Liquid SIMD: Abstracting SIMD Hardware using Lightweight Dynamic Mapping

Nathan Clark[1], Amir Hormati[1], Sami Yehia[2], Scott Mahlke[1], and Krisztián Flautner[2]

[1]Advanced Computer Architecture Laboratory
University of Michigan - Ann Arbor, MI
{ntclark, hormati, mahlke}@umich.edu

[2]ARM, Ltd.
Cambridge, United Kingdom
{sami.yehia, krisztian.flautner}@arm.com

Abstract

Microprocessor designers commonly utilize SIMD accelerators and their associated instruction set extensions to provide substantial performance gains at a relatively low cost for media applications. One of the most difficult problems with using SIMD accelerators is forward migration to newer generations. With larger hardware budgets and more demands for performance, SIMD accelerators evolve with both larger data widths and increased functionality with each new generation. However, this causes difficult problems in terms of binary compatibility, software migration costs, and expensive redesign of the instruction set architecture. In this work, we propose Liquid SIMD *to decouple the instruction set architecture from the SIMD accelerator. SIMD instructions are expressed using a processor's baseline scalar instruction set, and light-weight dynamic translation maps the representation onto a broad family of SIMD accelerators. Liquid SIMD effectively bypasses the problems inherent to instruction set modification and binary compatibility across accelerator generations. We provide a detailed description of changes to a compilation framework and processor pipeline needed to support this abstraction. Additionally, we show that the hardware overhead of dynamic optimization is modest, hardware changes do not affect cycle time of the processor, and the performance impact of abstracting the SIMD accelerator is negligible. We conclude that using dynamic techniques to map instructions onto SIMD accelerators is an effective way to improve computation efficiency, without the overhead associated with modifying the instruction set.*

1 Introduction

Single-instruction multiple-data (SIMD) accelerators are commonly used in microprocessors to accelerate the execution of media applications. These accelerators perform the same computation on multiple data items using a single instruction. To utilize these accelerators, the baseline instruction set of a processor is extended with a set of SIMD instructions to invoke the hardware. Intel's MMX and SSE extensions are examples of two generations of such instructions for the x86 instruction set architecture (ISA). SIMD accelerators are popular across desktop and embedded processor families, providing large performance gains at low cost and energy overheads.

While SIMD accelerators are a proven mechanism to improve performance, the forward migration path from generation to generation is a difficult problem. SIMD hardware evolves in terms of width and functionality with each generation. For example, the Intel MMX instructions operated on 64-bit vectors and this was expanded to 128-bit for SSE2. The opcode repertoire is also commonly enhanced from generation to generation to account for new functionality present in the latest applications. For example, the number of opcodes in the ARM SIMD instruction set went from 60 to more than 120 in the change from Version 6 to 7 of the ISA.

SIMD evolution in desktop processors has been relatively stagnant recently, with vector lengths standardizing at 4 to 8 elements. However, this is not the case in embedded systems. For example, the ARM Neon SIMD instructions were extended from 4 to 16 8-bit elements in 2004 [5]. Other recent research [22] has proposed vector lengths of 32 elements are the most suitable size for signal processing accelerators. Undoubtedly, SIMD architectures are still evolving in many domains.

Migration to new generations of SIMD accelerators is very difficult, though. Once an application is targeted for one set of SIMD instructions, it must be rewritten for the new set. Hand-coded assembly is commonly used to exploit SIMD accelerators; thus, rewriting applications is time consuming, error prone, and tedious. Programming with a library of intrinsics can mitigate the problem to some degree, but software migration still requires substantial effort, as code is usually written assuming a fixed SIMD width and functionality.

To effectively deal with multiple generations of SIMD accelerators and overcome the software migration problems, this paper investigates the use of delayed binding with SIMD accelerators. Delayed binding is a technique used in many areas of computer science to improve the flexibility and the efficiency of systems. For example, dynamic linkers delay the binding of object code to improve portability and space efficiency of applications; dynamic compilers take advantage of late binding to perform optimizations that would otherwise be difficult or impossible without exact knowledge of a program's runtime environment [17]. Examples of delayed binding in processors include the use of trace caches and various techniques for virtualization [26]. Just as in software systems, these techniques aim to improve flexibility and efficiency of programs, but often require non-trivial amounts of hardware and complexity to deploy.

In this work, delayed binding of SIMD accelerators is accomplished through compiler support and a translation system, collectively referred to as *Liquid SIMD*. The objective is to separate the SIMD accelerator implementation from the ISA, providing an abstraction to overcome ISA migration problems. Compiler support in Liquid SIMD translates SIMD instructions into a virtualized representation using the processor's baseline instruction set. The compiler also isolates portions of the application's dataflow graph to facilitate translation. The translator dynamically identifies

these isolated dataflow subgraphs, and converts them into architecture-specific SIMD instructions.

Liquid SIMD offers a number of important advantages for families of processor implementations. First, SIMD accelerators can be deployed without having to alter the instruction set and introduce ISA compatibility problems. These problems are prohibitively expensive for many practical purposes. Second, delayed binding allows an application to be developed for one accelerator, but be utilized by completely different accelerators (e.g., an older or newer generation SIMD accelerator). This eases non-recurring engineering costs in evolving SIMD accelerators or enables companies to differentiate processors based on acceleration capabilities provided. Finally, SIMDized code in a Liquid SIMD system can be run on processors with no SIMD accelerator or translator, simply by using native scalar instructions.

The contributions of this paper are fourfold:

- It describes an compiler/translation framework to realize Liquid SIMD, which decouples the SIMD hardware implementation from the ISA.
- It develops a simple, ISA-independent mechanism to express width-independent SIMDization opportunities to a translator.
- It presents the design and implementation of a lightweight dynamic translator capable of generating SIMD code at runtime.
- It evaluates the effectiveness of Liquid SIMD in terms of exploiting varying SIMD accelerators, the runtime overhead of SIMD translation, and the costs incurred from dynamic translation.

2 Overview of the Approach

SIMD accelerators have become ubiquitous in modern general purpose processors. MMX, SSE, 3DNow!, and AltiVec are all examples of instruction set extensions that are tightly coupled with specialized processing units to exploit data parallelism. A SIMD accelerator is typically implemented as a hardware coprocessor composed of a set of functional units and an independent set of registers connected to the processor through memory. SIMD accelerator architectures vary based on the width of the vector data along with the number and type of available functional units. This allows for diversity in two dimensions: the number of data elements that may be operated on simultaneously and the set of available operations.

The purpose of this work is to decouple the instruction set from the SIMD accelerator hardware by expressing SIMD optimization opportunities using the processor's baseline instruction set. Expressing SIMD instructions using the baseline instruction set provides an abstract software interface for the SIMD accelerators, which can be utilized through a lightweight dynamic translator. This lessens the development costs of the SIMD accelerators and provides binary compatibility across hardware and software generations.

There are two phases necessary in decoupling SIMD accelerators from the processor's instruction set. First, an offline phase takes SIMD instructions and maps them to an equivalent representation. Second, a dynamic translation phase turns the scalar representation back into architecture-specific SIMD equivalents.

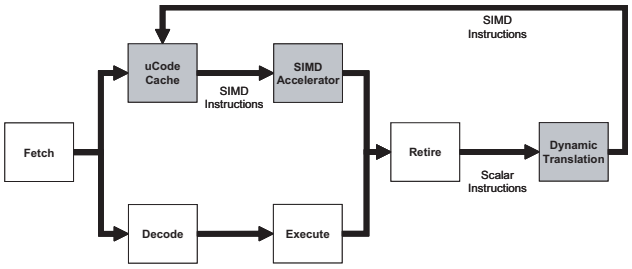

Figure 1. Pipeline organization for Liquid SIMD. Gray boxes represent additions to a basic pipeline.

