence, our cost model accumulates the cost of hazards, which are cases in which there is

no instruction set available and the code generator would have to generate code to compensate for the missing instruction. In other words, the decision procedure attempts to estimate the overhead associated with each of the methods.

2.2. Proposed Coloring

The graph coloring problem we formalized can be solved optimally using a known [Kleinberg and Tardos 2005] polynomial-time algorithm. Due to the fact that we only need to select between two kinds of legalizations we only need to color the graph with two colors. The general problem of graph coloring is NP-complete, but the special case of two colors is solved in polynomial time. In the domain of computer vision, this algorithm is known as "background/foreground segmentation".

We generalize our DAG to a weighted nondirected graph, and connect each of the nodes to new source and sink nodes. The weight of each edge to the source node is the cost of one color, and the weight to the sink node is the cost of the other color. The max-flow min-cut partitioning of the constructed graph is the optimal coloring. The black parts of the graph is one legalization type and the white is the other. The algorithm was implemented at the LLVM basic block structure in which each instruction is a node in a DAG.

2.3. Vectorization Configuration

Type legalization can actually affect the vectorization decision. Vectorizing compilers need to decide on several parameters when vectorizing. The most important parameter is the Vectorization Factor (VF). This parameter decides on the number of elements to pack in an SIMD vector. A vectorization factor of one means that the program remains scalar. A VF of four means that four elements are computed in parallel, generating types such as $<4 \times i32>$.

Traditionally, vectorizing compilers relies on a table of legal operations which are supported by the instruction set. Vectorization is often aborted if the vectorizing compiler chooses to use an operation which is not available in the operation table.

Hybrid type legalization simplifies the configuration of VF, and enables the vectorization of additional programs. Our approach allows the vectorizing compiler to select the vectorization factor based on the common type used in the program, and all other vector types are handled by the code generator.

2.4. Code Generator Implementation

In this section, we describe the LLVM code generator. The first stage in the LLVM code generator [Koranne and Koranne 2011] is the SelectionDAG[2] phase. The SelectionDAG phase is made up of several transformations that lower the incoming program, which is in a target-independent representation, down to a scheduled list of machine instructions. The implementation of the SelectionDAG phase itself is target independent. It uses target-specific information, provided by the different backends, using the Target Lowering Information (TLI) interface.

Figure 6 depicts the general structure of the SelectionDAG phase. The LLVM SelectionDAG phase operates on a single basic block and consists of the following steps: The *build initial DAG* phase performs a simple translation from the input LLVM code to the SelectionDAG data structure. At this point, the SelectionDAG contains nodes of illegal types and illegal operations. The first *optimize DAG* stage performs simple peephole optimizations on the SelectionDAG to simplify it and recognize meta-instructions, such as multiply-accumulate for targets that support these operations. The *legalize DAG types* stage transforms SelectionDAG nodes to only use data types which can

[2]SelectionDAG also denotes the data structure used in this module.

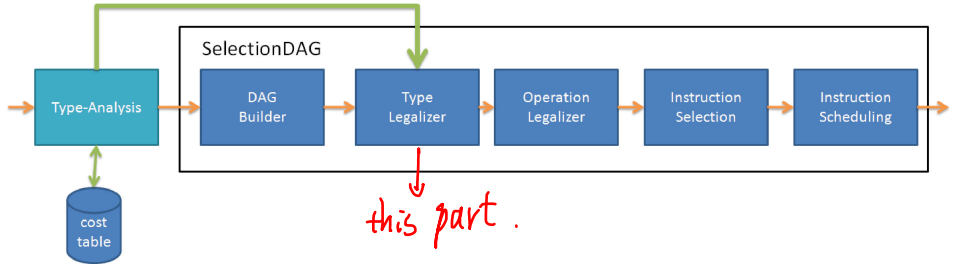

Fig. 6. Hybrid type legaliztion policy design.

fit in a machine register. The second *optimize DAG* is run to clean up redundancies exposed by type legalization. The *legalize DAG ops* stage transforms SelectionDAG nodes to replace any operations that are not supported by the processor. The third *optimize DAG* is run to eliminate inefficiencies introduced by operation legalization. The *instruction selection* phase matches machine instructions and replaces all target independent instructions. Finally, the *SelectionDAG scheduling* phase assigns a linear order to the instructions in the target instruction DAG.

