From Relational Verification to SIMD Loop Synthesis

Gilles Barthe[1] Juan Manuel Crespo[1] Sumit Gulwani[2] César Kunz[1,3] Mark Marron[1]

[1]IMDEA Software Institute, [2]Microsoft Research, [3]Technical University of Madrid

{gilles.barthe, juanmanuel.crespo, cesar.kunz, mark.marron}@imdea.org, sumitg@microsoft.com

Abstract

Existing pattern-based compiler technology is unable to effectively exploit the full potential of SIMD architectures. We present a new program synthesis based technique for auto-vectorizing performance critical innermost loops. Our synthesis technique is applicable to a wide range of loops, consistently produces performant SIMD code, and generates correctness proofs for the output code. The synthesis technique, which leverages existing work on relational verification methods, is a novel combination of deductive loop restructuring, synthesis condition generation and a new inductive synthesis algorithm for producing loop-free code fragments. The inductive synthesis algorithm wraps an optimized depth-first exploration of code sequences inside a CEGIS loop. Our technique is able to quickly produce SIMD implementations (up to 9 instructions in 0.12 seconds) for a wide range of fundamental looping structures. The resulting SIMD implementations outperform the original loops by 2.0×-3.7×.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors-Optimization; C.1.1 [*Single Data Stream Architectures*]: VLIW architectures

Keywords Program Vectorization, Program Synthesis, Deductive Synthesis, Inductive Synthesis, Relational Program Verification

1. Introduction

Single Instruction Multiple Data (SIMD) instructions sets (such as SSE on x86 or NEON on ARM) provide high throughput and power efficient data-parallel operations. These operations can process 128 bits in a single instruction and can often do so in the same number of cycles (and power usage) needed to process a single 32 bit value via the standard ALU execution path. These features have proven invaluable in accelerating multimedia and high performance computing applications, and are critical to achieving both good application performance and battery life in many mobile computing environments. Despite these advantages and their proven value in practice, the use of SIMD operations has been limited to a relatively small set of (often hand optimized) applications. Extending these benefits to a wider range of programs via automatic compiler vectorization has, in practice, been limited by three major challenges: the presence of pointers, sub-optimal data layout, and complex data driven control flow. In this paper we explore a new approach to auto-vectorization that is intended to address the last two of these challenges. This approach allows us to produce efficient SIMD implementations for many loops that are present in foundational libraries such as the STL for C++ or the BCL for C#.

PPoPP'13, February 23–27, 2013, Shenzhen, China.


```
//Simple widget struct with a tag and a score value
struct { int tag; int score; } widget;

int exists(widget* vals, int len, int tv, int sv) {
  for (int i = 0; i < len; ++i) {
    int tagok = vals[i].tag == tv;
    int scoreok = vals[i].score > sv;
    int andok = tagok & scoreok;
    if (andok) return 1;
  }
  return 0;
}
```

Figure 1. Initial Loop.

```
int exists_sse(widget* vals, int len, int tv, int sv) {
  m128i vectv = [tv, tv, tv, tv];
  m128i vecsv = [sv, sv, sv, sv];

  int i = 0;
  for (; i < (len - 3); i += 4) {
    m128i blck1i = load_128(vals + i);
    m128i blck2i = load_128(vals + i + 2);

    int tvswizzle = SHF_ORDER(0, 2, 0, 2);
    int svswizzle = SHF_ORDER(1, 3, 1, 3);

    m128i tagvs = shuffle_i32(blck1i, blck2i, tvswizzle);
    m128i scorevs = shuffle_i32(blck1i, blck2i, svswizzle);

    m128i cmprl = cmpeq_i32(vectv, tagvs);
    m128i cmprh = cmpgt_i32(vecsv, scorevs);
    m128i cmpr = and_i128(cmprl, cmprh);

    int match = !allzeros(cmpr);
    if (match) return 1;
  }

  for (; i < len; i++) {
    int tagok = vals[i].tag == tv;
    int scoreok = vals[i].score > sv;
    if (tagok & scoreok) return 1;
  }
  return 0;
}
```

Figure 2. SIMD Implementation.

Motivating Example. Consider the program fragment in Figure 1, which consists of a loop that traverses an array of `widget` structs (of length `len`). The loop body checks if the values in the `tag` and `score` fields satisfy certain properties and if so returns 1 immediately. If no such `widget` is found then 0 is returned.

$$
\begin{array}{rrcl}
(variables) & & & a : \text{Array} \mid i, x : \text{Int32} \mid s : \text{Struct} \mid \mathbf{v} : \mathbf{Vector} \\
(fields) & & & f \in \text{Field} \\
(constants) & c & ::= & \mathbb{Z} \mid \mathbf{SHF\_ORDER}(\mathbf{c},\mathbf{c},\mathbf{c},\mathbf{c}) \\
(expr) & e & ::= & c \mid x \mid a[i] \mid e.f \mid x \circ x, \text{ where } \circ \in \{+,=,\&,\ldots\} \mid \mathbf{allzeros}(\mathbf{v}) \\
(vector\ expr) & \mathbf{ve} & ::= & \mathbf{v} \mid \langle \mathbf{e},\mathbf{e},\mathbf{e},\mathbf{e} \rangle \mid \mathbf{load\_128}(\mathbf{a},\mathbf{i}) \mid \mathbf{shuffle\_i32}(\mathbf{v},\mathbf{v},\mathbf{c}) \\
 & & & \mid \mathbf{op}(\mathbf{v},\mathbf{v}), \text{ where } \mathbf{op} \in \{\mathbf{add\_i32}, \mathbf{cmpeq\_i32}, \mathbf{and\_i128}, \ldots\} \\
(stmts) & st & ::= & x := e \mid a[i] := x \mid e.f := x \mid \mathsf{skip} \mid \mathbf{v} := \mathbf{ve} \mid \mathbf{store\_128}(\mathbf{a},\mathbf{i},\mathbf{v}) \\
(block) & b & ::= & st \mid b; b \mid \mathsf{if}\ x\ \mathsf{then}\ b\ \mathsf{else}\ b \\
(flowblock) & f & ::= & b \mid \mathsf{return}\ x \mid \mathsf{break} \mid f; f \mid \mathsf{if}\ x\ \mathsf{then}\ f\ \mathsf{else}\ f \\
(loop) & \ell & ::= & \mathsf{for}\ i := e;\ i \bowtie e';\ i = i \pm c;\ \mathsf{do}\ f, \text{where } \bowtie \in \{=,\neq,<,>,\leq,\geq\} \wedge e' : \text{Int32} \wedge e' \text{ is invariant in } f \\
(fragment) & \delta & ::= & f; \ell; \boldsymbol{\ell*}; f
\end{array}
$$

Figure 3. Program Fragment Language and SIMD extensions (in bold)

This loop contains two major challenges from the viewpoint of automatic vectorization. First is that since the loop can exit on any iteration (*i.e.* `return 1`) the loop carries a control flow dependence on all previous iterations. Second is the fact that the data is poorly laid out for SIMD processing – it is in an *array of structs*. Thus, doing a block load from the array will get a mixture of the `tag` and `score` fields. Since these fields are processed differently in the loop body (`tag == tv` vs. `score > sv`) the mixture prevents the direct use of SIMD operations (which apply the same operation to each value). Thus, this loop body does not fit into a standard vectorization template form. Attempting to write a compiler that recognizes and transforms this loop appropriately based on a set of pattern matching rules is unattractive from both an implementation effort and complexity standpoint.