Converting SIMD instructions into an equivalent scalar representation requires a set of rules that describe the conversion process, analogous to the syntax of a programming language. The conversion can either be done at compile time or by using a post-compilation cross compiler. It is important to note that the SIMD-to-scalar conversion is completely orthogonal to automated SIMDization (i.e., conversion can be done in conjunction with compiler-automated SIMD code or with hand coded assembly). Further, no information is lost during this conversion. The resulting scalar code is functionally equivalent to the input SIMD code, and a dynamic translator is able to recover the SIMD version provided it understands the conversion rules used.

Dynamic translation converts the virtualized SIMD code (i.e., the scalar representation) into processor-specific SIMD instructions. This can be accomplished using binary translation, just-in-time compilation (JITs), or hardware. Offline binary translation is undesirable for three reasons. First, there is a lack of transparency; user or OS intervention is needed to translate the binary. Second, it requires multiple copies of the binary to be kept. Lastly, there is an accountability issue when applications break. Is the application developer or the translator at fault?

JITs or virtual machines are more viable options for dynamic translation. However, in this work we present the design of a dynamic translator using hardware. The main benefit of hardware-based translation over JITs is that it is more efficient than software approaches. This paper shows that the translation hardware is off the processor's critical path and takes less than 0.2 mm^2 of die area. Additionally, hardware translation does not require a separate translation process to share the CPU, which may be unacceptable in embedded systems. Nothing about our virtualization technique precludes software-based translation, though.

The remainder of this paper describes a compiler technique for generating code for an abstracted SIMD interface, coupled with a post-retirement hardware method for dynamic translation. Our high level processor architecture is presented in Figure 1. A basic pipeline is augmented with a SIMD accelerator, post-retirement dynamic translator, and a microcode cache that stores recently translated SIMD instructions. This system provides high-performance for data parallel operations without requiring instruction set modifications or sacrificing binary compatibility.

3 Liquid SIMD Compilation

The purpose of the compiler in the Liquid SIMD framework is to translate SIMD instructions into an equivalent scalar representation. That is, the compiler re-expresses

SIMD instructions using an equivalent set of instructions from the processor's scalar ISA. Since the scalar ISA is Turing-complete, any SIMD instruction can be represented using the scalar ISA. The challenge is finding a representation that is easy to convert back to SIMD and is also relatively efficient in its scalar form.

It is important note that this paper is not proposing any techniques that rely on the compiler to automatically SIMDize a program. While the approach presented could be used in conjunction with automatic SIMDization techniques [6, 13, 20, 21, 31], this is not the main focus of this work. Instead, we focus on how to design a scalar representation of SIMD code, which executes correctly on a baseline processor, and is amenable to runtime translation.

3.1 Hardware and Software Assumptions

Before describing the actual strategy for abstraction, it is important to explicitly state some assumptions about the hardware targeted and applications to be run. First, it is assumed that the targeted SIMD accelerators operate as a separate pipeline. That is, the SIMD accelerator shares an instruction stream and front end with a baseline pipeline, but has separate register files and execution units.

Second, it is assumed that the SIMD accelerator uses a memory-to-memory interface. That is, when executing SIMD instructions, the basic sequence of events is a loop that loads vectors, operates on them, and finally stores the vectors back to memory. In this model, there is no register-to-register communication between the scalar register file and the vector register file, and intermediate data not stored to memory is not accessed by successive loops. The assumption that there is little register-to-register communication is validated by production SIMD accelerators, which usually have either very slow or no direct communication between the two register files. The lack of intermediate data communication between loops is a side-effect of the types of loops being optimized; typically the ideal size of a vector, from the software perspective, is much too large to fit into the hardware vector size. For example, one of the hot loops in 171.swim operates on vectors of size 514. If hardware supported vectors that long, then computed results could be passed between successive loops in a register. Since the results do not fit in hardware, the results have to be passed through memory.

A last assumption is that the application must be compiled to some maximum vectorizable length. That is, even though the binary will be dynamically adjusted based on the vector width supported in the hardware, there is some maximum vector width supported by the binary. The reason for this assumption is due to memory alignment. Most SIMD systems restrict memory accesses to be aligned based on their vector length. To enforce such alignment restrictions, the compiler aligns data based on an assumed maximum width. The binary can be dynamically adjusted to target any width less than the maximum. The trade off here is code size may unnecessarily increase if an accelerator supports narrower widths than the assumed vector size.

Implicit in this alignment restriction is the assumption that targeted accelerators only support execution widths that are a power of 2 (i.e., 2, 4, 8, ...). That is, a binary compiled for maximum vector width of 8 could not (easily) be dynamically translated to run on a 3-wide SIMD accelerator, because data would be aligned at 8 element boundaries in the binary. Assuming SIMD accelerators are power-of-2 widths is certainly valid for the majority of SIMD accelerators in use today.

3.2 Scalar Representation of SIMD Operations

With these assumptions in mind, we now discuss how to convert SIMD instructions into an equivalent scalar representation. The conversion rules are shown in Table 1. This section will walk through the thinking behind these rules, and Section 3.4 will demonstrate the usage of the rules in a detailed example.

The most natural way to express SIMD operations using scalar instructions is by creating a scalar loop that processes one element of the SIMD vector per iteration. Since SIMD accelerators have a memory-memory interface, vector loads can be converted to scalar loads using the loop's induction variable to select a vector element. The size of a vector's elements is derived from the type of scalar load used to read the vector (e.g., load-byte means the vector is composed of 8-bit elements). Similar to memory accesses, data parallel SIMD operations can be represented with one or more scalar instructions that perform the same computation on one element of the vector. Essentially, any data parallel SIMD instruction can be converted to scalar code by operating on one element of the SIMD vector at a time.

If any SIMD operation does not have a scalar equivalent (e.g., many SIMD ISAs but few scalar ISAs support saturating arithmetic), then the scalar equivalent can be constructed using an idiom consisting of multiple scalar instructions. For example, 8-bit saturating addition could be expressed in the ARM scalar ISA as `r1 = add r2, r3; cmp r1, 0xFF; movgt r1, 0xFF`, where the move instruction is predicated on the comparison. Vector masks, or element-specific predication, is another common example of a SIMD instruction that would likely be constructed using idioms. A dynamic translator can recognize that these sequences of scalar instructions represent one SIMD instruction, and no efficiency is lost in the dynamically translated code. Again, the scalar instruction set is Turing-complete, so any data parallel SIMD instruction *can* be represented using scalar instructions. The only downside is potentially less efficient scalar code if no dynamic translator is present in the system.

More complicated SIMD instructions, which operate on all vector elements to produce one result (e.g., max, min, and sum), can be represented using a loop-carried register in the scalar loop. For example, category (4) in Table 1 shows how a vector min can be represented. If the result register is used both as a source and destination operand, and no other operation defines `r1` in the loop, then `r1` will accumulate the minimum of each vector element loaded into `r2`. The dynamic translator can easily keep track of which registers hold loop-carried state, such as `r1` in this example, meaning vector operations that generate a scalar value fit into the Liquid SIMD system.

One difficulty in using a scalar loop representation of SIMD instructions is handling operations that change the order of vector elements. Permutation instructions illustrate this problem well. Suppose a loop is constructed and begins operating on the first element of two SIMD vectors. After several data parallel instructions, a permutation reorders the vector elements. This means that the scalar data that was being operated on in one loop iteration is needed in a different iteration. Likewise, the permutation causes scalar data from future (or past) iterations to be needed in the current iteration.