In this work we focus on improving the type legalization phase. The LLVM type legalizer uses the TLI (Target Lowering Interface) interface for querying the different backends for the available physical register sizes. During this phase, scalar types are legalized by either "promoting" them to larger scalars or "expanding" them by breaking them down into smaller scalars. Vector types are traditionally legalized either by splitting them into smaller vectors (down to scalars, if needed) or by widening them to a wider legal size.

2.5. Element Promotion of Floating Point Vectors

In this work, we discuss the promotion of integer vectors. We have described the motivation for selecting one of two legalization algorithms. The same motivation also applies to floating point calculations. However, there are several differences between floating point types and integers which require special consideration. Element promotion of floating point types is considerably different and raises special challenges. A common case of element promotion of floating point types would be a conversion between a single-precision value (F32) to a double-precision value (F64). Unlike integers, calculations performed on F32 do not yield the same values as calculations made on F64 types. This is due to the fact that bigger data types perform calculations with higher accuracy. Another potential problem is the behavior of "denormalized" values. Different floating point values have different denormalized ranges, which makes correct rounding difficult. In our work, we have focused on integer element promotion. Although we did not do so in our research, it is possible to promote floating point data types despite the aforementioned limitations.

2.6. SelectionDAG Limitations

In this article we describe the legalization of types inside a single basic block. In LLVM, type legalization is a part of the SelectionDAG phase that performs instruction selection on a DAG, which is a single basic block. When we implemented LLVM's vector promotion code we ran across a few problems with the single basic block approach. One excellent example is AVX masks. In the LLVM-IR masks are represented as a vector of booleans, while in X86 masks are the highest bit in each word. When comparing two vectors of type $<8\ x\ i32>$ the result is the LLVM-IR $<8\ x\ i1>$. In SelectionDAG this type needs to be legalized into a register-sized type that can be selected. When

we type legalize this type we have two options: XMM-sized registers and YMM-sized registers. Our current implementation of vector promotion selects the smaller register kind, and peephole optimizations usually fix the sequence of cmp-select instructions into the optimal sequence. However, when the producer of the mask is in one basic block and the consumer is in another we usually generate poor code because the peephole DAG optimizations can't operate on the whole chain.

3. EXPERIMENTAL RESULTS

In this section, we present the experimental results we obtained using the modified LLVM compiler. We begin by describing the evaluation of the code generation phase, and continue by discussing the influence of the improved code generator on vectorizing compilers.

3.1. Code Generation Experimental Environment

In this section we only evaluate code generation aspects. We have evaluated our proposed method by comparing the execution time of multiple programs and evaluated programs from three main sources. The focus of this section is the evaluation of the modified LLVM code generation, and not vectorization, and thus we needed to test input programs that were already vectorized. First, we evaluated hand-coded programs which use vector types. Second, we used the ISPC 1.0.6 [Pharr and Mark 2012] compiler to generate vectorized code. ISPC compiles a C-based SPMD programming language to run on the SIMD units of CPUs. Last, we used programs which were vectorized by the GCC 4.5 autovectorizer. We used the GCC plugin DragonEgg [DragonEgg 2011] to convert the vectorized GCC IR to LLVM IR. We evaluated our hybrid type legalization system using a recent version of the LLVM compiler[3]. We used an Intel Corei7-2600 processor running at 3.40 GHz to run our workloads. We disabled the power-saving features, such as Turbo [Rotem et al. 2009] to achieve accurate measurements. We ran each test five times and used the geomean value. We calculated the standard deviation to ensure that the results are predictable. In our tests we targeted the SSE4.2 instruction set because it is the latest vector instruction set to be fully supported by the LLVM compiler. We ran our tests in 64-bit mode to allow access to all 16 XMM registers in order to reduce the influence of register pressure.

We believe that this extensive experimentation across programs from multiple domains and multiple vectorization compilers provides a strong motivation for considering hybrid type legalization for sparse vector architectures.

The first group of workloads is based on ISPC programs. Finite Impulse Response (FIR) uses the unsigned char type. Linear Feedback Shift Register (LFSR) operates on short integers. Multi is a set of microbenchmark kernels that operate on unsigned chars. MD5 is the message-digest algorithm which operates on 32-bit types. Popcnt counts the number of active bits in a 16-bit integer and saves the result in an 8-bit integer. Blum is the implementation of the Blum glow effect, which is a common shader in computer graphics. It is comprised of several kernels which operate on 8-bit integers. The next group of workloads are hand-coded using vector types. The programs are written in C, using the clang "ext_vector_type" language extension. Sum of Absolute Differences (SAD) and SquareSum use chars. Cvtsd converts an array of chars to floats, and Convolution is a 2D convolution on shorts. The last group of workloads were vectorized by GCC. Dot8 and Dot16 are the dot product algebraic operation on arrays of chars and shorts. Xor8 and Xor16 loop over arrays of chars and shorts and xor the inputs into a third array. Arith8 and Arith16 are based on Example 1 in the GCC

[3]Revision 139692.

