# Efficient Selection of Vector Instructions using Dynamic Programming

Rajkishore Barik*
Intel Corporation, Santa Clara, CA
rajkishore.barik@intel.com

Jisheng Zhao and Vivek Sarkar
Rice University, Houston, TX
{jisheng.zhao,vsarkar}@rice.edu

***Abstract*—Accelerating program performance via SIMD vector units is very common in modern processors, as evidenced by the use of SSE, MMX, VSE, and VSX SIMD instructions in multimedia, scientific, and embedded applications. To take full advantage of the vector capabilities, a compiler needs to generate efficient vector code automatically. However, most commercial and open-source compilers fall short of using the full potential of vector units, and only generate vector code for simple innermost loops.**

**In this paper, we present the design and implementation of an auto-vectorization framework in the back-end of a dynamic compiler that not only generates optimized vector code but is also well integrated with the instruction scheduler and register allocator. The framework includes a novel *compile-time efficient dynamic programming-based* vector instruction selection algorithm for straight-line code that expands opportunities for vectorization in the following ways: (1) *scalar packing* explores opportunities of packing multiple scalar variables into short vectors; (2) judicious use of *shuffle* and *horizontal* vector operations, when possible; and (3) *algebraic reassociation* expands opportunities for vectorization by algebraic simplification.**

**We report performance results on the impact of auto-vectorization on a set of standard numerical benchmarks using the Jikes RVM dynamic compilation environment. Our results show performance improvement of up to 57.71% on an Intel Xeon processor, compared to non-vectorized execution, with a modest increase in compile-time in the range from 0.87% to 9.992%. An investigation of the SIMD parallelization performed by v11.1 of the Intel Fortran Compiler (IFC) on three benchmarks shows that our system achieves speedup with vectorization in all three cases and IFC does not. Finally, a comparison of our approach with an implementation of the Superword Level Parallelization (SLP) algorithm from [20], shows that our approach yields a performance improvement of up to 13.78% relative to SLP.**

***Keywords*-Vectorization; Instruction Selection; Dynamic Programming; Dynamic Optimization**

## I. Introduction

Increasing demand on applications in the multimedia, graphics, embedded, and numerical computing domains have resulted in the addition of SIMD vector units in many current processor architectures. These machines range from general purpose processors [7], [25] to massively parallel supercomputers [23]. The vector units support short SIMD vector instructions that exploit data-level parallelism by performing multiple identical scalar operations in parallel. Given the mounting power constraints in processor design, upcoming architectures are expanding the size of the SIMD unit to perform more operations in parallel, *e.g.,* the AVX extensions in the x86 SSE architectures now support vector sizes of 256 bits instead of 128 bits [4].

We believe that applications need to be automatically vectorized in a compiler framework instead of being manual rewritten for vector execution. While the productivity benefits of compiler vectorization are obvious, the performance benefits stem from the fact that SIMD vectorization needs to be tightly integrated with the register allocator and instruction scheduler phases of a compiler back-end. Some compilers like GCC, Intel C compiler and Sun JDK support only a limited set of loop-based vectorization scenarios that are driven by loop unrolling (*e.g.,* [11]). Such loop-based vectorization techniques can not handle more complex data dependences such as permutation of data and packing of scalar data values within the loop body.

Though the compiler community has studied the problem of automatic parallelization for more than four decades, automatic short SIMD vectorization poses additional challenges that make the task of a compiler hard. We believe that there are three key challenges: (1) performing vectorization in the back-end of a compiler where low-level scalar and vector instructions need to be co-optimized; (2) permutations required for *interleaved data accesses* that often defeat common vectorization patterns such as packing, processing, and unpacking; (3) performing vectorization in a compile-time efficient manner for use in dynamic compilation and in fast edit-compile time development cycles.

The first challenge raises an interesting point as to whether automatic vectorization should be performed closer to the source-level or on a lower-level intermediate representation closer to the machine-level. While source-level vectorization is convenient since aliasing and alignment information are readily available at that level, the selection of vector instructions at the source-level is usually decoupled from standard back-end optimizations such as instruction scheduling and register allocation. On the other hand, vectorization as a back-end pass has the advantage of operating on optimized code and thereby leverages more accurate cost analysis while working closely with the instruction scheduling and register allocation phases. The lack of aliasing and alignment information in a back-end can be addressed by propagating the information from the source-level down to the back-end. In contrast to past work

*This work was done while Rajkishore Barik was at Rice University.

that has focused on source-level loop-based vectorization, this paper focuses on vectorization as a back-end pass in a dynamic compilation environment or any other environment where compile-time efficiency is of interest.