Despite these complications it is possible to construct an efficient SIMD implementation using the SSE instructions found in x86 processors (see Figure 2). The program first loads two data blocks of 128 bits each (two `widget` structs per load) from the array via the `load_128` operation. The SSE implementation handles the array-of-struct issue by *swizzling* [14] the four `tag` values into one SSE register (`tagvs`) and the four `score` fields into a second SSE register (`scorevs`). This is done by computing two swizzle masks, `tvswizzle` and `svswizzle`, and using them to control how the data that was loaded from the array is unpacked by the `shuffle_i32` operations. The `tvswizzle` mask indicates that the 0^{th} and 2^{nd} entries, which contain the `tag` fields, should be loaded from `blck1i` and `blck2i`, and these four values should be placed into `tagvs`. Similarly the 1^{st} and 3^{rd} values, which contain the `score` fields, should be placed into `scorevs`. Once these values are unpacked it is then simple to apply the appropriate SIMD equality (`cmpeq_i32`) and greater than (`cmpgt_i32`) operations to compare the four `tag` fields and the four `score` fields. These comparison operations produce bitmasks in the result vector, all 1's if the test result is true and all 0's if the result is false, for each 32 bit value. The results of these comparisons are then bitwise anded in one step via the `and_i128` operation. The final test (`!allzeros(cmpr)`) checks if any of the `widgets` processed satisfy the constraints and if one does then the `match` value will be 1. Since the original loop simply returns on finding a matching `widget` the SSE loop returns 1 if any of the four `widgets` being processed match (*i.e.* the `allzeros` value is 0). The resulting SSE implementation outperforms the simple loop by well over a factor of $2\times$ for large numbers of iterations and is 25% faster even on small iteration counts.

There are a number of challenges present when designing a system to automatically vectorize loops, such as the one in Figure 1. The first challenge is structuring the vectorization algorithm such that it is *applicable* to a wide range of loops and variations in how they are implemented [21]. This is critical to ensuring that the auto-vectorization is consistently able to find optimized implementations for loops in the input programs and thus improve performance in practice. The next challenge is that the process of vectorizing code often adds complexity and overhead. In order to avoid slowing down the program instead of speeding it up, it is useful to be able to predict if (and when) the SIMD implementation will *reliably improve* the performance of the program relative to the initial implementation. Finally, a fundamental issue in any compiler optimization is *correctness*. Since compiler bugs may introduce errors into every program that is compiled, it is critical to ensure that the resulting SIMD code is equivalent to the input program.

Contributions. To construct an auto-vectorization algorithm that achieves the desired *applicability*, *reliable improvement*, and *correctness* objectives, this paper makes the following contributions:

- A new methodology for program optimization (Section 4) based on a novel combination of: *deductive* rewriting of loop and control-flow structures, *inductive* synthesis of the desired code blocks, and a novel construction based on relational program verification to connect the deductive and inductive steps.
- The methodology is applied to the problem of auto-vectorization of irregular loops that have sub-optimal data layouts and complex data driven control flow. In particular we look at library code from the C++ STL or the C# Base Class Libraries.
- An efficient technique for inductive synthesis of loop-free code fragments, based on a novel combination of concrete program execution, bounded search techniques, and symbolic counter example generation methods (Section 5).
- An experimental evaluation of the auto-vectorizer on a set of challenge loops and real-world applications (Section 7). The results show that the technique performs well in practice: producing SIMD implementations which outperform the original implementations by $2.0\times$-$3.7\times$. We also apply the technique to vectorize loops in the SPEC `483.Xalan` benchmark to obtain a 5.5% reduction in runtime.

2. Relational Verification

We begin by reviewing *relational verification* [8, 42] and explain how this technique can be used to reason about the equivalence of two implementations of a loop. We will then introduce two novel forms of *equivalence relations* on program variables for showing the equivalence of a scalar and a vectorized loop.

2.1 Relational Verification Background

The key insight in relational verification is that given two similar programs one does not need to know the exact functionality of the two programs in order to show that they are equivalent. It is sufficient to show that, at the appropriate *synchronization points* during their execution, the states of the two programs are equivalent under some relation. Consider the loops:

```
int sum = 0;
for(int i = 0; i < n; i++)
  sum += i;
```

```
int j = -1;
int sum = 0;
for(int i = 0; i < n; i++) {
  j++;
  sum += j;
}
```

We begin by renaming any variables v which appear in both programs as $v_{\langle 1 \rangle}$ for the value of the variable in the first program on the left and $v_{\langle 2 \rangle}$ for the value of the variable in second program on the right. After this renaming then the equality relation for the states of the two programs is $i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge j = i_{\langle 1 \rangle} - 1 \wedge sum_{\langle 1 \rangle} = sum_{\langle 2 \rangle}$.

Using this relationship we can show these two loops compute the same value. We begin by checking that, when the loop iterations are run in lockstep, at every iteration the states of the programs are equivalent under the relation. Once we have shown that the equivalence relations hold at every loop iteration we can show that they hold after the exit of the loop as well. Thus, we can generate a proof that the two loops compute the same values for the final sums and are observationally equivalent, i.e. $sum_{\langle 1 \rangle} = sum_{\langle 2 \rangle}$.

A critical step in this process is obtaining suitable equality relations. Techniques for obtaining some of these relations, particularly relating to loop structure and conditional control flow, have been developed in previous work [4]. Loop splitting and unrolling are standard transformations which make latent data-parallelism in the loop body more easily exploitable. As SIMD operations operate on k values at a time we need to restructure the loop so that (1) the iteration count of the loop is a multiple of k and (2) that there are k exposed values to operate on. Similarly we can separate the expected hot path in the loop body from the branches that may lead to abnormal loop exits. This restructuring can be viewed as a variation on the *hot-trace* with a *guarded trace-exit* flow restructuring that is commonly done in *Tracing Just-In-Time Compilers* [1, 7].

2.2 Relational Verification of SIMD Loops

In this work we are primarily interested in showing the equivalence of a scalar loop and a loop using SIMD instructions. Thus, to leverage the relational verification machinery we need to identify a suitable set of equivalence relations that may hold between a scalar loop and the corresponding SIMD implementation. We have identified two commonly occurring forms for these equivalence relations, invariant and reduction expressions Section 4, which are sufficient to enable the scalar/SIMD loop equivalence verifications we are interested in. Consider the following loops which illustrate the needed equivalence relations:

```
int x = ...;
int hash, i = 0;


for(; i < n; i += 4) {
  hash ^= A[i] & x;
  hash ^= A[i+1] & x;
  hash ^= A[i+2] & x;
  hash ^= A[i+3] & x;
}
```

```
int x = ...;
int hash, i = 0;
m128i hv = [0, 0, 0, 0];
m128i xv = [x, x, x, x];
for(; i < n; i += 4) {
  m128i d = load_128(A + i);
  m128i t = and_i128(d, xv);
  hv = xor_i32(hv, t);
}
hash = (hv.r0 ^ hv.r1 ^
        hv.r2 ^ hv.r3);
```

The first loop on the left contains several variables with live ranges that span multiple iterations of the loop. The variable `x` is an invariant value in the loop on the left and a vectorized invariant version `xv` is used in the second loop on the right. The variable `hash` is a reduction variable in the first loop. The loop on the right represents these accumulated values in four `i32` values in the vector variable `hv` and adds a final reduction at the exit of the loop. Thus, the relations needed to show these loops are equivalent on each lockstep iteration are: $i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge xv = [x_{\langle 1 \rangle}, x_{\langle 1 \rangle}, x_{\langle 1 \rangle}, x_{\langle 1 \rangle}] \wedge hash_{\langle 1 \rangle} = (hv.r0 \hat{} hv.r1 \hat{} hv.r2 \hat{} hv.r3)$.

Given these relations it is straight forward to use a relational verification technique to show the program fragments are equivalent. In practice we use a *product program construction* [4] with off the shelf SMT solvers to solve the generated verification conditions. The equivalence relation is clearly satisfied on the first entry to the loop. Then inductively we can see that if the equivalence holds on iteration k then in iteration $k + 1$ it will again hold after executing both loop bodies. The final step is then simply to show that when the loop exits, and after executing the final reduction after the loop when the relational equivalence invariant holds, that $hash_{\langle 1 \rangle} = hash_{\langle 2 \rangle}$.

In practice these new relations, *invariant expression vectorization* and *reduction variable vectorization*, along with previously known relations (Section 2.1) for reasoning about loop control-flow restructuring are sufficient to verify the equivalence of the loops that are of interest in this work. In Section 4 we will formalize the definitions for the invariant and reduction vectorization equivalence relations. Additionally, we show how to leverage the relational verification methodlogy to construct the constraints needed to synthesize a vectorized body given a scalar implemention of a loop.

3. Problem Description & Algorithm Overview

This section presents a formalization of the program fragment language that we want to vectorize and the language with SIMD instructions that the auto-vectorization algorithm produces as output. We also provide an overview of the auto-vectorization algorithm.