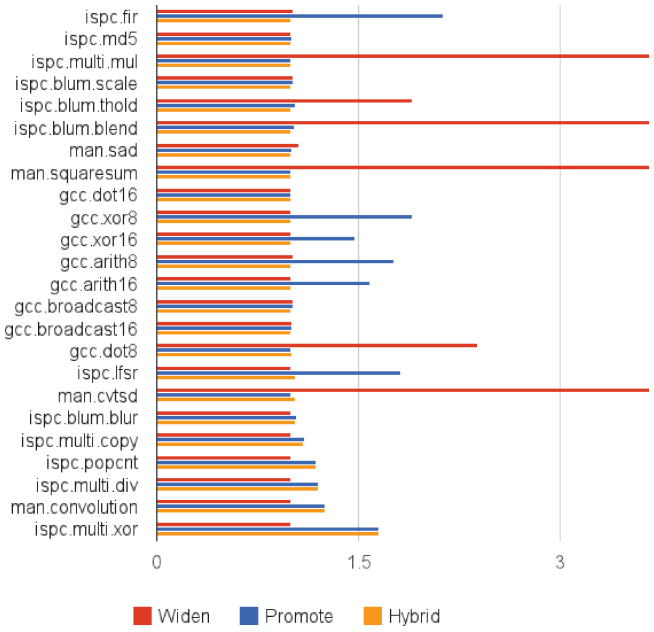

Fig. 7. Comparison of legalization methods.

autovectorization manual. Broadcast8 and Broadcast16 are based on Example 2 with char and short types.

3.2. Experimental Evaluation

Figure 7 shows the comparison of execution times among the three legalization kinds. The times are normalized to the best score and the lower the number, the better the result.

For each workload, each legalization kind is compared to the best legalization kind for this workload. The vector-widen legalization kind is 37% slower than the optimal legalization (the best of the three results). The vector promote is 26% times slower than the optimal legalization, while the hybrid legalization kind is only 5% slower than the optimal legalization kind. The 5% gap between the optimal legalization kind and the hybrid legalization kind is due to cases where the decision procedure fails to select the best legalization kind for one of the types.

4. RELATED WORKS

The vector element promotion legalization is based on the scalar promotion type legalization, which is a well-known technique. Redwine and Ramsey [2004] present an efficient method for executing narrow types on machines with wider register types. They use the term “integer widening” to describe the widening of scalar integers to wider types.[4] Almost every narrow operation can be widened by sign- or zero-extending the operands and using a target machine instruction at its natural width. Extensions, though, can sometimes be avoided. They formalize the procedure of type promotion which enables the efficient transformation of scalars. The LLVM compiler already

[4]In the context of vector legalization, we refer to this operation as “promotion” and use the term “widening” to describe the increase in the number of vector elements.

implements this method for promoting scalars, and we have expanded this method and applied it to vectors. Some programs require computations to use fewer bits of precision than are provided by the physical registers on the target machines. Kawahito et al. [2002] present an effective algorithm for eliminating sign extensions for 32-bit integers on 64-bit architectures for Java programs.

The LLVM compiler toolkit implements a platform-independent vector instruction set (LLVA) that exposes static information about vector parallelism while avoiding the use of hardware-specific parameters [Bocchino and Adve 2006]. LLVA provides both arbitrary-length vectors (for targets that allow vectors of arbitrary length, or where the target length is not known) and fixed-length vectors (for targets that have a fixed vector length, such as subword SIMD extensions).

In the field of vectorizing compilers, there are several common approaches for transforming scalar code to code which uses SIMD instructions. Some vectorizing compilers [Tenllado et al. 2005; Barik et al. 2010] combine several scalar operations within a single basic block. This approach is usually combined with other compiler optimizations, such as loop unrolling, which increases the vectorization opportunities. Other compilers of parallel programming languages rely on explicit language features for vectorization [Karrenberg and Hack 2011; Pharr and Mark 2012]. These languages were designed to be easily autovectorized. Some other compilers use polyhedral transformation for vectorization [Trifunovic et al. 2009; Grosser et al. 2011]. These compilers model loops as polytypes and apply affine transformations on them. This allows the generation of loops with no inter-iteration dependency, which is a requirement of vectorization. Some compilers vectorize the innermost loops [Tenllado et al. 2005; Barik et al. 2010; Grosser et al. 2011], while others vectorize the outer loops [Nuzman and Zaks 2008; Karrenberg and Hack 2011; Pharr and Mark 2012]. Some compilers, such as GCC, support Superword-Level Parallelism [Larsen and Amarasinghe 2000] (SLP vectorization). SLP vectorization does not benefit from type legalization because it has a very accurate cost model that avoids illegal types.