The key contributions of this paper include:

- a novel *dynamic programming approach* to automatic vector instruction selection. The dynamic programming technique is driven by a cost-model that accounts for instruction costs and register pressure.
- a *scalar packing* optimization pass in the instruction selection algorithm that combines scalar operations into vector operations. Other optimizations including algebraic reassociation are also performed to expand the opportunities for vector code generation.
- a *compile-time efficient* solution to automatic vector instruction selection.
- an experimental evaluation of the auto-vectorization framework on six standard numerical benchmarks in the back-end of Jikes RVM dynamic compiler. Our results show a performance improvement of up to 57.71%, compared to the non-vectorized execution with a modest increase in compile time in the range from 0.87% to 9.992%. An investigation of the SIMD parallelization performed by v11.1 of the Intel Fortran Compiler (IFC) on three benchmarks for which Fortran versions are available shows that our system achieves speedup with vectorization in all three cases and IFC does not. Finally, a comparison of our approach with an implementation of the Superword Level Parallelization (SLP) algorithm from [20], shows that our approach yields a performance improvement of up to 13.78% relative to SLP.

The rest of the paper is organized as follows. In Section II, we present a motivating example to illustrate the effectiveness of our approach. The overall automatic vectorization framework is described in Section III. Section IV describes the automatic vector instruction selection algorithm. Experimental evaluations are presented in Section V. We discuss related work in Section VI and conclude in Section VII.

## II. Motivating Example: Complex arithmetic

Consider the innermost loop nest of a double precision complex arithmetic computation kernel of the NAS Parallel Benchmark FT code shown in Figure 1(a). The code fragment initializes the *REAL* and *IMAG* components of local complex numbers *x11* and *x21* from an input complex number (elided for simplicity). This code uses two-element arrays for complex variables and $2N$-element arrays of doubles for $N$-element arrays of complex variables. Local variables *x11* and *x21* are used as operands for a complex add and the result is stored in complex array *scr*. A traditional vectorization algorithm that operates close to the source-level can vectorize most of the complex number operations for array accesses in this example except for two intricate scenarios: (1) the reversed pattern of accesses to the *u1* complex number components in lines 7 and 8, *e.g.,* on line 8, the imaginary component of *u1* is multiplied by the *REAL* component of $x11 - x21$; (2) combining the subtraction and addition operations performed on scalar values within a vector register in lines 7 and 8. If we carefully observe the loop-nest, we can see that several standard compiler optimizations can be applied to this loop-nest such as scalar replacement for arrays, common subexpression elimination, loop-invariant code motion, and dead-code elimination. The resulting optimized code is shown in Figure 1(b). The only memory operations needed are stores to the *scr* arrays.

To vectorize this kernel in a dynamic compilation environment for a managed runtime such as Java Virtual Machine or a .NET runtime, one is faced with several other difficulties. The first and foremost difficulty is to ensure that the kernel is free of exceptions and misalignment issues. In particular, the array accesses should not result in an *array-bounds-check* exception. Additionally, the compiler has to check that *scr* and *u1* are not pointing to the same array (i.e., they are not aliased to each other) to ensure that the loads of *u1* can be safely hoisted outside the loop. Also, the runtime values of *REAL*, *IMAG*, *off1*, and *off2* do not make the array accesses mis-aligned. We can handle these difficulties by specializing and creating an outlined version of the kernel that is aligned to 16-byte boundaries and is guaranteed to be bounds-check free and alias-free.

The 2-address intermediate back-end code generated within a compiler is shown in Figure 1(c). Existing vectorization algorithms when applied at such a low-level of the intermediate representation as in Case (c) usually only vectorize the store and load operations since they do not expend much effort on packing scalar variables. For example, the work on super-word parallelism in [20] would not be able to handle the interchange of scalar values in lines 23 and 24 with respect to lines 18 and 20, since the scalar values in the vector register need to be reversed in the next operation.