SIMD Category	Example SIMD Instruction	Scalar Equivalent	Comments
(1) Data parallel; operates on two vectors	`v1 = vadd v2, v3`	`r1 = add r2, r3`	Used for any operation which has an equivalent scalar operation. SIMD operations without a scalar equivalent (e.g., saturating arithmetic) must construct an idiom using multiple instructions.
(2) Data parallel; operates on a vector and a scalar supported constant	`v1 = vand v2, 0xFF`	`r1 = and r1, 0xFF`	Analogous to category (1)
(3) Data parallel; operates on vector and non-scalar supported constant	`v1 = vor v2, 0xFF00FF00`	`r3 = ld [cnst + ind]` `r1 = or r2, r3`	Compiler inserts a read-only array, `cnst`, into the code, which stores the unsupported constant. The array is indexed using the loop's induction variable to retrieve the appropriate portions during each scalar iteration.
(4) Reductions; multiple vector elements used to compute one result	`r1 = vmin v2`	`r1 = min r1, r2`	Loop-carried dependence (`r1`) is used to represent that each element of the vector is used to calculate one result.
(5) Memory accesses	`v1 = vldb [addr]`	`r1 = ldb [addr + ind]`	Induction variable is used to select one vector element to operate on each iteration. Loads are used to identify width of vector elements (e.g., byte or halfword).
(6) Base-plus-displacement memory accesses	`[addr + r1] = vstr v2`	`r3 = add r1, ind` `[addr + r3] = str r2`	Similar to category (5),

(7) Permutations; reorders vector elements	`v2 = vld [addr]`
	`v1 = vbfly v2`
		`r3 = ld [bfly + ind]`
		`r4 = add ind, r3`
		`r1 = ld [addr + r4]`
			Compiler inserts a read-only array, `bfly`, into the code, which stores how elements are reordered. This is used in conjunction with the induction variable to bring in vector elements in a different order. Values stored in `bfly` uniquely identify a permutation.

(8) Permutations; reorders vector elements	`v1 = vbfly v2`
	`[addr] = vstr v1`
		`r3 = ld [bfly + ind]`
		`r4 = add ind, r3`
		`[addr + r4] = str r1`
			Analogous to category (7), but writes elements to memory in a different order, instead of reading them.

Table 1. Rules for translating SIMD instructions into scalar equivalents. Operands beginning with `r` are scalars, operands beginning with `v` are vectors, and `ind` is the loop's induction variable.

To overcome this problem, we propose limiting permutation instructions to only occur at memory boundaries of scalar loops. This allows the reordering to occur by using loads or stores with a combination of the induction variable and some statically defined offset. Essentially, this loads the correct element for each iteration.

The last two rows of Table 1 briefly illustrate how reordering at memory boundaries works. In category (7), a butterfly instruction reorders the elements of `v2`. In order for the scalar loop to operate on the correct element each iteration, the induction variable needs to be modified by an offset, based on what type of permutation is being performed. The compiler creates a read-only array, `bfly`, that holds these offsets. Once the offset is added to the induction variable, the scalar load will bring in the appropriate vector element. A dynamic translator uses the offsets to identify what type of permutation instruction is being executed in the scalar equivalent. Offsets are used, as opposed to absolute numbers, to ensure vector width independence of the scalar representation.

The downside of using offsets to represent permutations is that element reordering operations must occur at scalar loop boundaries using a memory-memory interface. This makes the code inherently less efficient than standard SIMD instruction sets, which can perform this operation in registers.

Using only the rules in Table 1 and simple idiom extensions, we were able to express the vast majority of the ARM Neon SIMD instruction [5] set using the scalar ARM ISA. Neon is a fairly generic SIMD instruction set, meaning the techniques developed here are certainly applicable to a wide variety of other architectures.

3.3 Limitations of the Scalar Representation

Although using this scalar representation has many benefits, there are some drawbacks that must be taken into consideration. The most obvious is that virtualized SIMD code will not be as efficient on scalar processors as code compiled directly for a scalar processor. This is primarily because of the memory-to-memory interface, the lack of loop unrolling, and the use of idioms. Performance overhead is likely to be minimal, though, since vectors in the working set will be cache hits, the loop branch is easy to predict, and the idioms used are likely to be the most efficient scalar implementation of a given computation. Another mitigating factor is that the scalar code can be scheduled at the idiom granularity to make the untranslated code as efficient as possible. As long as the idioms are intact, the dynamic translator will be able to recover the SIMD code.

Another drawback of the proposed virtualization technique is increased register pressure. Register pressure increases because the scalar registers are being used to represent both scalars and vectors in the virtual format. Additionally, temporary registers are needed for some of the proposed idioms. This could potentially cause spill code which degrades performance of both the scalar representation and translated SIMD code. Empirically speaking, register pressure was not a problem in the benchmarks evaluated in this paper.

```
for(i = 0; i < 128; i += 8) {
  for(j = i, n = 0; n < 4; j++, n++) {

    k = j + 4;

    tr = ar[i] * RealOut[k] -
           ai[i] * ImagOut[k];

    RealOut[k] = RealOut[j] - tr;
    RealOut[j] += tr;
  }
}
```

Figure 2. Example FFT loop.

A last limitation is that there are two classes of instructions, from ARM's Neon ISA, which are not handled by the proposed scalar representation. One such instruction is `v1 = VTBL v2, v3`. In the `VTBL` instruction, each element of `v2` contains as an index for an element of `v3` to write into `v1`. For example, if the first element of `v2` was 3, then the third element of `v3` would be written into the first element of `v1`. This is difficult to represent in the proposed scalar representation, because the induction variable offset, which defines what vector elements are needed in the current loop iteration, is not known until runtime. All other permutation instructions in Neon define this offset statically, allowing the compiler to insert a read-only offset array in the code.

The second class of unsupported instructions is interleaved memory accesses. Interleaving provides an efficient way to split one memory access across multiple destination registers, or to write one register value into strided memory locations. This is primarily used to aggregate/disseminate structure fields, which are not consecutive in memory. There is no scalar equivalent for interleaved memory accesses, and equivalent idioms are quite complex.

The performance of certain applications will undoubtedly suffer from not supporting these two classes. None of the benchmarks evaluated utilized these instructions, though, meaning the most important SIMD instructions *are* supported by the proposed scalar representation.

3.4 SIMD to Scalar Example

To illustrate the process of translating from SIMD to the scalar representation, this section walks through an example from the Fast Fourier Transformation (FFT) kernel, shown in Figure 2. There is a nested loop here, where each iteration of the inner loop operates on eight elements of floating point data stored as arrays in memory. This is graphically illustrated in Figure 3. The compiler (or engineer) identifies that these operations are suitable for SIMD optimization and generates vector load instructions for each eight element data segment. The compiler then schedules vector operations for the loaded data so that the entire inner loop may be executed as a small sequence of SIMD operations, shown in Figure 4(A).

Figure 4(B) presents the scalar mapping of the SIMD code from Figures 3 and 4(A). Here, the vector operations of the SIMD loop are converted into a series of sequential operations, and the increment amount of the induction variable is decreased from eight to one, essentially converting each eight element operation into a single scalar operation. The vector load and butterfly instructions in lines `2-5` of the SIMD code are converted into a set of address calculations and load instructions in lines `2-5` of the scalar code. As previously mentioned, SIMD permutation operations are converted into scalar operations by generating a constant array of offset values added to the loop's induction variable. These offsets are stored in the static data segment of the program at the label `bfly`. The value stored at the address `bfly` plus the induction variable value is the offset of the element of the data array to be loaded in the current iteration.