3.1 Input and Output Loop Languages

Input Language. The work in this paper operates on a core imperative language shown in the non-bold portion of Figure 3. For simplicity, this language consists of variables and operations on three types: 32-bit integers, user defined structures (with named fields) and arrays of either structures or integers. This language extends naturally to include other integer sizes, floating point values, etc. The expressions e in the language cover the standard sets of arithmetic, bitwise, comparison, and access operations. The language admits the standard suite of assignments to locals, array locations, and fields in structs.

To focus on blocks of code that are suitable for vectorization, we distinguish between blocks consisting of simple assignments with conditional flow inside a single iteration b, and blocks of statements that may contain non-local control flow f. The grammar describes the structure of the loops that we are interested in vectorizing and which are likely to benefit most from the conversion to a SIMD implementation – innermost loops that are free of function calls. However, in practice the technique can be applied more aggressively by explicit inlining of function calls or by providing explicit pre/post semantics for an inner loop or method call.

To ensure that the loop is amenable to vectorization, we also impose some semantic restrictions: (1) the loop limit expression e' is invariant, (2) the iteration variable is only updated by linear operations, and (3) the updates are done uniformly on all paths of the loop. Finally, we define a program fragment δ as a single loop with possible loop initialization and clean-up code.

SIMD Output Language. The output of the auto-vectorization algorithm is a program in the SIMD extended language shown in Figure 3, including the terms in bold. The output language extends the input language with a set of SIMD instructions similar to what is present in the Intel SSE4 instruction set. For simplicity, we assume that all of the vectors ($v \in V$) are 128 bits which can contain 4 integers of 32 bits each. We extend the constant set with macros for shuffle constants and add the `allzeros` operation to the set of expressions that produce integer values (e). The SIMD expressions (**ve**) treat each 128 bit vector either as a single bit set of 128 bits for logical operations (*e.g.* `or_i128` or `and_i128`) and as four 32 bit integer values for arithmetic and comparison

operations (*e.g.* `add_i32` or `cmpgt_i32`). We add operations to load (`load_128`) and store (`store_128`) 128 bits at a time. Finally, we allow the fragment to contain a sequence of loops.

3.2 Algorithm Overview

The auto-vectorization algorithm is depicted in Figure 4. This flow diagram shows how we first apply deductive restructuring to the loop to expose data parallelism using deductive rewritings. From this restructured loop and the associated equivalence relations from Section 2 we extract a loop-free block of code from the loop body which will be replaced with a sequence of synthesized SIMD instructions. This synthesized code is then patched back into the loop. Finally we compute a cost scoring function and a proof of correctness for the final code fragment.

Restructuring and Pre/Post Generation. The loop is first restructured via standard loop splitting/unrolling and if-conversion. We also introduce vector variables (Section 4) which are used in the synthesis phase. The condition generator examines the restructured program and equivalence relations that are built up during the resturcturing to construct the needed synthesis pre/post conditions.

Inductive Concolic Synthesis. The synthesis phase (Section 5) takes the pre/post conditions produced by the previous step and produces a sequence of instructions that realize the specified behavior. The synthesizer uses a novel combination of concrete program execution and counter example generation, which we call *concolic synthesis*. This combined search approach quickly produces an efficient sequence of instructions that satisfies the pre/post conditions and this sequence of instructions is the output of this phase.

Merge and Cost Ranking Function. The final step in the algorithm is to patch in the synthesized code for the hole in the program and to clean up any dead or loop invariant code that may have been created in the vectorization step. The final program is passed to the cost ranking computation (Section 6), and a proof of correctness is computed for the program.

Output. The output of the algorithm is (1) the SIMD optimized program, (2) a proof of equivalence between the SIMD implementation and the original program, and (3) a cost ranking function.

This approach provides the ability to use deductive heuristic rules to quickly rewrite a loop to expose parallelism and enables the reduction of the synthesis problem to small blocks of code. The synthesis component provides a simple inductive method to construct efficient code blocks that is robust to a wide range of structures, and variations on these structures, that appear in the loop bodies. The relational verification methodology provides a connection between the inductive and deductive approach allowing them to co-operate to maximize the strengths of each approach. As an additional benefit the correctness certificate enables the pre-compilation of code in a managed language, such as C#, to assembly code which can be deployed and JITed without violating the safety guarantees of the language [26, 27].

4. From Relational Verification to Synthesis

We borrow the general concept of turning an appropriate verification methodology into a synthesis algorithm from [38] and extend the core idea to apply it to our problem domain. The approach in [38] requires a specification of the program to be synthesized as a pre/post condition pair (ϕ, ψ) and a template T (with only first-order holes) that form the full-correctness proof for the program. In our setting there are two natural possibilities for constructing the pre/post conditions: (1) the minimal loop invariant required for correctness and (2) the precondition and postcondition for the loop body. Unfortunately, both of these options are unsatisfactory. The computation of loop invariants (even with the limited language in Figure 3) is an undecidable problem. Conversely, pre/post conditions based on only the loop body can be computed efficiently. However, the resulting conditions are highly restrictive and cannot be used for loops that require certain vector registers to be live across loop iterations (such as reduction variables).

We use results from the area of *Relational Program Verification* [4, 8, 30] and the transformation/verification rules outlined in Section 2 to generate the synthesis pre/post conditions. To expose or create data parallelism which can be exploited in the SIMD synthesis step we utilize standard loop restructuring rules (splitting, unrolling, and if-conversion) and rules for introducing vectorized variables or constants. Each rule consists of (1) a loop rewriting template and (2) a template for the equivalence relation between the original loop and the rewritten version. The equivalence relations are used to generate the desired synthesis condition and an equivalence proof between the original loop and the vectorized version (or to reject the vectorized version if a proof cannot be generated).

4.1 Introduction of Vectorized Variables/Constants

Vectorization of Loop Invariant Expressions. We identify variables and expressions e that are invariant across loop iterations in the standard manner – either none of the values used in e are modified in the loop body or they are assigned the result of another loop invariant expression. For each invariant expression, e of type Int32, we introduce the corresponding *vectorized* version, $\texttt{vece} := \langle e, e, e, e\rangle$, where `vece` is a fresh variable name and the initialization is done before the loop. We accumulate the loop invariant expressions and the corresponding vector variables as tuples $(e, \texttt{vece})$ in the set V_e. The equality relationship that should exist between the scalar and vector forms is given by:

$$\mathsf{Inv}(V_e) = \bigwedge_{(e,v)\in V_e} v = [e_{\langle 1\rangle}, e_{\langle 1\rangle}, e_{\langle 1\rangle}, e_{\langle 1\rangle}]$$

Vectorization of Reduction Variables. We define a reduction variable x as a candidate for reduction variable vectorization when: x is not used as an array index and all paths through the loop contain an assignment of the form $x := x \bullet e$ where $\bullet$ is commutative. This definition heuristically identifies a reduction variable x, introduces a vector version `vecx`, and adds the appropriate initialization before the loop with reduction at loop exit. This definition is unsafe to use in general as we have ignored the effects of the rest of the loop body and their interaction with the reduction operator. However, in the case where the transformation is unsafe we will not be able to produce a proof of equivalence in the relational verification step and will reject the resulting program. The equality relationship that should exist between the scalar and vector forms is given by:

$$\mathsf{Reduce}(V_r) = \bigwedge_{(x,v,\bullet)\in V_r} \begin{pmatrix} v = [r_0, r_1, r_2, r_3] \wedge \\ x_{\langle 1\rangle} = x_{\langle 2\rangle} \bullet (r_0 \bullet r_1 \bullet r_2 \bullet r_3) \end{pmatrix}$$

4.2 Final Equivalence Relation

Once we have the identified the loop invariant expressions, the reduction variables, and the input and output variables (I and O) for the loop, the next step is to construct the final equivalence relations at the desired synchronization points.