Figure 1(d) depicts the code generated by our vectorizer framework. It includes scalar packing for values held in the *x11_REAL* and *x11_IMAG* temporaries using *pack* instructions that correspond to traditional move instructions on a target architecture. The *shuffle* instruction on line 13 takes care of the interleaved data accesses of *u1* array. The *vaddsub* on line 14 combines the subtraction and addition operations in a vector register. Then standard arithmetic operations can be performed on the *REAL* and *IMAG* values.

## III. Overall Framework

Figure 2 depicts the components of our vectorization framework along with their interaction with other components of the dynamic compiler. The dynamic compiler selectively chooses methods for higher level of optimization including the auto-vectorization optimization proposed in this paper. As with other optimizations, the selection criteria for vectorization is guided by profiling information, i.e., based on the profiling information of calling contexts.

The auto-vectorization framework is invoked as a back-end pass after the intermediate code has been specialized and optimized at a higher level. The higher-level optimizations include

```
InnerLoopBody:
1:   x11[REAL] = ...
2:   x11[IMAG] = ...
3:   x21[REAL] = ...
4:   x21[IMAG] = ...
5:   scr[REAL+off1] = x11[REAL] + x21[REAL]
6:   scr[IMAG+off1] = x11[IMAG] + x21[IMAG]
7:   scr[REAL+off2] =
        u1[REAL] * (x11[REAL] - x21[REAL])
      - u1[IMAG] * (x11[IMAG] - x21[IMAG])
8:   scr[IMAG+off2] =
        u1[IMAG] * (x11[REAL] - x21[REAL])
      + u1[REAL] * (x11[IMAG] - x21[IMAG])
```

(a) Original Code

```
1a:   u1_REAL = u1[REAL]
1b:   u1_IMAG = u1[IMAG]
InnerLoopBody:
1:   x11_REAL = ...
2:   x11_IMAG = ...
3:   x21_REAL = ...
4:   x21_IMAG = ...
5:   scr[REAL+off1] = x11_REAL + x21_REAL
6:   scr[IMAG+off1] = x11_IMAG + x21_IMAG
7a:  t1 = x11_REAL - x21_REAL
7b:  t2 = x11_IMAG - x21_IMAG
7:   scr[REAL+off2] = u1_REAL * t1 - u1_IMAG * t2
8:   scr[IMAG+off2] = u1_IMAG * t1 + u1_REAL * t2
```

(b) Optimized code after Scalar Replacement, CSE, LICM, and Dead Code elimination.

```
1:  load u1_REAL, u1[REAL]
2:  load u1_IMAG, u1[IMAG]
InnerLoopBody:
3:  mov   x11_REAL, ...
4:  mov   x11_IMAG, ...
5:  mov   x21_REAL, ...
6:  mov   x21_IMAG, ...
7:  mov   a1, x11_REAL
8:  add   a1, x21_REAL
9:  store scr[REAL+off1], a1
10: mov   a2, x11_IMAG
11: add   a2, x21_IMAG
12: store scr[IMAG+off1], a2
13: mov   t1, x11_REAL
14: sub   t1, x21_REAL
15: mov   t2, x11_IMAG
16: sub   t2, x21_IMAG
17: mov   a3, t1
18: mul   a3, u1_REAL
19: mov   a4, t2
20: mul   a4, u2_IMAG
21: sub   a3, a4
22: store scr[REAL+off2], a3
23: mul   t1, u1_IMAG
24: mul   t2, u1_REAL
25: add   t1, t2
26: store scr[IMAG+off2], t1
```

(c) Intermediate 2-address Code

```
1:  vload  v0, u1[REAL:IMG]
InnerLoopBody:
2:  pack    v1, x11_REAL:x11_IMAG
3:  pack    v2, x21_REAL:x21_IMAG
4:  vmov    v3, v1
5:  vadd    v3, v2
6:  vstore  scr[REAL+off1:IMAG+off1], v3
7:  vmov    v4, v1
8:  vsub    v4, v2
9:  vmov    v5, v4
10: vmul    v5, v0
11: vswap   v6, v0
12: vmul    v4, v6
//shuffle values across vector registers
13: shuffle    v5.upper64, v4.lower64
// horizontal operation using vaddsub
14: vaddsub v5, v4
15: vstore  scr[REAL+off2:IMAG+off2], v5
```

(d) Optimized Intermediate Code

Fig. 1. Example: Double-precision complex floating point arithmetic in NPB-FT benchmark. Assumptions: (1) each vector register is 16 byte wide; (2) all arrays are aligned at 16 byte boundaries; (3) *REAL* and *IMAG* are compile-time constants with values 0 and 1 respectively; (4) *off1* and *off2* are divisible by 2 – making *scr* array accesses aligned.