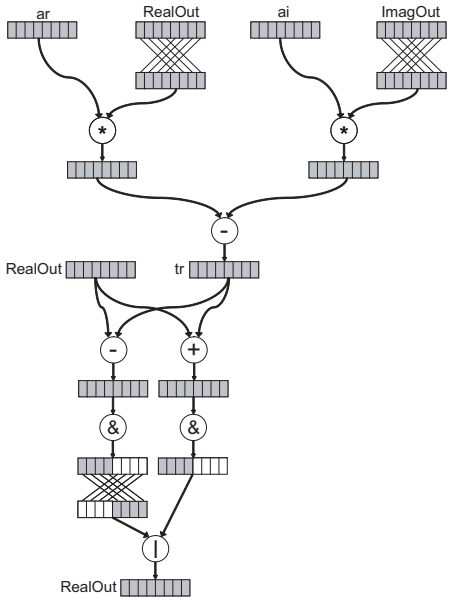

Figure 3. Vector representation of Figure 2.

Most of the vector operations from the SIMD code in lines `6-18` are data parallel, and simply map to their scalar equivalent operation (e.g., the `vmult` on SIMD line 8 is converted to a `mult` on scalar line 8). However, there are a few considerations that need to be made for non-parallel operations. Note that the operation on line `17` of the SIMD code requires that all of the values in `vf3` be computed before the `or` operation, because the `vbfly` operation in line `15` exchanges the position of the first and last vector element. In order to properly transform this code segment into a set of scalar instructions, the loop body for the scalar code must be terminated early, and the operands to the `or` operation must be calculated and stored in a temporary location at the end of each loop iteration, as shown in lines `18-19` of the scalar code. Then, a second loop is created (lines `24-30`) that performs the serial `or` operation across each element of data. By separating scalar equivalents in different loops, the compiler essentially performs a loop fission optimization to ensure that certain SIMD operations are fully completed before others in the next loop are started.

3.5 Function Outlining

Once the SIMD instructions are translated into scalar code, the compiler needs some way to identify to the translator that these portions of code are translatable. This is accomplished by outlining the code segment as a function, similar to the technique proposed in [10]. The scalar equivalent code is surrounded by a branch-and-link and a return instruction so that the dynamic translator is notified that a particular region of code has potential for SIMD optimization.

In the proposed hardware-based translation scheme, when a scalar region is translated into SIMD instructions, the SIMD code is stored in the microcode cache (see Figure 1), and the branch-and-link is marked in a table in the proces-

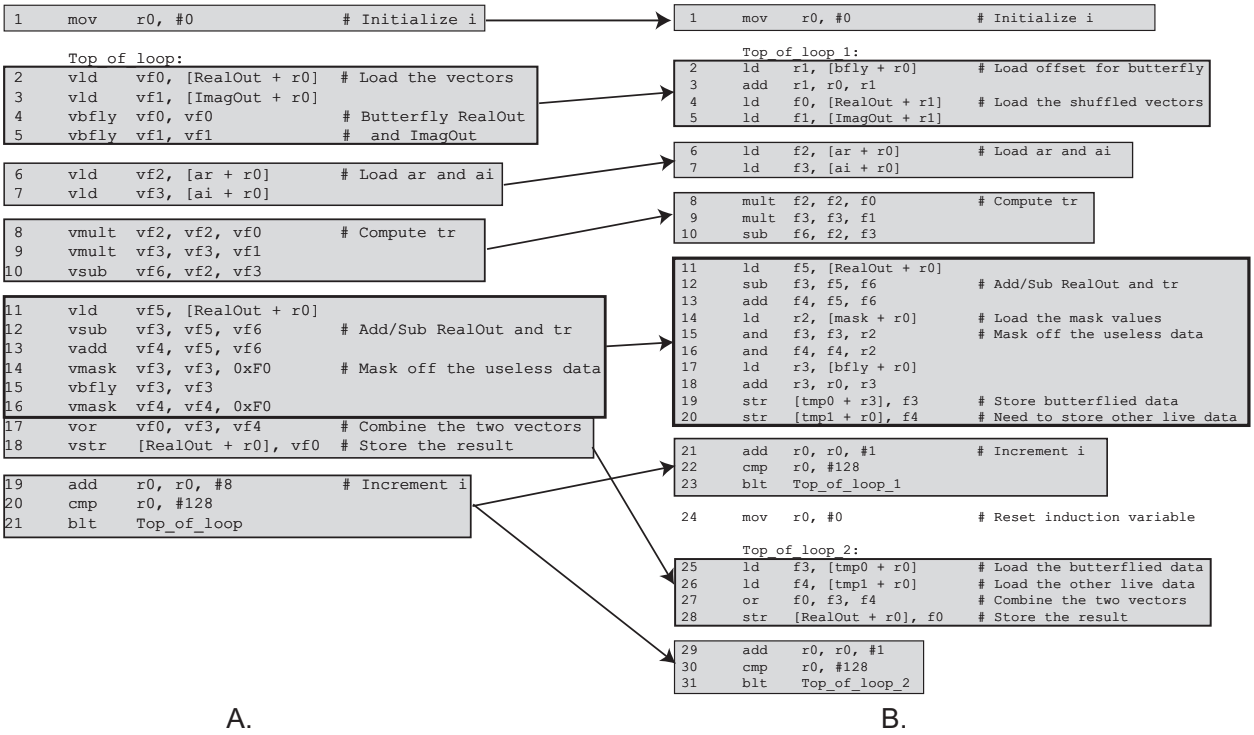

Figure 4. (A) SIMD code for Figure 2, and (B) scalar representation of the SIMD code in Figure 4(A).

sor's front end. The next time this branch is encountered, the front end can utilize the SIMD accelerator by simply accessing the SIMD instructions in the microcode cache and ignoring the branch. This allows a processor to take advantage of SIMD accelerators without explicit instruction set modifications.

One potential problem with marking translatable code regions by function calls is false positives. This happens if the dynamic translator creates SIMD code for a function that was not meant to be SIMDized. Typically, this is not a problem. ABIs require that functions have a very specific format, which does not match the outlined function format described for scalarized loops. Therefore, the dynamic translator would not be able to convert most non-translatable functions. Even if the translator was able to convert a function that it was not meant to, the SIMD code would be functionally correct as long as there were no memory dependences between scalar loop iterations. Remember, the translator is simply converting between functionally equivalent representations. The scenario of a false positive that produces incorrect code is highly unlikely, but the only way to guarantee correctness is to mark the outlined functions in some unique way (e.g., a new branch-and-link instruction that is only used for translatable regions).

4 Dynamic Translation to SIMD Instructions

Once a software abstraction is defined for describing SIMD instructions using a scalar ISA, there needs to be a runtime method for translating them back into SIMD instructions. As mentioned in Section 2, there are many valid ways to do this: in hardware at decode time, in hardware after instruction retirement, or through virtual machines or JITs. The software abstraction presented in the previous section is independent of the translation scheme.

Here, the design of a post-retirement hardware translator is presented. Hardware was chosen because the implementation is simple, it adds little overhead to the baseline processor, and hardware is more efficient than software. Post-retirement hardware was chosen, instead of decode time, because post-retirement is far off the critical path of the processor. Our experiments in Section 5 and previous work [15] both show that post-retirement optimizations can be hundreds of cycles long without significantly affecting performance. The biggest downside to a post-retirement dynamic mapping is that the modified microcode needs to be stored in a cache and inserted into the control stream in the pipeline frontend.

4.1 Dynamic Translation Hardware

From a high level, the translator is essentially *a hardware realization of a deterministic finite automaton that recognizes patterns of scalar instructions to be transformed into SIMD equivalents.* Developing automata (or state machines) to recognize patterns, such as the patterns in Table 1, is a mature area of compiler research. A thorough discussion of how to construct such an automata is described in [1].