In our setting the synchronization points correspond to the program points at the normal control-flow entries and exits of the loop bodies. By definition the variables in V_e or V_r are in scope at both the loop body entry and exit points so the special equality conditions for them are the same at both points. For the variables not in the V_e or V_r sets we check if they are in the input or output variable sets (I or O) and if so add an equality condition between the versions in the two programs. We say a variable x is *simple* at a

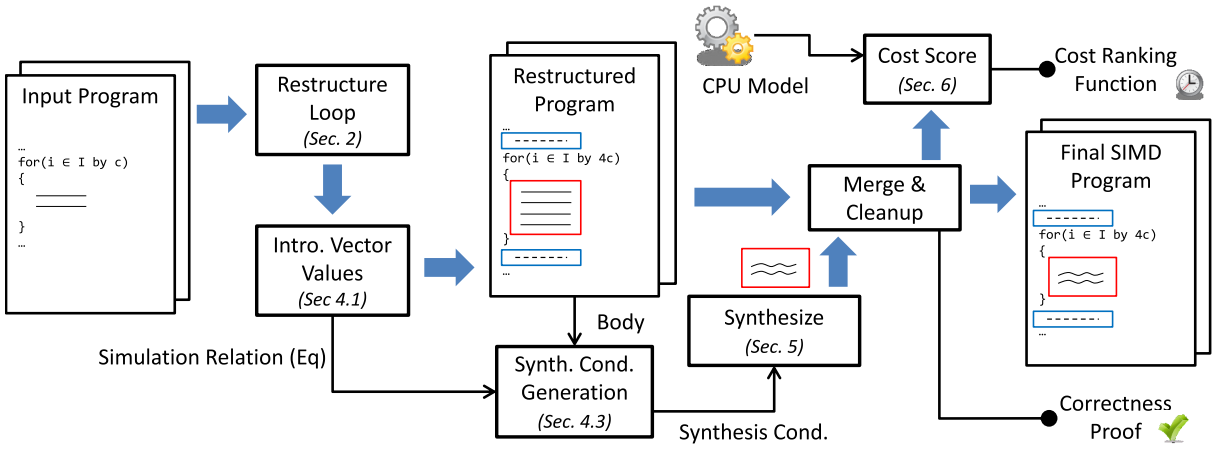

Figure 4. Overview of the auto-vectorization algorithm.

program point if it is defined at that point and it is not an invariant or reduction variable. We define the equivalence relation for the loop entries E_{pre} and exits E_{post} as:

$$E_{pre} = \mathsf{Inv}(V_e) \wedge \mathsf{Reduce}(V_r) \wedge \bigwedge_{x \in X} x_{\langle 1 \rangle} = x_{\langle 2 \rangle}$$
$$\text{where } X = \{x \in I | x \text{ simple at the loop entries}\}$$
$$E_{post} = \mathsf{Inv}(V_e) \wedge \mathsf{Reduce}(V_r) \wedge \bigwedge_{x \in X} x_{\langle 1 \rangle} = x_{\langle 2 \rangle}$$
$$\text{where } X = \{x \in I \cup O | x \text{ simple at the loop exits}\}$$

4.3 Partial Program and Condition Generation

As the natural candidate code for conversion to a SIMD implementation is the normal control-flow block of the loop body. We replace the normal control-flow block between the loop entry and exit with a *hole* [36]. Using the equality relations from E_{pre} and E_{post}, along with the *weakest preconditions* computed with them, we can construct pre/post conditions ϕ and ψ for the hole which are used to construct replacement code to fill the hole.

Given our choice synchronization points as the loop normal control-flow entries/exits, the required verification condition is of the form $E_{pre} \Rightarrow \mathsf{wp}(b_1, \mathsf{wp}(b_2, E_{post}))$ where b_1, b_2 are the loop bodies from the left and right programs respectively and wp computes weakest preconditions. If b_2 is a *hole* then this verification condition is a specification for the required code. We compute the synthesis pre/post conditions (ϕ, ψ) for our synthesis hole by taking the code for the block we want to replace and compute:

$$\phi = \exists I_{\langle 1 \rangle} E_{pre} \text{ where } I_{\langle 1 \rangle} = \{x_{\langle 1 \rangle} | x \in I\}$$
$$\psi = \exists I_{\langle 1 \rangle} (E_{pre} \wedge \mathsf{wp}(b_1, E_{post})) \text{ where } I_{\langle 1 \rangle} = \{x_{\langle 1 \rangle} | x \in I\}$$

This construction lifts the relational program verification methodology to a synthesis condition generation methodology. Further, it reduces the problem of synthesizing loopy programs to the problem of synthesizing straight line code. However, it does so in a way that preserves cross loop information as well as context from before/after the loop body. This context ensures that the generated conditions are as *relaxed* as possible, enabling the generation of optimized code in the synthesizer, while still ensuring the equivalence of the original and optimized programs.

4.4 Running Example

Figure 5 shows the result of applying these transformations to the input code from Figure 1. The resulting fragment has two loops, the first one has a loop guard that ensures the loop iteration count is a multiple of 4 while the second loop handles the remaining iterations. The first loop has been unrolled 4 times and the variables have been uniquely renamed to expose 4 independent sets of values for the vectorization. The if-conversion step has swept the conditional guards and abnormal loop exit to the single flow block at the end of the loop.

After the loop restructuring and introduction of vector variables, `vectv` and `vecsv` for `tv` and `sv` respectively, we have the following following equivalence post relation for the body:

$$\begin{aligned} E_{post} = \; & match_{\langle 1 \rangle} = match_{\langle 2 \rangle} \wedge vals_{\langle 1 \rangle} = vals_{\langle 2 \rangle} \\ & \wedge i_{\langle 1 \rangle} = i_{\langle 2 \rangle} \wedge tv_{\langle 1 \rangle} = tv_{\langle 2 \rangle} \wedge sv_{\langle 1 \rangle} = sv_{\langle 2 \rangle} \\ & \wedge vectv = [tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}, tv_{\langle 1 \rangle}] \\ & \wedge vecsv = [sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}, sv_{\langle 1 \rangle}] \end{aligned}$$

The code shown in Figure 6 has had the normal control flow code in the loop, from the first statement to the `if`, replaced with a `[HOLE]` as a place holder for the code we want to synthesize. The pre/post conditions we want to generate (ϕ and ψ) for use in the synthesis step are shown before/after the hole.

The only assignments to externally visible variables that can be made by the synthesized code are specified by the set O. Thus, we simplify the computed post condition, ψ, by assuming that all variables not in O have the same values before/after the synthesized code. As the only variable in O is `match`, the interesting parts of the generated synthesis pre/post conditions are:

$$\begin{aligned} \phi = \; & (vectv = \langle tv, tv, tv, tv \rangle \wedge vecsv = \langle sv, sv, sv, sv \rangle) \\ \psi = \; & (match = ((vals[i].tag = tv \wedge vals[i].score > sv) \\ & \vee (vals[i+1].tag = tv \wedge vals[i+1].score > sv) \\ & \vee (vals[i+2].tag = tv \wedge vals[i+2].score > sv) \\ & \vee (vals[i+3].tag = tv \wedge vals[i+3].score > sv))) \end{aligned}$$

After synthesizing the SIMD code for these conditions and substituting it in for the hole we get the final program shown in Figure 2. Using the equivalence relations E_{pre} and E_{post} we can compute and discharge a set of verification conditions for the original input loop and the final SIMD implementation which serve as a correctness proof for the transformation. Finally, using the construction in Section 6 we can produce a cost function for the relative performance of the input and SIMD loops.

```
int i;
for (i = 0; i < len-3; i+=4) {
  int tagok0 = vals[i].tag == tv;
  int scoreok0 = vals[i].score > sv;
  int andok0 = tagok0 & scoreok0;

  ...

  int tagok3 = vals[i+3].tag == tv;
  int scoreok3 = vals[i+3].score > sv;
  int andok3 = tagok3 & scoreok3;

  match = andok0 | andok1 | andok2 | andok3;
  if (match) return 1;
}

for (; i < len; ++i) {
  int tagok = vals[i].tag == tv;
  int scoreok = vals[i].score > sv;
  int andok = tagok & scoreok;
  if (andok) return 1;
}
```

Figure 5. Running example after structural transformation.

```
int i;
for (i = 0; i < len-3; i+=4) {
  φ
  [HOLE]
  ψ
  if (match) return 1;
}

for (; i < len; ++i) {
  int tagok = vals[i].tag == tv;
  int scoreok = vals[i].score > sv;
  int andok = tagok & scoreok;
  if (andok) return 1;
}
```

Figure 6. Running example after hole insertion and pre/post condition locations shown.