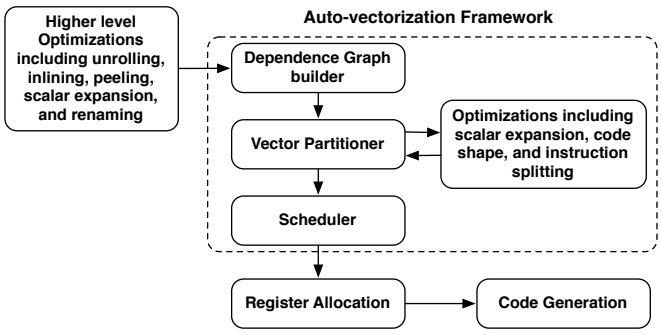


Fig. 2. Overall framework for auto-vectorization in a dynamic compiler. The vector partitioner is introduced as a backend pass.

standard alignment-specific optimizations such as loop peeling, scalar expansion, scalar replacement, and loop-unrolling. The vectorization pass is followed by other back-end passes like instruction scheduling and register allocation. Additional aliasing and alignment information is passed down from the high-level intermediate representation to disambiguate memory references in the vectorization framework.

Within the vectorization framework, we build a data dependence graph for each region of straight-line code (currently at a basic block level). The dependence graph is then used to perform cost-based vector instruction selection. The instruction selection phase expands opportunities for generating vector code by performing aggressive scalar packing, code shape optimizations like algebraic reassociation, and scalar expansion. The generated vector and scalar instructions are then scheduled on the target system. The scheduler needs to pay special attention to various architectural considerations, e.g., the Intel SSE unit supports two register ports that can be simultaneously read and written. Finally the register allocation phase is invoked before generating the assembly code for the target SIMD architecture.

## IV. Automatic Vector Instruction Selection using Dynamic Programming

Given the data dependence graph of the intermediate code that represents the flow of values in a straight-line code of a basic block and the underlying SIMD architecture specific instructions, the goal of optimal vector instruction selection is to find an optimal covering of the dependence graph nodes using both scalar and vector instructions. The nodes in the dependence graph consist of individual intermediate 2-address instructions. The edges in the dependence graph consists of flow dependences[1]. Memory dependences are represented conservatively in the dependence graph such that they do not reorder load and store *IR* instructions. From now on, we will refer to a "scalar tile" as a scalar *IR* instruction that has a one-to-one mapping with a machine instruction and a "vector tile" as a sequence of scalar *IR* instructions that can be replaced by a single vector machine instruction. Each node in the dependence graph can have at most two incoming edges since the *IR* is assumed to be in 2-address code form, but a node can have several outgoing edges indicating sharing of values. This yields a DAG structure for the dependence graph. For convenience, we add a dummy *nop* sink node in the dependence graph to make it connected.

In this section we describe a vector instruction selection algorithm to select both scalar and vector tiles in a compile-time efficient manner. Our vector instruction selection algorithm uses a dynamic programming based approach that computes minimum cost at every node of the dependence graph. Unlike normal instruction selection, we consider several possibilities to pack scalar values into vector registers. We also expand opportunities for vector instruction selection using the optimizations described in Section IV-A.

The dependence graph for a straight-line code region is denoted by $D = \langle N, E\rangle$, where $N$ consists of the set of *IR* instructions and $E = \{\langle n_1, n_2\rangle\}$, where $\langle n_1, n_2\rangle$ denotes a flow dependence from $e_1 \in N$ to $e_2 \in N$. As mentioned earlier, other forms of register dependences such as anti and output are handled by a renaming phase prior to the instruction selection pass. The dependence graph, $D$ is pre-processed to be made single-sink by adding a *nop* node.