The structure of the proposed post-retirement dynamic translator is shown in Figure 5. To prove the practicality of this structure, it was implemented in HDL (targeting the ARM ISA with Neon SIMD extensions) and synthesized using a 90 nm IBM standard cell process. The results of the synthesis are shown in Table 2. Notice that the control generator runs at over 650 MHz, and takes up only 174,000 cells (less than 0.2 mm^2 in 90 nm), without using any custom logic. This shows that the hardware impact of the control

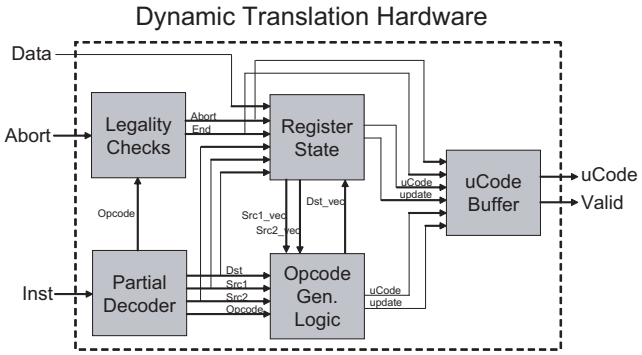

Figure 5. Structure of the proposed translator.

Description	Crit. Path	Delay	Area
8-wide Translator	16 gates	1.51 ns	174,117 cells

Table 2. Synthesis results for the dynamic translator.

generator is well within the reach of many modern architectures.

Partial Decoder: The dynamic translator has three inputs from retirement of the baseline pipeline: the instruction that retires (`Inst` in the figure), the data value that instruction generated (`Data`), and an abort signal (`Abort`). Initially, the retired instruction is fed into a partial decoder to determine the source/destination operands and the opcode. It is only a partial decoder, because it only needs to recognize opcodes that are translatable; any other opcodes simply cause translation to abort mapping of the outlined function. This portion of the control generator is potentially redundant, depending on the microarchitecture, because the retiring instruction will likely have the opcode and operand information stored in its pipeline latch. Overall, the partial decoder only takes a few thousand cells of die area, so it does not contribute significantly to the area overhead; it is responsible for 5 of the 16 gates in the critical path, though.

Legality Checks: The purpose of the legality checker in the dynamic translator is to monitor the incoming instructions to ensure that they can be translated. Scalar instructions that do not map to a SIMD equivalent generate an abort signal that flushes stateful portions of dynamic translator. In addition to an instruction generated abort signal, there is an abort signal from the base pipeline to stop translation in the event of a context switch or other interrupt. The legality checker also signals when a subgraph has finished mapping, enabling the microcode buffer to write the translated SIMD instructions into the microcode cache. The legality checks only comprise a few hundred cells and do not occur on the critical path.

Register State: After the instruction is decoded, the operands/opcode access some state, which is indexed based on the register numbers. This register state determines the translation strategy for this instruction. Register state also includes whether or not a register represents a scalar or vector, the size of the data currently assigned to the register (e.g., 16 or 32 bit), and previous values stored in the register. The opcode and register state comprise the data used to transition between states in the automata.

Overall, there are 56 bits of state per register and a large number of MUXes in the register state module, making this structure comprise 55% of the control generator die area. Since the ARM ISA only has 16 architectural integer registers, 55% of the die area is likely proportionally smaller than dynamic translators targeting architectures with more registers. Additionally, this structure will increase in area linearly with the vector lengths of the targeted accelerator.

The previous values assigned to each register are stored in the register state in order to identify operations that are defined using offsets in memory (e.g., the butterfly instruction discussed in Section 3). Recall that instructions that reorder elements within a vector are encoded by loading an offset vector, adding the offsets to the induction variable, and using that result for a memory access. In the dynamic translator, load instructions cause the data to be written to the destination register's state. When a data processing instruction uses that destination register as a source operand, (e.g., to add those values to the induction variable), then the previous values of the address are copied to the data processing instruction's destination register state. When a memory access instruction uses a source that has previous values recorded in the register state, this signals that a shuffle may be occurring. Those previous values (i.e., the offset vector) are used to index a content addressable memory (CAM), and if there is a hit, the appropriate shuffle is inserted into the SIMD instruction stream. If the CAM misses, then the offset being loaded is a shuffle not supported in the SIMD accelerator and translation is aborted. Note that storing the entire 32 bits of previous values is unnecessary, because the values are only used to determine valid constants, masks, and permutation offsets; numbers that are too big to represent simply abort the translation process. The process of reading a source register's previous values, and conditionally writing them to the destination register, accounts for 11 of the 16 gates on the critical path.

Opcode Generation Logic: Once register state for an instruction's source operands has been accessed, it is passed to the opcode generation logic. Opcode generation logic uses simple combinational logic to determine how to modify an opcode based on the operands. This essentially performs the reverse of the mapping described in Section 3, using rules defined in Table 3. For example, if the incoming instruction is a scalar load, then the opcode logic will write a vector load into the microcode buffer and tell the register state to mark the destination as a vector. Likewise, if the incoming instruction is an add, and the register state says both source registers are vectors, opcode generation logic will write a vector add into the microcode buffer and mark the destination register as a vector. A small amount of state is kept alongside this logic to recognize idioms of scalar instructions. Whenever an idiom is detected, this logic has the ability to invalidate previously generated instructions in the microcode buffer.

Opcode generation logic is fairly simple provided the SIMD instruction format is similar to the equivalent scalar instructions, since the scalar instructions require little modification before insertion into the microcode buffer. This is the case with our implementation, and thus the logic only takes up approximately 9000 cells. Control generation is not on the critical path in the current implementation, but it is very close to being critical. It likely would be on the critical path if there was not good correlation between baseline and SIMD instruction formats.

Microcode Buffer: The final component of the dynamic translator is the microcode buffer. This is primarily just a register array used to store the SIMD instructions until a region of scalar code has completed mapping. The maximum

Scalar Instruction	Current Register State	Updated Register State	Instruction(s) Generated
(1) r1 = mov #const		r1 is marked as the induction variable	r1 = mov #const
(2) r1 = ld [r2 + r3]	r2 is a scalar; r3 is the induction variable	r1 is a vector; size of r1 is recorded (i.e., byte, halfword, etc.); value loaded is stored in r1	v1 = vld [r2 + r3]
(3) r1 = ld [r2 + r3]	r2 is a scalar; r3 is a vector; r3 has values loaded into it from an offset array	r1 is a vector; size of r1 is recorded	v1 = vld [r2 + ind] v1 = vpermute v1
(4) [r1 + r2] = str r3	r1 is a scalar; r2 is the induction variable		[r1 + r2] = vstr v3
(5) [r1 + r2] = str r3	r1 is a scalar; r2 is a vector; r2 has values loaded into it from an offset array		v3 = vpermute v3 [r1 + r2] = vstr v3
(6) r1 = dp r2, r3	r2 is a vector; r3 is a vector	r1 is a vector; size of r1 is recorded	v1 = vdp v2, v3
(7) r1 = dp r2, r3	r2 is a vector; r3 is a vector; r3 has values loaded into it	r1 is a vector; size of r1 is recorded	v1 = vdp v2, #const
(8) r1 = dp r2, r3	r2 is a vector; r3 is the induction variable (or vice-versa); r2 has values loaded into it	r1 is a vector; values loaded into r2 are copied to r1	None: this format is only used to update the induction variable for permutations.
(9) r1 = dp r1, r2	r1 is a scalar; r2 is a vector	r1 is a scalar	r1 = vred v2
(10) r1 = add r1, #1	r1 is the induction variable		r1 = add r1, SIMD_width
(11) any other instruction	all source operands are scalar		The input instruction is passed unmodified

Table 3. Rules used to dynamically translate the scalar code to SIMD code. `dp` **refers to any data processing opcode, and** `vred` **refers to a vector opcode that reduces a vector to one scalar result (e.g., min).**

length of a microcode sequence was limited to 64 instructions in this implementation. Section 5 shows that this is sufficient for the benchmarks examined. At 32 bits per instruction, the microcode buffer contains 256 bytes of memory, which accounts for a little more than half of its 77,000 cells of die area. The rest of the area is consumed by an alignment network for collapsing instructions when idioms or permutations invalidate previously generated instructions.