5. Inductive SIMD Synthesis

The synthesis algorithm takes a pre/post condition pair (ϕ, ψ), a set of instructions to select from *Stmts*, the set of input variables I and outputs O, and a maximum cost for the program to be synthesized ($cost_m$). The output is a program p which is a sequence of statements such that for any state valuation s that satisfies the pre-condition ϕ, the execution of p starting in s yields a state valuation s' that satisfies the postcondition ψ. Inspired by work on *concolic testing* [9, 32] our concolic synthesis algorithm uses a combination of a top-level counter-example driven loop (based on symbolic methods) to find interesting values for the inputs I and an efficient search for candidate programs p (based on concrete execution over these input values). The symbolic reasoning in the top-level loop (Algorithm 1) ensures that each new input provides useful information, which forces behavioral differences, while the use of concrete values in the program search subroutine (Algorithm 2) provides an efficient method for generating candidate programs.

5.1 Counter-Example Generation Loop

The top-level *CEGIS* (Counter-Example Guided Inductive Synthesis [35]) loop in Algorithm 1 iteratively constructs a set of *concrete state valuations* (a mapping of values to variables) and searches for a candidate program p that satisfies the postcondition ψ when run on these state valuations, line 6. On line 4 the algorithm attempts to symbolically construct a new input state valuation s that is a counter-example for the correctness of the program p – *i.e.* ψ does not hold on the result of running p on s. If such an example can be found it is added to the set on line 5 and the loop is repeated, if we can prove that no such example exists then p is the desired program and we return on line 8, and if we cannot decide if such an example exists then the synthesis fails. The initialization of the concrete state valuations set, the underlined call to *GenInitialStates* on line 2 is an optimization, described in Section 5.3, to minimize the number of iterations of the CEGIS loop.

Algorithm 1: Top-Level CEGIS Loop

input : pre ϕ, post ψ, statements *Stmts*, inputs I, outputs O, max. cost $cost_m$, disjunctive precondition χ
output: program p

1 $p \leftarrow$ skip;
2 $S \leftarrow \underline{\mathit{GenInitialStates}(\chi)}$;
3 **while** GenModel$(\exists \vec{V}, \phi \wedge \neg \text{wp}(p, \psi)) \notin \{\text{unsat}, \text{fail}\}$ **do**
4 $\quad s \leftarrow \mathit{GenModel}(\exists \vec{V}, \phi \wedge \neg \mathit{wp}(p, \psi))$;
5 $\quad S \leftarrow S + s$;
6 $\quad p \leftarrow \mathit{Search}(\langle\rangle, S, \psi, \emptyset, \emptyset, \mathit{Stmts}, I, O, \mathit{cost}_m)$;
7 $\quad$ **if** $p = \perp$ **then return** fail;
8 **return** (GenModel$(\exists \vec{V}, \phi \wedge \neg \text{wp}(p, \psi)) = \text{unsat})$ *?* p *: fail*;

5.2 Candidate Program Search

The *Search* method, Algorithm 2, performs the search for a program p_{res} that when run on the input list of state valuations S produces a list of state valuations that satisfy the post condition ψ. The *naive search*, i.e., the algorithm excluding the underlined code, is a depth first enumeration of possible sequences of instructions from the set *Stmts*. If we reach a point where every state valuation in S satisfies ψ then we have a candidate program and can return it, line 9. Otherwise the current program is extended with another instruction from *Stmts*, yielding p_i, and this statement is applied to each of the state valuations in S, yielding S_i. The new values, p_i and S_i, are then used in the recursive search call on line 14. As this naive search approach is computationally intractable for instruction sequences of length greater than four [12, 16, 23] we introduce several optimizations below.

5.3 Synthesis Optimizations

Initial State Valuations. The set of input state valuations S in the top-level CEGIS loop (Algorithm 1) plays a critical role in the number of iterations required for the loop to terminate. Every new state valuation that is added to S is, by construction, a counter-example and when no further counter-examples can be generated the loop terminates. Thus, we initialize S with a number of input valuations that are likely to provide good initial constraints and as a result we will need to generate very few additional counter-examples. As in concolic testing we note that different paths through the program are likely to exercise different behaviors. Thus, we alter the synthesis algorithm to take a *disjunctive pre-condition* χ, which is a disjunction of *per path* weakest preconditions from the input program. The *GenInitialStates* method produces a state valuation for each clause in the disjunctive pre-condition and we use these to initialize S on line 2.

Search Merging. The naive search builds redundant instruction sequences that repeatedly generate the same program state valuations, *e.g.* repeatedly add and then subtract a constant. We also observe that the search re-explores equivalent state valuations that are reachable on different instruction paths, *e.g.* $(a + (b + c)) - d$ and $(a + b) + (c - d)$. We can eliminate this redundant exploration by merging branches in the instruction sequence search tree that are actually exploring the same set of state valuations. This is

done by adding a set, *Seen*, of state valuations that have been seen during previous search steps (checked on line 2). If we encounter a state valuation that has been previously seen it means that either (1) the current instruction sequence has redundant instructions, in which case it is suboptimal, or (2) we have already explored the state valuations reachable from the current valuation, and so continuing exploration on this sequence of instructions merely re-visits previously seen state valuations. In these cases, pending a check on cost information described below, we simply abort the current branch of the search on line 3.

Cost Bounds. In our application we are only interested in minimal cost code sequences. Using the cost model from Section 6 we score the initial program fragment that is being replaced and compute cost scores for each program generated during the search. With this information we can immediately stop searching, line 1, if the current program has a larger score, $cost_m$, than the input program or current best solution. This bound is updated as needed to be the best found so far in a standard branch-and-bound manner, line 17.

We further refine how the search handles state valuations that have been seen previously by noting that computational cost is monotone. Thus, repeating the exploration of a previously visited state with a higher cost program will not discover a faster program. However, if the current instruction sequence was able to produce the current state valuation more efficiently than previous instruction sequences then it may be possible to reach target state valuations satisfying ψ without exceeding the cost bound $cost_m$. Thus, on line 3 we check if we have found a less costly instruction sequence, if not we return immediately but if the new instruction sequence is less costly we update the min cost for this state valuation on line 4 and continue the search (re-exploring as needed).

Stack Machine. In order to limit the introduction and lifetime of intermediate values, as well as to reduce the combinatorial problems of selecting which variables to use/modify in each instruction, we extend the concrete execution state valuation with an evaluation stack. The use of an evaluation stack is a common way to simplify the operation of an abstract machine (*e.g.* the .Net and Java virtual machines) by removing the need to explicitly refer registers or to introduce explicit temporary variables. In the instruction selection step, line 11, we assume instructions take their arguments from the evaluation stack and place the result on the stack. We also extend the instruction set with operations to load input variable values on the stack and to pop values off of the stack into output variables. The introduction of an explicit evaluation stack allows us to place a bound on stack depth, line 8. This biases the search to avoid instruction sequences that produce large numbers of intermediate values which would produce code with high register pressure.

Incremental Search Expansion. We can obtain additional performance by using incremental expansion of the search parameters. In general the operations used by the original program (`==`, `&`, ...) are the same type of operations that will be needed in the SIMD version. Thus, we start with only the corresponding vector operations and basic load/store operations in the set of instructions (*Stmts*). If we fail to find a suitable program we extend this set with additional operations such as the `shuffle` and other bitmasking operations. Finally, if this larger set fails we let *Stmts* be the set of all instructions. Similarly, we start with a small eval stack, in our case depth 4 increasing to 6 if the first search fails. This allows us to improve the performance of the synthesizer in many cases but still allows the incremental exploration of the full program space as desired.