Let $k$ denote the width of the underlying SIMD vector unit. Let $T$ be the set of scalar and vector tiles enumerating the machine instructions. We say a scalar tile $t \in T$ matches a node $n \in N$ in $D$ if $n$ can be performed using the machine instruction $t$ (*i.e.,* there is one-to-one correspondence between an *IR* instruction and a tile) and the direct-predecessors of $n$ match with those in $t$. Similarly, we say a vector tile $t \in T$ matches a tuple of $k$ nodes $\langle n^1, n^2, \cdots, n^k\rangle$, if $\langle n^1, n^2, \cdots, n^k\rangle$ can be performed using the machine instruction $t$. We denote the operation specified at a node $n \in N$ as $op(n)$ and the cost of performing the operation as $op_cost(n)$. We define a scalar cost map, $scost : N \rightarrow Z^+$ as the cost of performing a node $n \in N$ using a matched scalar tile.

[1]Anti- and output- dependences on scalar operations can be eliminated by a renaming phase before instruction selection.

We define $vcost : (N \times N \times \cdots \times N) \rightarrow Z^+$ as the cost of performing a vector tuple $\langle n^1, n^2, \cdots, n^k\rangle$ of instructions in a vector tile. Let $pack_cost : (N \times N \cdots \times N) \rightarrow Z^+$ be the cost of packing $\langle n^1, n^2, \cdots, n^k\rangle$ into a vector register. Let $unpack_cost : (N \times N \cdots \times N) \times N \rightarrow Z^+$ be the cost of unpacking a node $n^i$ from a vector register containing $\langle n^1, n^2, \cdots, n^k\rangle$. Let $shuffle_cost : (N \times N \cdots \times N) \rightarrow Z^+$ be the cost of shuffling the contents of a vector register containing $\langle n^1, n^2, \cdots, n^k\rangle$. We assume uniform cost for shuffling any number of scalar values within a vector register. A tuple of *IR* instructions $v = \langle n^1, n^2, \cdots, n^k\rangle$ is *vectorizable* if there exists a vector tile that can combine the scalar operations in the vector tuple. Let the predicate $vect(n^i)$ indicate *true* if $n^i$ is packed with other *IR* instructions in a vector tile. Let $shuffle(\langle n^1, n^2, \cdots, n^k\rangle)$ represent any desired shuffled combination of $\langle n^1, n^2, \cdots, n^k\rangle$.

*Definition 4.1:* Let us now describe the legal rules for combining scalar operations to generate a vector tuple $v = \langle n^1, n^2, \cdots, n^k\rangle$:

- no pair of nodes, $n^i$ and $n^j$ in $v$ are dependent on each other, *i.e.,* there exists no path of flow dependence between them in the dependence graph;
- Replacing $\langle n^1, n^2, \cdots, n^k\rangle$ by a single node in the full dependence graph (including anti, output dependence) does not create a cycle [27];
- $v$ is *vectorizable*;
- if each operation $n^i$ in $v$ is a memory operation such as load and store *IR* instructions, then the memory operations must be from contiguous memory locations and aligned.

Let $V$ denote the set of vector tuples generated from the instruction selection algorithm. Given the above definitions, we now describe the steps to compute costs for scalar nodes in the dependence graph. The cost of matching a dependence graph node $n$ of the form shown in Figure 3 with a scalar tile is given in Figure 4.

The scalar cost computation involves considering several scenarios and choosing the minimum cost among them: (1) if the predecessors of $n$ were paired in a vector register already, then we consider the cost of performing a horizontal operation on the vector register using a matched tile; (2) if the predecessors of $n$ were paired in a vector register in a reversed order, then we shuffle the values in the vector register and perform a horizontal operation on the vector register; (3) if any of the predecessors was packed with another node in a vector register already, then extract the scalar value from the vector register; (4) if none of the predecessors were packed in any vector register – then add up the individual scalar cost of each predecessor. Note that cases (1) and (2) are special cases of the more general case (3) in 2-way SIMD unit. Finally this cost is added to the cost of the matched scalar tile to generate the cost of covering a scalar tile for a node $n$ in $D$.