Recall that the register state is used to detect when memory operations are indexed using a previously loaded offsets from constant arrays (Categories (7) and (8) in Table 1). When this situation is detected, the opcode generation logic will insert the appropriate permutation and memory instructions. At this point, the previously generated vector load of the offset vector can safely be removed. Removing this instruction while inserting multiple other instructions requires an alignment network. It should be noted that removing the offset load is not strictly necessary for correctness, and eliminating this functionality would greatly simplify the microcode buffer.

After the microcode buffer receives the End signal from the legality checker, SIMD instructions are written into the microcode cache. SIMD code will then be inserted into the pipeline upon subsequent executions of the outlined function.

4.2 Dynamic Translation Example

To better illustrate how the dynamic translation hardware functions, Table 4 shows an example, translating the scalar loop in Figure 4(B) back into SIMD instructions for an 8-wide SIMD accelerator. The second loop from Figure 4(B) would be translated in a similar manner, and not refused with the original fissioned loop. Translation is very straightforward for the vast majority of opcodes in the example, making the design of a hardware dynamic translator simple.

Instruction 1, the move, is the first instruction to enter the dynamic translator. As per the rules in Table 3, `r0` is marked as the induction variable in the register state, and the instruction is inserted into the microcode buffer unmodified.

Next, instruction 2 is translated. This is a load based on a scalar (the address `bfly`) and the induction variable (`r0`). Table 3 shows this is translated into a standard vector load. `R1` is marked as a vector and the value loaded is stored as a previous value of `r1` in the register state. After that, instruction 3 is translated. The register state shows that `r0` is the induction variable and `r1` is a vector with previous values associated with it. This instruction generates no instruction.

Now instruction 4 needs to be translated. Since one of the sources, `r1`, has previous values associated with it, this load may correspond to a shuffle instruction. The register state will look at the previous values, use them to CAM into a ROM and see that these offsets correspond to a known permutation instruction. In parallel, the load is being turned into a vector load by the opcode generation logic. Both of these instructions are inserted into the microcode buffer. Additionally, a pointer from the register state is used to remove the vector load created for instruction 2; a load of the offset is not necessary once the butterfly is inserted. This process of creating a load and shuffle is repeated for instruction 5.

Translating the remaining instructions in this example is just a matter of applying the rules presented in Table 3. Any

	Scalar Instruction	SIMD Generated
1	mov r0, #0	mov r0, #0
2	ld r1, [bfly + r0]	v1 = vld [bfly + r0]
3	add r1, r0, r1	
4	ld f0, [RealOut + r1]	vf0 = vfld [RealOut + r0] vf0 = vbfly vf0
5	ld f1, [ImagOut + r1]	vf1 = vfld [ImagOut + r0] vf1 = vbfly vf1
6	ld f2, [ar + r0]	vf2 = vfld [ar + r0]
7	ld f3, [ai + r0]	vf3 = vfld [ai + r0]
8	mult f2, f2, f0	vf2 = vmult vf2, vf0
9	mult f3, f3, f1	vf3 = vmult vf3, vf1
10	sub f6, f2, f3	vf6 = vsub vf2, vf3
11	ld f5, [RealOut + r0]	vf5 = vld [RealOut + r0]
12	sub f3, f5, f6	vf3 = vsub vf5, vf6
13	add f4, f5, f6	vf4 = vadd vf5, vf6
14	ld r2, [mask + r0]	v2 = vld [mask + r0]
15	and f3, f3, r2	vf3 = vmask vf3, #const
16	and f4, f4, r2	vf4 = vmask vf4, #const
17	ld r3, [bfly + r0]	v3 = vld [bfly + r0]
18	add r3, r0, r3	
19	str [tmp0 + r3], f3	vf3 = vbfly vf3 [tmp0 + r0] = vstr vf3
20	str [tmp1 + r0], f4	vf4 = vbfly vf4 [tmp1 + r0] = vstr vf4
21	add r0, r0, #1	r0 = add r0, #8
22	cmp r0, #128	cmp r0, #128
23	blt Top_of_loop_1	blt Top_of_loop_1

Table 4. Example translating scalar representation from Figure 4(B) back into SIMD instructions.

instruction that does not match the rules defined in that table does not meet the proposed scalar virtualization format, and causes translation to abort. Once all scalar instructions have been translated, the outlined function returns, and the microcode buffer writes the SIMD instructions into the microcode cache. This enables the SIMD code to be inserted into the instruction stream upon subsequent encounters of the outlined function.

5 Evaluation

To evaluate the Liquid SIMD system, an experimental framework was built using the Trimaran research compiler [29] and the SimpleScalar ARM simulator [3]. Trimaran was retargeted for the ARM instruction set, and was used to compile scalar ARM assembly code. The ARM assembly code was then hand-modified to include SIMD optimizations and conversion to the proposed scalar representation using a maximum targeted SIMD width of 16. Automatic SIMDization would have been used had it been implemented in our compiler. Again, automatic SIMDization is an orthogonal issue to abstracting SIMD instruction sets.

In our evaluation, SimpleScalar was configured to model an ARM-926EJ-S [2], which is an in-order, five stage pipelined processor with 16K, 64-way associative instruction and data caches. A parameterized SIMD accelerator, executing the Neon ISA, was added to the ARM-926EJ-S SimpleScalar model to evaluate the performance of SIMD accelerators for various vector widths. Simulations assumed dynamic translation took one cycle per scalar instruction in an outlined function. However, we demonstrate that dynamic translation could have taken tens of cycles per scalar instruction without affecting performance.

Liquid SIMD was evaluated using fifteen benchmarks from SPECfp2000 (171.swim, 179.art, 172.mgrid), SPECfp95 (101.tomcatv, 104.hydro2d), SPECfp92 (052.alvinn, 056.ear, 093.nasa7), MediaBench (GSM Decode and Encode, MPEG2 Decode and Encode), and common signal processing kernels (FFT, LU, FIR). The set of benchmarks evaluated was limited by applicability for SIMD optimization and the current capability of the ARM port of our compiler. None of these limitations were a result of the Liquid SIMD technique.

Benchmark	Mean	Max
052.alvinn	12.5	13
056.ear	34.5	36
093.nasa7	45.5	59
101.tomcatv	35.5	61
104.hydro2d	27.2	40
171.swim	37.8	51
172.mgrid	46.2	62
179.art	12.8	19
MPEG2 Dec.	12.5	13
MPEG2 Enc.	14.5	19
GSM Dec.	25	25
GSM Enc.	19.5	28
LU	11	11
FIR	11	11
FFT	31.3	38

Table 5. Number of scalar instructions in outlined function(s).

Dynamic Translation Requirements: In order to further understand the costs of Liquid SIMD, we first studied characteristics of benchmarks that impact design of a dynamic translator. One such characteristic is the required size of the microcode cache. The microcode cache is used to store the SIMD instructions after an outlined procedure call has been translated. This characteristic is also important for software-based translators, as it affects the size of code cache needed for the application.