The context of our work is code generation and not the vectorization of programs. We improve the code generation type legalization phase, regardless of the vectorization technology. In our experimental results section, we present vector workloads which were hand-vectorized as well as programs which were autovectorized by multiple compilers.

In the field of compilers and code generation, there is extensive research in the areas of instruction selection, register allocation, and instruction scheduling. Compiler operation legalization is often discussed in the context of Application Specific Instruction Processors (ASIP) [Jain et al. 2001] in which the compiler needs to synthesize the hardware, rather than use a physical resource.

In our work, we use a decision procedure to select a configuration of the legalization pass. Selecting the optimal sequence of passes is a well-researched problem in the field of compilers. Cavazos et al. [2006] construct a hybrid register allocator which chooses between linear-scan and graph coloring algorithms. Machine learning has also been used extensively [Monsifrot et al. 2002; Agakov et al. 2006; Stephenson et al. 2003; Fursin et al. 2008] for selecting and configuring optimization parameters. Runtime information [Cavazos et al. 2007] can also be used to improve the optimization configuration.

Vectorizing compilers need to have detailed information about the hardware to make good vectorization decisions. For example, the SIMD machine register width determines the desired vectorization factor (the number of lanes computed simultaneously). The GCC-based Vapor SIMD compiler [Nuzman et al. 2011] developed by Nuzman et al. attempts to separate the vectorization phase from the code generation phase.

Much like LLVA [Bocchino and Adve 2006], they define special bytecode to represent vectorization primitives. Unlike LLVA which defines explicit SIMD widths, the Vapor compiler defines higher-level constructs which can be lowered to different vector widths on multiple platforms. Due to its abstraction, the Vapor compiler can even target scalar processors. This study opposes the common belief that vectorizing compilers must be aware of the target architecture. AnySL has a similar internal representation during the packetization [Karrenberg and Hack 2011] phase, but this representation is not serialized as in the Vapor compiler.

Our work focuses on improving the code generation of vector instructions, while the Vapor and AnySL compilers focus on autovectorizing programs. The aforesaid studies and the current study are orthogonal and complementary.

Type legalization is performed not only in LLVM. The Intel Fortran/C++ compiler (ICC) is a powerful vectorizing compiler targeting several architectures. This commercial compiler is described in several publications [Tian 2007; Bik et al. 2002]. ICC has a vectorization unit that generates vector code which is type legal and operation legal. The intra-register vectorization phase [Bik et al. 2002] attempts to find a legal operation for the vectorized value, and if none is found, the vectorization is aborted. Thus, type legalization happens on scalars prior to autovectorization. The GNU Compiler Collection (GCC) has an autovectorizer that generates vector instructions. Additionally, it accepts user vector code which is declared using the attribute "vector_size". Unsupported vectors and complex numbers are lowered to scalar operations. GCC does not vectorize programs if there are sequences for which there is no simple vector sequence. This is implemented in tree-vect-generic. The vector support does not necessarily need to be a single instruction, rather it can be a sequence of instructions. For example, on x86, the multiplications of $<4 \times i8>$ types are emulated using a sequence which includes the $<8 \times i16>$ type and the use of masks. The Path64 compiler does not implement vector type legalization at all, and scalarizes illegal types.

To the best of our knowledge, we are the first to consider trade-offs in vector type legalization and to consider combining several legalization kinds.

5. CONCLUSIONS

In this work, we present a hybrid type legalization method for lowering vector types, which is based on a decision procedure that combines two kinds of type legalization. The decision procedure considers the available instruction set as well as the operations used in the program. To the best of our knowledge, this is the first work to consider the available instruction set when deciding on the kind of type legalization. Our experiments using multiple vectorizing compilers demonstrate that hybrid type legalization outperforms the existing kinds of type legalization. We have demonstrated that legalizing vector types by promoting each element outperforms widening the vector in most cases, and that combining the two methods, based on the available instruction set and the operations used, outperforms the other static methods.