The cost of covering a set of dependence graph nodes (as shown in Figure 5) with a vector tile is shown in Figure 6. The notation $n^i_t$ used in Figure 6 denotes the $t$-th operand of a scalar operation at node $n^i$. The cost computation involves

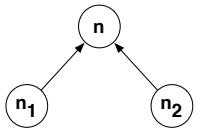


Fig. 3. Scalar Cost Computation Scenario

$$
scost(n) = min \begin{cases}
op_cost(op(n)) & if\ \langle n_1, n_2\rangle \in V \wedge\ \ op(n) \in T \\
shuffle_cost(\langle n_2, n_1\rangle) + op_cost(op(n))) & if\ \langle n_2, n_1\rangle \in V \wedge\ op(n) \in T \\
unpack_cost(\langle p_1, \cdots, n_1, \cdots, p_k\rangle, n_1) + unpack_cost(\langle m_1, \cdots, n_2, \cdots, m_k\rangle, n_2) + op_cost(op(n)) & if\ \langle p_1, \cdots, n_1, \cdots, p_k\rangle \in V \wedge\ \langle m_1, \cdots, n_2, \cdots, m_k\rangle \in V \\
unpack_cost(\langle p_1, \cdots, n_1, \cdots, p_k\rangle, n_1) + scost(n_2) + op_cost(op(n)) & if\ \langle p_1, \cdots, n_1, \cdots, p_k\rangle \in V \\
scost(n_1) + unpack_cost(\langle p_1, \cdots, n_2, \cdots, p_k\rangle, n_2) + op_cost(op(n)) & if\ \ \langle p_1, \cdots, n_2, \cdots, p_k\rangle \in V \\
scost(n_1) + scost(n_2) + op_cost(op(n)) & otherwise
\end{cases}
$$

Fig. 4. Cost computation for scalar tile selection

a series of possibilities for packing and unpacking individual scalar operations. The equations also take into account for *shuffle* operations in vector register. The final vector tile cost is obtained by taking a minimum of all the possibilities.

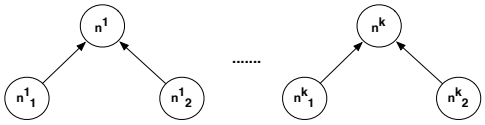


Fig. 5. Vector cost computation scenario

Using the *scost* and *vcost* equations from Figure 4 and Figure 6, the dynamic programming algorithm is presented in Figure 7. There are three passes to the algorithm. The first pass walks over the dependence graph in a top down fashion (source to sink) and computes minimum scalar and vector costs for every possible scalar and vector operation in the dependence graph respectively. The cost of the designated sink node is the best cost estimation of generating machine code for the set of *IR* instructions. In a second pass, the best cost is used to determine the best scalar or vector tile at every node in the dependence graph. This is obtained by making a bottom-up pass (sink to source) over the dependence graph. Finally, a top down pass over the dependence graph is performed and best tile for each dependence graph node is used to generate code.

In Figure 9, we show the steps of applying Algorithm 7 to the example program in Figure 1(c). The final vector code is shown in Figure 1(d).

### A. Additional Optimizations

Algorithm 7 judiciously uses shuffle instructions to choose best estimates for scalar and vector cost. However, there are several other optimization scenarios which can easily be incorporated into our auto-vectorization algorithm. These scenarios are depicted in Figure 10. Case 1 is the scenario where values are shuffled across multiple vector registers. Case 2 shows the standard scenario of swapping values within a vector register using shuffle instructions. Cases 3 and 4 depict scenarios where scalar expansion technique can be used to expand either the same scalar operation or multiple scalar operations into a vector register. Case 5 is the case where *algebraic reassociation* is used to reshape the code for vectorization. In particular, we use algebraic reassociation for the following scenarios: (1) $t1-t2$ is transformed to $t1+(-t2)$ for expanding the opportunity for another "+" operation to be vectorized; (2) $t1*(t2+t3)$ is rewritten as $t1*t2+t1*t3$ using distributivity – this scenario is shown in Case 5. Reshaping of code requires that the operations be commutative, associative, and distributive. Special care must be taken while dealing with floating point values *i.e.,* if strict conformance is required, then standard algebraic reassociation can not be performed, *e.g.,* rewriting $a/b$ to $a*(1/b)$.

### B. Moderating register pressure

One of the key considerations in optimized machine code generation is to utilize the register files efficiently. To that end, we can extend our cost model to support register pressure. Let $R$ be the set of available scalar and vector registers. We redefine scalar cost, *scost* : $N \times R \rightarrow Z^+$ as the cost of performing a node $n \in N$ using a matched scalar tile and $r \in R$ number of registers. Similarly, we redefine *vcost* : $(N \times N \times \cdots \times N) \times R \rightarrow Z^+$ as the cost of performing a vector tuple $\langle n^1, n^2, \cdots, n^k\rangle$ in a vector tile using $r \in R$ number of registers. These new *scost* and *vcost* cost functions are incorporated into our algorithm in Figure 7 on lines 14 and 21 respectively.