We found that supporting eight or more SIMD code sequences (i.e., hot loops) in the control cache is sufficient to capture the working set in all of the benchmarks investigated. One question remaining then is how many instructions are required for each of these loops. With a larger control cache entry size, larger loops may be translated, ultimately providing better application performance. The downside is increased area, energy consumption, and latency of the translator. However, large loops that do not fit into a single control cache entry may be broken up into a series of smaller loops, which do fit into control cache. The downside of breaking loops is that there will be increased procedure call overhead in the scalarized representation. This section later demonstrates that procedure call overhead is negligible when using an 8-entry control cache.

Table 5 presents the average and maximum number of instructions per hot loop in the benchmarks. In some benchmarks, like 172.mgrid and 101.tomcatv, hot loops in the Trimaran-generated assembly code consisted of more than 64 instructions, and were broken into two or more loops. This decreased the number of instructions in each loop dramatically because it also reduced the number of load and store instructions caused due to register spills. Table 5 shows that 172.mgrid and 101.tomcatv have the largest outlined functions with a maximum of nearly 64 instructions. In most of these benchmarks, it would be possible to decrease the number of instructions per loop to less than 32 in order to decrease the size of the microcode cache.

Benchmark	< 150	< 300	> 300	Mean
052.alvinn	0	0	2	19984
056.ear	0	0	3	96488
093.nasa7	0	0	12	23876
101.tomcatv	0	0	6	16036
104.hydro2d	0	0	18	24346
171.swim	0	0	9	33258
172.mgrid	0	0	13	5218
179.art	0	0	5	2102224
MPEG2 Dec.	0	1	1	269
MPEG2 Enc.	0	3	1	257
GSM Dec.	0	0	1	358
GSM Enc.	0	0	1	538
LU	0	0	1	15054
FIR	0	0	1	13343
FFT	0	0	3	7716

Table 6. Number of cycles between the first two consecutive calls to outlined hot loops. The first three columns show the number of outlined hot loops that have distance of less than 150, less than 300, and greater than 300 cycles between their first two consecutive calls.

These results lead us to propose a control cache with 8 entries of 64 SIMD instructions each. Assuming each instruction is 32 bits, this would total a 2 KB SRAM used for storing translated instruction sequences.

Another benchmark characteristic that affects dynamic translator design is latency between two executions of hot loops. Translation begins generating SIMD instructions for outlined scalar code the first time that a code segment is executed. If translation takes a long time, then SIMD instructions might not be available for many subsequent executions of that hot loop. This restricts the performance improvement achievable from a Liquid SIMD system. Moreover, if translation takes a long time, then the dynamic translator will need some mechanism to translate multiple loops at the same time.

Table 6 shows the number of cycles between the two first consecutive calls to outlined hot loops for the benchmarks. In all benchmarks except MPEG2 Encode and Decode, there is more than 300 cycles distance between outlined procedure calls. The reason for large distances is that the scalar loops usually iterate several times over dozens of instructions, and also because memory accesses tend to produce cold cache misses. Table 6 shows that there is significant time for hardware based dynamic translation to operate without adversely affecting performance. A carefully designed JIT translator would likely be able to meet this 300 cycle target, as well.

Performance Overhead from Translation: Figure 6 illustrates the speedup attained using one Liquid SIMD binary (per benchmark) on machines supporting different width SIMD accelerators. Speedup reported is relative to the same benchmark running on a ARM-926EJ-S processor without a SIMD accelerator and without outlining hot loops. Compiling with outlined functions would have added a small overhead (less than 1%) to the baseline results.

In the ideal case, a SIMD-enabled processor with unlimited resources can achieve a speedup of $\frac{1}{\frac{S}{W}+(1-S)}$, where S is SIMD optimizable fraction of the code and W is the accelerator vector width. Some of the factors that decrease the amount of speedup in real situations are cache miss penalties, branch miss predictions, and trip count of the hot loop.

As expected, speedup generally increases by increasing the vector width supported in the SIMD hardware. In some of the benchmarks, like MPEG2 Decode, there is virtually no performance gain by increasing the vector width from 8 to 16. This is because the hot loop(s) in these benchmarks operate on vectors that are only 8 elements. Supporting larger vector widths is not beneficial for these applications. 179.art shows the least speedup of any of the benchmarks run. In this case, speedup is limited because 179.art has many cache misses in its hot loops. FIR showed the highest speedup of any benchmark because approximately 94% of its runtime is taken by the hot loop, the loop is fully vectorizable, and there are very few cache misses.

Figure 6 shows that SIMD acceleration is very effective for certain benchmarks. However, this fact has been well established and is not the purpose of this paper. The main purpose of Figure 6 is to demonstrate the performance overhead of using dynamic translation in a Liquid SIMD system. Overhead stems from executing SIMD loops in their scalar representation whenever the SIMD version does not reside in the microcode cache. To evaluate the overhead, the simulator was modified to eliminate control generation. That is, whenever an outlined function was encountered, the simulator treated it like native SIMD code.

The performance improvement from using native instructions was measured for all fifteen benchmarks. Of these benchmarks, the largest performance difference occurred in FIR, illustrated in the callout of Figure 6. Native SIMD code provided 0.001 speedup above the Liquid SIMD binary. This demonstrates that the performance overhead from virtualization is negligible.

Code Size Overhead: Compilation for Liquid SIMD does increase the code size of applications. Code size overhead comes from additional branch-and-link and return instructions used in function outlining, converting SIMD instructions to scalar idioms, and also from aligning memory references to a maximum vectorizable length (discussed in Section 3). Obviously, too much code size expansion will be problematic, creating instruction cache misses, which may affect performance.

To evaluate code size overhead, the binary sizes of unmodified benchmarks were compared with Liquid SIMD versions. The maximum difference observed occurred in hydro2d, and was less than 1%. The reason behind this is that the amount of SIMD code in the benchmarks is very small compared to the overall program size. Code size overhead is essentially negligible in Liquid SIMD.

6 Related Work

Many different types of accelerators have been proposed to make computation faster and more efficient in microprocessors. Typically, these accelerators are utilized by changing the instruction set; that is, statically placing accelerator control in the application binary. This means that the binary will not run on systems without that accelerator, or even systems where the accelerator has changed slightly.

To allow more flexibility in the instruction set, some previous work [9, 10, 19, 24, 28, 32] has recognized the benefits of dynamically binding instructions to an accelerator. Many different methods have been proposed to generate microcode for the various targeted accelerators at runtime. For example, work by Hu [18, 19] demonstrated the effectiveness of using binary translation software to dynamically generate control for one type of accelerator, a 3-1 ALU. The rest of these techniques utilize trace cache based hardware structures, to

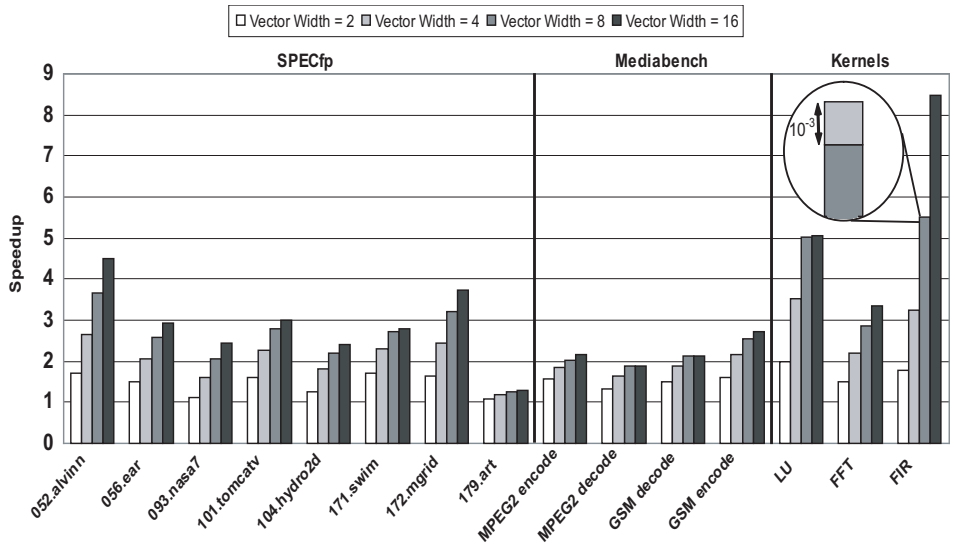

Figure 6. Speedup for different vector widths relative to a processor without SIMD acceleration. The callout shows the speedup improvement for a processor with built-in ISA support for SIMD instructions.

perform translation. Our method evolves this approach for SIMD accelerators.