Algorithm 2: Concrete Program State Search

input : program p, state valuations S, post ψ,
seen set *Seen*, seen cost *Cost*, instructions *Stmts*,
inputs I, outputs O, max. cost $cost_m$
output: program p_{cand}

1 **if** $\text{cost}(p) \geq cost_m$ **then return** $\perp$;
2 **if** $S \in Seen$ **then**
3 **if** $cost(p) \geq Cost(S)$ **then return** $\perp$;
4 $Cost \leftarrow Cost + [S \rightarrow cost(p)]$;
5 **else**
6 $Seen \leftarrow Seen \cup \{S\}$;
7 $Cost \leftarrow Cost + [S \rightarrow cost(p)]$;
8 **if** $\text{Stack}_{\text{depth}}(S) > \text{Max}_{\text{stack}}$ **then return** $\perp$;
9 **if** $\forall s \in S \,.\, \psi$ *holds for* s **then return** p;
10 $p_{res} \leftarrow \perp$;
11 **foreach** stmt $\in$ Stmts$\cup\{\text{ldv}(v)|v \in I\} \cup \{\text{stv}(v)|v \in O\}$ **do**
12 $p_i \leftarrow p + inst$;
13 $S_i \leftarrow ApplyInstToAll(inst, S)$;
14 $p_o \leftarrow Search(p_i, S_i, \psi, Seen, Cost, Stmts, I, O, cost_m)$;
15 **if** $p_o \neq \perp \wedge cost(p_o) < cost_m$ **then**
16 $p_{res} \leftarrow p_o$;
17 $cost_m \leftarrow cost(p_o)$;
18 **return** p_{res};

6. Cost Ranking Function

The computation of absolute costs for arbitrary blocks of code is a challenging problem [39]. However, we do not need to compute the absolute costs of the programs. As we are only interested in identifying the best performing program from a set of candidates we only need to model cost in a way that allows relative comparison of two programs. Further, our more restricted program fragment language and vectorization application possess a number of simplifying features. The impacts of branch mis-prediction are parameterized as described below while the the uniform array accesses required for vectorization imply that the caching/prefetching in the processor will behave in a consistent and uniform manner.

We assume that we are given a model of the processor architecture, M, which contains the standard information on execution unit resources and latencies as well as branch mis-predict costs M_{miss}. We parametrize the remaining program fragment behaviors based on the conditionals C (*i.e.* `if` statements) and the loops L that appear in the program fragment:

$B_p : C \mapsto [0, 1)$	The mis-predict probability of each branch.
$B_t : C \mapsto [0, 1)$	The probability that the true path is taken.
$L_c : L \mapsto \mathbb{N}$	The number of times a loop is executed.

From these parameters we construct a cost ranking function $Perf^M : (\delta, B_p, B_t, L_c) \mapsto \mathbb{R}$. The cost of a straight line block of code is simply the sum of each statement as reported by the underlying processor model M. The cost of a branch statement, β with true branch β_t and false branch β_f is:

$$\begin{aligned} &Perf^M(\beta, B_p, B_t, L_c) = \\ &\quad B_p(\beta) * M_{\text{miss}} + B_t(\beta) * Perf^M(\beta_t, B_p, B_t, L_c) \\ &\quad + (1 - B_t(\beta)) * Perf^M(\beta_f, B_p, B_t, L_c) \end{aligned}$$

The cost of a loop statement, ℓ with the body ℓ_{body} is simply $M_{\text{miss}} + L_c(\ell) * Perf^M(\ell_{body}, B_p, B_t, L_c)$. We can compute the cost ranking function for a fragment where $\delta = f_{init}; \ell_1 \ldots \ell_k; f_{exit}$ in the natural way as the sum of all the costs:

$$\begin{aligned} &Perf^M(\delta, B_p, B_t, L_c) = Perf^M(f_{init}, B_p, B_t, L_c) \\ &\quad + (\Sigma_{\ell_1 \ldots \ell_k} Perf^M(\ell_i, B_p, B_t, L_c)) + Perf^M(f_{\text{exit}}, B_p, B_t, L_c) \end{aligned}$$

In this paper we report results when running the code on an Intel i7 processor. We can construct a (very) simple model M for this processor with: a normalized latency of 1 per operation, a 3 wide execution unit, a mis-predict cost $M_{\text{mis}} = 12$, a uniform mis-predict probability of 5% for forward conditional branches, and a 1% mis-predict rate for loop back and exit branches. Using this model the cost ranking function for the SIMD loop in Figure 2 is $\mathit{Perf}_{\delta}^{M}(\delta_{opt}, n_1, n_2) = 24 + n_1 * 5.16 + n_2 * 3.16$ and for the original loop in Figure 1 the function is $\mathit{Perf}_{\delta}^{M}(\delta_{orig}, n_1') = 12 + n_1' * 3.16$.

The estimated asymptotic speedup can be computed by observing that as n_1' becomes large the costs of the loops are proportional to $n_1' * 3.16$ for the original loop and $n_1' * 1.29$ for the SIMD loop (since the SIMD loop processes 4 elements per iteration). Thus, the cost ranking functions predict a speedup of $2.44\times$, closely matching the empirically observed speedup of $2.5\times$. To find the predicted break even point we solve for the values where the cost ranking functions for the original loop and SIMD loop are equal, $n_1' = 8$ for our functions. As we see in Section 7 this matches well with the experimentally seen break-even of between 4 and 8.

Even with our simple processor model, which can be built using readily available information, the resulting static cost predictions are both precise and, as we would like in a static compiler, conservative. This simple model can be further improved via either more detailed architecture descriptions [39] or autotuning [41] to identify key performance parameters in the processor models. As the cost estimation functions are parametrized on branch mispredict, branch taken, and loop count information it is also possible to evaluate them and select the best implementation based on runtime data, as in a Tracing JIT [1, 7].

7. Experimental Evaluation

To evaluate the approach presented in this paper we selected 18 loops which represent fundamental classes of algorithms (find, exists, accumulate, map, etc.) that are found in standard libraries such as the STL for C++ or the base class libraries for C# (or Java). These algorithms cover many common loop idioms that appear in real world code. In this section we examine 6 benchmarks in detail. Four benchmarks – CountIf, Find, Lexo, Equals – come from the C++ STL (specialized for random access iterators). Two benchmarks – FindIf and the running example Exists – are from the .Net base class libraries (BCL) and are implementations of methods in the `List<T>` class. Finally, the CyclicHash comes from production C++ code and implements a hash code function for data blocks. For the methods that take user defined lambda expressions – CountIf, FindIf, and Exists – we used non-trivial instantiations for the lambda code, *e.g.* the running example in Figure 1.

7.1 Transformation and Synthesis Performance

Table 1 shows the time required to vectorize each program fragment (memory is always less than 100MB). In practice the synthesis step accounts for 90% or more of this time. Thus, the table shows the number of input states the synthesis started with, the number of additional iterations the CEGIS loop needed, and the number of instructions in the final synthesized block. As we can see in this table the resource requirements vary greatly even for similarly sized code blocks. This variability is not surprising as the synthesis is fundamentally a search in a very large state space. However, in all of the cases the synthesizer was able to produce an optimized SIMD program. These programs consisted of up to 9 instructions and covered a diverse set of comparison, bitwise, and swizzling operations. The results also show the impact the *disjunctive precondition* generation heuristic has on the total number of iterations (taking only 1 iteration for all but one case).

Benchmark	Time(s)	Init./Iters.	Insts.
CountIf	0.136s	16/1	8
Find	0.053s	6/1	4
Lexo	0.056s	6/1	5
Equals	0.667s	10/1	5
Exists	0.120s	6/2	9
CyclicHash	0.998s	16/1	5

Table 1. *Time* required by the synthesizer. *Init* number of examples in S and *Iters* of the CEGIS loop. *Insts* in the final SIMD code.

7.2 Performance of SIMD Loops

To compare the performance of the synthesized SIMD loops and the original scalar implementations we implemented a driver loop which executes each loop, on inputs of various sizes, 5 million times in a simple timing loop. The evaluation was done on an Intel i7 running Windows 7 (32 bit) and Visual Studio C++ compiler (Version 16 for x86) with the default optimization settings.

Figure 7 contains a chart for each of the benchmark loops. This chart shows the experimentally measured performance improvement seen with the synthesized SIMD implementation and the performance improvements predicted by the analytical cost functions in Section 6. The logarithmic x-axis is the number of iterations that the original loop expected to execute. For fixed count loops like CountIf and CyclicHash this is the size of the input array. For loops with abnormal returns (the remaining four loops) this is the expected number of iterations before the loop exits. As we use a uniform distribution for where the element of interest is in the input the expected number of iterations is half the length of the input array. The y-axis is the speedup of the SSE implementation relative to the original scalar implementation. Finally we mark the break-even line where the performance of the SSE implementation and original implementation are equal.