### C. Discussion

It is well-known that optimal instruction selection on DAG data structures is NP-hard [6], [3]. We presented a dynamic programming based approach to perform both scalar and vector instruction selection simultaneously. The algorithm ignores the fact that there can be shared nodes in the dependence graph whose cost estimation may not yield optimal results

$$
\begin{aligned}
\mathit{vcost}(\langle n^1, \cdots, n^k \rangle) \quad = \quad \mathit{min} \Bigg\{ & \\
& \mathit{vcost}(\langle n_1^1, \cdots, n_1^k \rangle) + && \mathit{if}\ \langle n_1^1, \cdots, n_1^k \rangle \in V \\
& \quad \mathit{vcost}(\langle n_2^1, \cdots, n_2^k \rangle) + && \quad \wedge\ \langle n_2^1, \cdots, n_2^k \rangle \in V \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \mathit{shuffle_cost}(\langle p_1, \cdots, p_k \rangle) + && \mathit{if}\ \langle p_1, \cdots, p_k \rangle = \mathit{shuffle}(\langle n_1^1, \cdots, n_1^k \rangle) \in V \\
& \mathit{shuffle_cost}(\langle m_1, \cdots, m_k \rangle) + && \quad \wedge \langle m_1, \cdots, m_k \rangle = \mathit{shuffle}(\langle n_2^1, \cdots, n_2^k \rangle) \in V \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \mathit{vcost}(\langle n_1^1, \cdots, n_1^k \rangle) + && \mathit{if}\ \langle n_1^1, \cdots, n_1^k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_2^p)=\mathit{true}} \mathit{unpack_cost}(\langle p_1, \cdots, n_2^p, \cdots, p_k \rangle, n_2^p) + && \quad \wedge \langle p_1, \cdots, n_2^p, \cdots, p_k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_2^p)=\mathit{false}} \mathit{scost}(n_2^p) + \mathit{pack_cost}(\langle n_2^1, \cdots, n_2^k \rangle) + && \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \mathit{shuffle_cost}(\langle p_1, \cdots, p_k \rangle) + && \mathit{if}\ \langle p_1, \cdots, p_k \rangle = \mathit{shuffle}(\langle n_1^1, \cdots, n_1^k \rangle) \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_2^p)=\mathit{true}} \mathit{unpack_cost}(\langle p_1, \cdots, n_2^p, \cdots, p_k \rangle, n_2^p) + && \quad \wedge \langle p_1, \cdots, n_2^p, \cdots, p_k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_2^p)=\mathit{false}} \mathit{scost}(n_2^p) + \mathit{pack_cost}(\langle n_2^1, \cdots, n_2^k \rangle) + && \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \mathit{vcost}(\langle n_2^1, \cdots, n_2^k \rangle) + && \mathit{if}\ \langle n_2^1, \cdots, n_2^k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_1^p)=\mathit{true}} \mathit{unpack_cost}(\langle p_1, \cdots, n_1^p, \cdots, p_k \rangle, n_1^p) + && \quad \wedge \langle p_1, \cdots, n_1^p, \cdots, p_k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_1^p)=\mathit{false}} \mathit{scost}(n_1^p) + \mathit{pack_cost}(\langle n_1^1, \cdots, n_1^k \rangle) + && \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \mathit{shuffle_cost}(\langle p_1, \cdots, p_k \rangle) + && \mathit{if}\ \langle p_1, \cdots, p_k \rangle = \mathit{shuffle}(\langle n_2^1, \cdots, n_2^k \rangle) \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_1^p)=\mathit{true}} \mathit{unpack_cost}(\langle p_1, \cdots, n_1^p, \cdots, p_k \rangle, n_1^p) + && \quad \wedge \langle p_1, \cdots, n_1^p, \cdots, p_k \rangle \in V \\
& \quad \textstyle\sum_{\forall p=1..k \wedge \mathit{vect}(n_1^p)=\mathit{false}} \mathit{scost}(n_1^p) + \mathit{pack_cost}(\langle n_1^1, \cdots, n_1^k \rangle) + && \\
& \quad \mathit{op_cost}(\mathit{op}(n)) && \\
& \textstyle\sum_{\forall p=1..k} \mathit{scost}(n_1^p) + \mathit{pack_cost}(\langle n_1^1, \cdots, n_1^k \rangle) && \mathit{otherwise} \\
& \textstyle\sum_{\forall p=1..k} \mathit{scost}(n_2^p) + \mathit{pack_cost}(\langle n_2^1, \cdots, n_2^k \rangle) && \\
& \mathit{op_cost}(\mathit{op}(n)) &&
\end{aligned}
$$