There is a great deal more related work if the scope of dynamic binding is expanded to include benefits other than accelerator utilization. Dynamic binding has long been used to support modern microarchitectures in the context of legacy ISAs, such as the use of micro-ops (including micro-op fusing) in Intel processors [16]. Another motivation for dynamic binding has been to enable runtime optimizations. Several standard compiler optimizations, such as dead code elimination and constant propagation, benefit from runtime information available to dynamic translators [17].

Continuous Optimization [14] and RENO [27] are both examples of dynamic translators that perform traditional compiler optimizations by translating instructions during the decode stage of pipelines. The rePLay [26] project similarly optimized code, but operated on instructions post-retirement. Post-retirement translation is attractive because there is usually a long latency between instruction retirement and its next use [15], effectively taking translation off the critical path.

Just in time compilers (JITs) and virtual machines, such as Dynamo [4], DAISY [12], and the Transmeta Code Morpher [11], are all examples software-only dynamic translators. Software dynamic translators have been proposed both for code optimizations and to translate one ISA to another.

Virtualizing a SIMD ISA is similar to the way modern graphics related shader applications [7] are executed. In these applications, pixel and vertex shaders are distributed in an assembly-like virtual language such as DirectX, which has support for SIMD. At runtime, the shaders rely on a virtual machine to translate the virtual SIMD instructions into architecture-specific SIMD instructions. The benefits of using scalar instructions to virtualize SIMD instructions, as opposed a virtual language, is that a translator is not necessary to run the application.

The hardware translator proposed in this paper is closely related to two other works [25, 30]. These papers developed methods to utilize SIMD hardware dynamically, without software support for identifying the instructions. That is, these works (often speculatively) create SIMD instructions from an arbitrary scalar binary. The hardware support required to perform this translation is generally more complicated than our proposed design, which merely recognizes and translates a set of predetermined instruction patterns.

The proposed hardware translator is also similar to work by Brooks [8] and Loh [23]. These papers propose using dynamic translation to detect when operations do not use the entire data path (e.g., only 8-bits of a 32-bit ALU), and then pack multiple narrow operations onto a single function unit.

Somewhat related to this paper are the decades of research that have gone into automated compiler-based SIMDization. Many of these techniques are summarized by Krall [20] for the UltraSparc VIS instruction set, and by Bik [6] for Intel's SSE instructions. Recent work [13, 31] has investigated techniques to vectorize misaligned memory references through data reorganization in registers. Other recent work [21] introduced techniques to extract vector operations within basic blocks and selective vectorization of instructions. Automatic SIMDization is completely orthogonal to the work in this paper; the SIMD virtualization scheme proposed here can be used in conjunction with or in the absence of any automated SIMD techniques.

The main contribution of this work is the development of a method for virtualizing SIMD instructions in a way amenable to dynamic translation. No previous work has done this. To demonstrate that our virtualization schema is easily translated, the design of a post-retirement hardware translator was presented in Section 4. Any other style of dynamic translator could have been used to prove this point, though.

7 Conclusion

Liquid SIMD is a combination of compiler support and dynamic translation used to decouple the instruction set of a processor from the implementation of a SIMD accelerator. SIMD instructions are identified and expressed in a virtualized SIMD schema using the scalar instruction set of a processor. A light-weight dynamic translation engine binds

these scalar instructions for execution on an arbitrary SIMD accelerator during program execution. This eliminates the problems of binary compatibility and software migration that are inherent to instruction set modification.

This paper presented a software schema powerful enough to virtualize nearly all SIMD instructions in the ARM Neon ISA using the scalar ARM instruction set. The design of a hardware dynamic translator was presented, proving that the software schema is translatable and that this translation can be incorporated into modern processor pipelines. Synthesis results show that the design has a critical path length of 16 gates and the area is less than 0.2 mm^2 in a 90 nm process. Experiments showed that Liquid SIMD caused code size overhead of less than 1%, and performance overhead of less that 0.001% in the worst case. This data clearly demonstrates that Liquid SIMD is both practical and effective at solving the compatibility and migration issues associated with supporting multiple SIMD accelerators in a modern instruction set.

8. Acknowledgments

Much gratitude goes to the anonymous referees who provided excellent feedback on this work. We owe thanks to Jason Blome and Mike Chu, who went to great lengths in assisting with preliminary drafts of this work, and also to Rodric Rabbah and Sam Larsen, who provided help with our evaluation infrastructure. This research was supported by ARM Limited, the National Science Foundation grant CCF-0347411, and equipment donated by Hewlett-Packard and Intel Corporation.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, Jan. 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[5] M. Baron. Cortex-A8: High speed, low power. *Microprocessor Report*, 11(14):1–6, 2005.

[6] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[7] M. Breternitz, H. Hum, and S. Kumar. Compilation, Architectural Support, and Evaluation of SIMD Graphics Pipeline Programs on a General-Purpose CPU. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 135–145, 2003.

[8] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proc. of the 5th International Symposium on High-Performance Computer Architecture*, page 13, 1999.

[9] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[10] N. Clark et al. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 272–283, June 2005.

[11] J. Dehnert et al. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.

[12] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 26–38, June 1997.

[13] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *Proc. of the SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 82–93, 2004.

[14] B. Fahs, T. Rafacz, S. Patel, and S. Lumetta. Continuous optimization. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 86–97. IEEE Computer Society, 2005.

[15] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[16] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):http://developer.intel.com/technology/itj/2003/volume07 issue02/, 2003.

[17] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[18] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar cisc processors. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 213–226, 2006.

[19] S. Hu and J. E. Smith. Using dynamic binary translation to fuse dependent instructions. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 213–226, 2004.

[20] A. Krall and S. Lelait. Compilation techniques for multimedia processors. *International Journal of Parallel Programming*, 28(4):347–361, 2000.

[21] S. Larsen, R. Rabbah, and S. Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 119–129, 2005.

[22] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006.

[23] G. H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proc. of the 35th Annual International Symposium on Microarchitecture*, pages 395–405, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[24] R. Nair and M. Hopkins. Exploiting instruction level parallelism in processors by caching scheduled groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, pages 13–25, June 1997.

[25] A. Pajuelo, A. Gonzlez, and M. Valero. Speculative dynamic vectorization. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 271–280, 2002.

[26] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[27] V. Petric, T. Sha, and A. Roth. Reno: A rename-based instruction optimizer. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 98–109, 2005.

[28] P. Sassone, D. S. Wills, and G. Loh. Static strands: safely collapsing dependence chains for increasing embedded power efficiency. In *Proc. of the 2005 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 127–136, June 2005.

[29] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[30] S. Vajapeyam, P. J. Joseph, and T. Mitra. Dynamic vectorization: A mechanism for exploiting far-flung ILP in ordinary programs. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 16–27, 1999.

[31] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 153–164, 2005.

[32] S. Yehia and O. Temam. From sequences of dependent instructions to functions: An approach for improving performance without ILP or speculation. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 238–249, June 2004.