The results in Figure 7 show that in general the SIMD implementations start to outperform the baseline implementations almost immediately (the *Actual* plot). For an iteration count of 8 only the Lexo loop is slower than the baseline implementation while CyclicHash is slightly better than break-even and the remaining loops show a 10% to 40% reduction in runtime. As the input size gets larger the performance differences get larger in favor of the SIMD loops. Once the iteration counts approach 32 the Lexo has passed the predicted break even point and is now faster than the baseline implementation. At iteration counts of 512 all the loops outperform the baseline by a factor of $2\times$ or more. Finally by iteration counts of 2048 the loops performance ratios are near their asymptotic speedup and now outperform the baseline implementations by between $2.0\times$ and $3.7\times$, which is near the $4\times$ maximum speedup we would expect from using 4 wide SSE instructions. These results demonstrate that the approach to vectorization described in this paper is *applicable* to a wide range of loops and produces SIMD implementations that consistently provide large performance increases (even on relatively small inputs).

The *Predicted* plots in Figure 7 show that in general the speedups predicted by the analytic cost model from Section 6 correlate well with the observed speedups – despite the relatively crude model used for the processor. The major exception to this trend is the Equals program where the predicted and actual performance diverge significantly for large iteration counts. Further investigation indicates that in this case the processor is able to optimize the loop execution in ways that are not captured by the simple processor model, M, used when constructing the cost functions. Thus there is room for improvement via either more detailed architecture descriptions [39] or autotuning [41] to identify key performance parameters in the processor models.

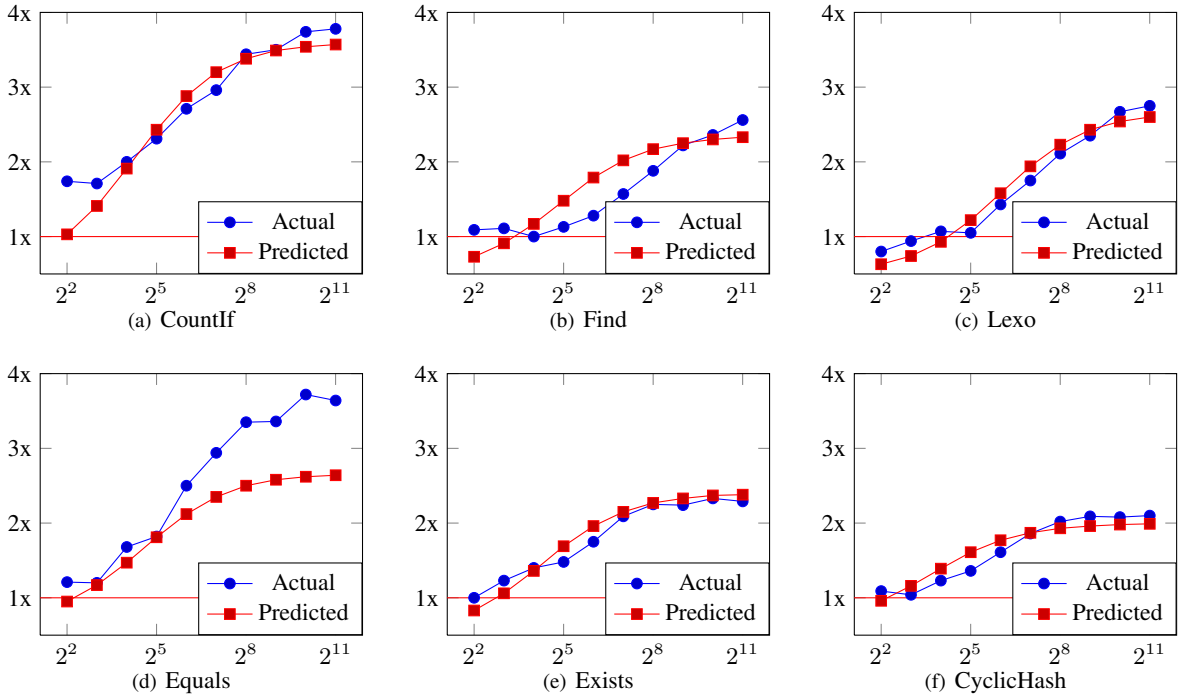

Figure 7. Speedup (Original Time / SSE Time) on Y axis ranging over the expected number of iterations in original loop on X axis.

To avoid performance degradations it is critical that the cost model is able to predict the break-even number of iterations (where the SIMD loop begins to outperform the original loop). In all our benchmarks we see that this number is well predicted by the cost model and in all cases is a conservative estimate (*i.e.* overestimating the number of iterations needed to break-even). Thus, these results demonstrate that the cost model defined in this work is an effective predictor for the relative performance of the loops and provides an effective means to check that a SIMD implementation will *reliably improve* the performance of the program in practice.

To validate that these results were not an artifact of the evaluation environment [25] we ran the evaluation on a second platform. This environment consisted of an Intel Core2 running Mac OS X (Tiger) and GNU C++ compiler (Version 4.2.1 x86). The results were, with one notable exception, consistent with the performance improvements seen on the Intel i7 platform. The outlier benchmark, *Exists*, had a break-even cost of 16 on the Core2 compared to a break-even of 8 on the i7. This increase is mainly a result of the `shuffle` operation being more expensive on the Core2 and the increased break-even is correctly predicted by our cost function.

7.3 Synthesis with Specialized Operations

In order to evaluate how the synthesis technique handles SIMD operations with unusual semantics we synthesized three common string operations from the C# `System.String` class: StringEquals, IndexOf and IndexOfAny. These can be implemented using the specialized *Packed Compare Strings* (PCMPESTRI) operation from SSE 4.2. The synthesis algorithm produces SIMD implementations for these loops using the specialized packed compare strings operation in less than 1 second for each benchmark. The speedups obtained ranged from 3.4× for StringEquals to 9.5× for IndexOfAny. These results demonstrate how the synthesis approach can be easily extended to make use of new, or unusual, instructions to produce optimized loop implementations.

7.4 Impact on 483.Xalan

The results in Section 7.1 show that the synthesized loop implementations consistently improve performance across a range of loops and input data sizes. To validate that the performance gains seen on the micro-benchmarks translate into similar performance gains in practice, we selected the 483.Xalan benchmark from SPEC CPU2006 [37] as a case study. This program makes heavy use of `std::vector<string*>` as a cache for commonly used strings and it uses the STL `find` algorithm (our Find benchmark) to find string pointers in the cache.

The cache behavior is very sensitive to the data that is being processed as shown in recent work on automatic data structure selection [17]. Replacing the `std::vector` with a `std::set` (or a `hashset`) resulted in performance improvements of up to 20% on the SPEC provided `train` input but when run with the SPEC provided `test` input the alternative data structure representation actually *degrades* performance by up to 20%. This swing from performance improvement to performance degradation is driven by the sensitivity of the cache to particular features of the input data set. Thus, this program tests both the performance impact of the SIMD code our synthesis produces and the robustness of the performance improvements on the various benchmark inputs.

Performance profiling of the 483.Xalan program shows that approximately 14% of the total runtime is spent executing the `find` algorithm on the cache. As our loop micro-benchmarks indicate that the SIMD `find` code is between 1.08× and 2.4× faster than the baseline implementation we would expect to see between a 1% (worst case) and 8% (best case) reduction in total runtime.

Table 2 shows the performance results obtained by replacing (by hand) the calls to the `find` algorithm with calls to our synthesized SIMD code. We show for each input provided in the SPEC test suite the size of the input and the percentage reduction in the total program execution time. The speedup indicates that the calls to the synthesized SIMD code are, depending on the inputs, 1.15

Input Data	Input Size	Improve(%)
`test`	28KB	5.5%
`train`	39MB	2%
`ref`	56MB	5%

Table 2. Runtime improvement(%) for `483.Xalan`.

to 1.5 times faster than the standard implementations (matching our expectations from the micro-benchmark results). In contrast to the widely variable speedup (and slowdown) seen by changing the underlying data structure, the use of the vectorized find loop showed consistent improvements of 2%-5% across the inputs.