REFERENCES

AGAKOV, F. V., BONILLA, E. V., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the Symposium on Code Generation and Optimization.* 295–305.

BARIK, R., ZHAO, J., AND SARKAR, V. 2010. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10).* 201–212.

BIK, A. J. C., GIRKAR, M., GREY, P. M., AND TIAN, X. 2002. Automatic intra-register vectorization for the intel architecture. *Int. J. Parallel Program. 30*, 2, 65–98.

Bocchino, R. L., Jr. and Adve, V. S. 2006. Vector LLVA: A virtual vector instruction set for media processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE'06)*. ACM Press, New York, 46–56.

Cavazos, J., Fursin, G., Agakov, F. V., Bonilla, E. V., O'Boyle, M. F. P., and Temam, O. 2007. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the Symposium on Code Generation and Optimization*. 185–197.

Cavazos, J., Moss, J. E. B., and O'Boyle, M. F. P. 2006. Hybrid optimizations: Which optimization algorithm to use? In *Proceedings of the 15th International Conference on Compiler Construction (CC'06) held as part of the Joint European Conferences on Theory and Practice of Software (ETAPS'06)*. 124–138.

Duncan Sands Dragonegg. 2011. A GCC plugin for the LLVM backend. http://dragonegg.llvm.org/.

Fursin, G., Namolaru, M., Yom-Tov, E., Zaks, A., Mendelson, B., Bonilla, E., Thomson, J., Leather, H., Williams, C., O'Boyle, M., Courtois, E., and Bodin, F. 2008. MILEPOST gcc: Machine learning based research compiler. http://gcc-ici.sourceforge.net/papers/fmtp2008.pdf.

Grosser, T., Zheng, H., Aloor, R., Simburger, A., Grosslinger, A., and Pouchet, L.-N. 2011. Polly - Polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*.

Intel Corp. 2011a. Intel openCL SDK. http://software.intel.com/en-us/articles/opencl-sdk/.

Intel Corp. 2011b. Advanced vector extensions (AVX) programming reference. http://software.intel.com/en-us/avx/.

Karrenberg, R. and Hack, S. 2011. Whole function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 141–150.

Kawahito, M., Komatsu, H., and Nakatani, T. 2002. Effective sign extension elimination. *SIGPLAN Not. 37*, 5, 187–198.

Kleinberg, J. and Tardos, E. 2005. Image segmentation. In *Algorithm Design*. Addison-Wesley Longman, Boston, MA, 392–395.

Koranne, S. And Koranne, S. 2011. Compiler construction. In *Handbook of Open Source Tools*. Springer, 241–284.

Jain, M. K., Balakrishnan, M., and Kumar, A. 2001. ASIP design methodologies: Survey and issues. In *Proceedings of the 14th International Conference on VLSI Design (VLSID'01)*. 76.

Larsen, S. and Amarasinghe, S. 2000. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. 145–156.

Lattner, C. and Adve, V. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 75.

Monsifrot, A., Bodin, F., and Quiniou, R. 2002. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA'02)*. 41–50.

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 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. 151–160.

Nuzman, D. and Zaks, A. 2008. Outer-loop vectorization: Revisited for short simd architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 2–11.

Pharr, M. and Mark, W. R. 2012. Ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the Conference on Innovative Parallel Computing (InPar'12)*. 1–13.

Redwine, K. and Ramsey, N. 2004. Widening integer arithmetic. In *Proceedings of the 13th International Conference on Compiler Construction*.

Rotem, E., Mendelson, A., Ginosar, R., and Weiser, U. 2009. Multiple clock and voltage domains for chip multi processors. In *Proceedings of the International Symposium on Microarchitecture*. 459–468.

Russell, R. M. 1978. The CRAY1 computer system. *Comm. ACM 21*, 1, 63–72.

Stephenson, M., Amarasinghe, S. P., Martin, M. C., and O'Reilly U.-M. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 77–90.

Tenllado, C., Pinuel, L., Prieto, M., Tirado, F., and Catthoor, F. 2005. Improving superword level parallelism support in modern compilers. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'05)*. ACM Press, New York, 303–308.

TIAN, X. 2007. Inside the intel 10.1 compilers new threadizer and new vectorizer for intel core 2 processors. *Intel Technol. J. 11*, 4.

TRIFUNOVIC, K., NUZMAN, D., COHEN, A., ZAKS, A., AND ROSEN, I. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 327–337.

Received November 2012; revised April 2013; accepted April 2013