8. Related Work

Vectorization. Automatic program vectorization is a challenging problem which requires the application of a wide range of techniques for effective vectorization including: loop transformation [18, 29], control flow dependency elimination [18], alignment optimizations [28, 40], and finding sets of operations that can be executed in parallel [19, 33]. However, previous work on compiler auto-vectorization has focused on what are traditionally considered *regular* applications (e.g., scientific codes, multimedia applications, encode/decode algorithms) and on special purpose libraries (codecs, encryption, etc.), where loops have well behaved termination conditions, data sets are of a fairly regular/large size, and data layouts are suited to SIMD computation.

In contrast, the work in this paper seeks to apply SIMD instructions to *irregular* loops from standard library implementations which often have poor data layouts, small iteration count loops, and extensive data dependent control flow. These types of programs present different and in many ways more difficult problems to the automatic construction of vectorized code. These difficulties are highlighted in a recent study by Maleki et. al. [21] which examines a number of state of the art vectorizing compilers and their ability to vectorize a range of loops. They conclude that modern compilers fail to vectorize many loop patterns due to a lack of development resources needed to build a compiler that can identify and treat all the needed loop and computation patterns.

The work in this paper focuses on the issues of sub-optimal data layouts and complex data driven control flow but does not examine issues involving indirect memory accesses via pointers. Recent work has begun to explore how to reorganize and traverse pointer based structures into flat structures which are amenable to SIMD computation [31]. In particular work on unique pointer referencing [3, 20] and object lifetime, either as a global invariant or in a localized section of code, based on static [20, 22] is a critical first step dealing with the challenges posed by pointers.

Verification. Translation validation is a general method for checking a posteriori that compiler runs are correct, i.e output target programs that are semantically equivalent to input programs [30, 43]. Product programs reduce relational verification to functional verification of a single program: instances include self-composition [6], cross-products [42], and their combination [4, 5]. These methods are able to validate a wide range of loop optimizations, including those needed by our method. In this work, we use product programs to generate synthesis conditions for loop bodies.

Relational Hoare Logic is a generalization of Hoare logic in which judgments involve two programs, and pre- and post- conditions are denoting relations on states [8]. Relational Hoare Logic is effective for proving the correctness of structure-preserving optimizations, and simple optimizations that alter the control flow of programs. However, the core logic of [8] does not support the kind of loop optimizations required for our examples.

Synthesis The area of program synthesis is gaining renewed interest [10, 11, 35]. Srivastava et.al. introduced the notion of *proof-theoretic synthesis* where the problem of synthesizing a loopy program, given a pre/post condition, is reduced to the problem of simultaneously synthesizing loop-free fragments and loop invariants [38]. This approach is limited to synthesis of simple programs whose total correctness proofs or loop invariants can be expressed as simple templates. In contrast, we reduce the problem of vectorizing a given loopy program, to the problem of synthesizing only a loop-free fragment (without the need to synthesize any sophisticated loop invariants). This reduction is enabled by our use of the powerful relational verification methodology, which allows us to separate the process of verification and synthesis by generating an over approximation of the equalities required for equivalence proof.

The problem of synthesizing loop-free programs has been addressed in a variety of domains including bit-vector algorithms [12, 15, 36], ruler/compass based geometry constructions [13], text transformations [24], and algebraic proof problems [34]. One class of technique is based on *constraint solving*, which involves reducing the synthesis problem to that of solving a SAT/SMT formula (inside a CEGIS loop) and let an off-the-shelf SAT/SMT solver efficiently explore the search space. The applicability of this technique has been limited to semi-automatic settings, where the user provides templates [36] or reasonable over-approximation of the number of times each base component is used in the desired program [12]. Another class of technique is based on *brute-force search*, which involves systematically exploring the entire state space of artifacts and checking the correctness of each candidate. This approach often requires use of non-trivial optimizations and performs best when the specification consists of examples as opposed to a formal relational specification. Past work has included optimizations such as goal-directed search [13], clues based on textual features of examples [24], and common subexpression evaluation [34]. In this work, we combine the CEGIS loop from constraint-solving approaches with brute force search approach and novel optimizations.

Superoptimization is the task of finding an optimal code sequence for a straight-line target sequence of instructions, and it is used in optimizing performance-critical inner loops. One approach to superoptimization has been to constrain the search space to a set of equality-preserving transformations [2, 16], and then select the one with the lowest cost. This approach is limited by the kind of transformations that it can generate. Another approach to superoptimization has been to use brute-force search and enumerate sequences of increasing length or cost, testing each for equality with the target specification [23]. We also use brute-force search, but combined with a CEGIS loop and non-trivial optimizations.

9. Conclusion

This work presents a new approach to addressing the challenges that are present when attempting to harness the performance and power advantages available from data-parallel SIMD operations. In particular we looked at the problem of auto-vectorizing loops that have sub-optimal data layouts and complex data driven control flow, as is frequently the case in general purpose library code from the C++ STL or the C# Base Class Libraries. Our approach is driven by three core objectives: to produce an auto-vectorizer that is *applicable* to a wide range of *irregular* loops, that produces code which *reliably improves* the performance of the loop, and that guarantees the *correctness* of the resulting SIMD code.

These objectives led us to a novel auto-vectorization approach based on *deductive* loop rewriting and *inductive* synthesis of loop-free code. The use of inductive synthesis for constructing the loop body makes it particularly robust when dealing with the multitude of variations on the basic loop forms (find, map, reduce, etc.) that

appear in practice. In addition this approach allows us to produce correctness proofs for the resulting code. We believe that this underlying approach of combining deductive code restructuring with inductive code generation represents a general and promising way forward in research on program compilation. Thus, this work is an important step in both expanding the set of programs that can be automatically SIMDized and in the larger problem of effective compilation for specialized hardware.

Acknowledgments

We would like to thank the PPoPP reviewers and Rastislav Bodik for their constructive comments and thoughts on this work. This work was supported in part by: European Projects FP7-318337 ENTRA, FP7-231620 HATS and FP7-256980 NESSoS, Spanish project TIN2009-14599 DESAFIOS 10, Madrid Regional project S2009TIC-1465 PROMETIDOS. César Kunz is funded by Spanish Juan de la Cierva programme (JCI-2010-08550). Juan Manuel Crespo is funded by FPI Spanish programme (BES-2010-031271).

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, 2000.

[2] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[3] E. Barr, C. Bird, and M. Marron. Collecting a Heap of Shapes. Technical Report MSR-TR-2011-135, Microsoft Research, Dec. 2011.

[4] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.

[5] G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: Asymmetric product programs for relational program verification. In *LFCS*, 2013.

[6] G. Barthe, P. R. DArgenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.

[7] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: A trace-based JIT compiler for CIL. In *OOPSLA*, 2010.

[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[10] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010. Invited talk paper.

[11] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *SYNASC*, 2012. Invited talk paper.

[12] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[13] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[14] Intel Optimization Manual (June 2011) – Section 6.5.1. http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[15] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[16] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[17] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. In *PLDI*, 2011.

[18] K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[19] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[20] K.-K. Ma and J. Foster. Inferring aliasing and encapsulation properties for java. In *OOPSLA*, 2007.

[21] S. Maleki, Y. Gao, M. Garzarán, T. Wong, and D. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011.

[22] M. Marron. Structural analysis: Shape information via points-to computation. Technical Report 1201.1277, arXiv, Jan. 2012.

[23] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, 1987.

[24] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML*, 2013.

[25] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, 2009.

[26] G. Necula. Proof-carrying code. In *POPL*, 1997.

[27] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI*, 1996.

[28] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.

[29] D. Nuzman and A. Zaks. Outer-loop vectorization: Revisited for short SIMD architectures. In *PACT*, 2008.

[30] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. In *TACAS*, 1998.

[31] B. Ren, G. Agrawal, J. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *CGO*, 2013.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13*, 2005.

[33] J. Shin, M. Hall, and J. Cha. Superword-level parallelism in the presence of control flow. In *CGO*, 2005.

[34] R. Singh, S. Gulwani, and S. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[35] A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[36] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[37] SPEC. Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/cpu2006/.

[38] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, 2010.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem: Overview of methods and survey of tools. *ACM TECS*, 7(3), 2008.

[40] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *CGO*, 2005.

[41] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005.

[42] A. Zaks and A. Pnueli. Covac: Compiler validation by program analysis of the cross-product. 2008.

[43] L. D. Zuck, A. Pnueli, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *J. UCS*, 9(3), 2003.