Fig. 6. Computation of cost for vector tile selection

due to sharing across multiple paths. Several approaches have been proposed in the literature to improve instruction selection within a DAG *e.g.,* (1) the DAG can be broken into a series of trees and each individual tree can be selected individually [27]; (2) the shared nodes can be separately processed after normal instruction selection to take the common subexpression costs into account [15]. For this paper, we use approach (1) in our implementation.

Scheduling of the generated scalar+vector code is important. After the algorithm in Figure 7 is performed, we schedule the code using a list scheduler that schedules basic blocks based on critical path length. The critical path length is computed using the dependence graph and relative cost estimates of each scalar and vector instruction.

## V. Experimental Results

In this section, we present an experimental evaluation of the automatic vectorization framework described in Sections III and IV in the Jikes RVM dynamic optimizing compiler [10]. We describe the overall compilation environment and then present results on performance improvement compared to an implementation of the vectorization approach described in [20]. We also report on compilation time overheads, and reductions in static code size.

*Benchmarks:* In this paper, we study the impact of automatic vector instruction selection on numerical applications. The benchmarks we consider include three serial Java numeric applications (`FT`, `CG`, and `MG`) obtained from the NAS Parallel Benchmark (NPB) Suite [9] and three additional serial Java applications (`SOR` and `LUFact` from Section 2 and `MolDyn` from Section 3) from the Java Grande Forum (JGF) [13] benchmark suite. The other benchmarks in these two suites did not offer much scope for performance improvement using automatic vectorization. Our choice of Java benchmarks follows from the use of Jikes RVM, which executes Java programs; however, the techniques in this paper should be applicable to C and Fortran programs as well. The scientific kernels in both JGF and NPB use double-precision floating-point operations. Note that our proposed algorithm in Section IV applies to any vector length.

*Implementation Details:* The vectorization framework described in Section IV was implemented in version 3.0.0 of the Jikes RVM Research Java Virtual Machine [1]. The boot image for Jikes RVM used a 32-bit x86 production configu-

```
1  Func AutoVectorize
     Input  : IR for method m and its dependence graph D = ⟨N, E⟩ (D is made connected by a dummy nop node – op_cost of a nop node is 0)
     Output: IR with vector instructions
2    FindBestCost (D); //First Pass
3    bestTile := FindBestTile (D); //Second Pass
4    Generate code using a topological sort over D and bestTile choice for each node; //Third Pass
5    return;

6  Func FindBestCost (D)
7    W := all the root nodes of D; //nodes with no incoming edges
8    scost := φ; //scost not yet computed
9    vcost := φ; //vcost not yet computed
10   visited := false; //no node is visited yet
11   vecSet := φ; //set of vector tuples generated
12   while !W.empty() do
13       n^k := remove a node from W;
14       Compute scost(n^k) using Figure 4;
15       Save the tile selection that yields the minimum value for scost(n^k);
16       visited(n^k) := true;
17       for each vector tuple ⟨n^1, ··· , n^k⟩ do
18           if visited(n^i) == true, ∀i = 1..k and ⟨n^1, ··· , n^k⟩ can be legally combined using Definition 4.1 then
19               v := ⟨n^1, ··· , n^k⟩;
20               add v to vecSet;
21               Compute vcost(v) using Figure 6;
22               Save the tile selection that is used to compute minimum value for vcost(v);
23       for each s ∈ succ(n^k) do
             //each successor of n^k
24           if all the pred(s) are visited then
25               add s to W;
26   return;

27 function FindBestTile (D)
28 W := leaf node of D;
29 while !W.empty() do
30     n := remove a node from W;
31     bestTile[n] := Find the scalar or vector tile that produced lowest cost for n; //Resolve ties arbitrarily
32     for each p ∈ pred(n) do
33         add p to W;
34 return;
```

Fig. 7. Dynamic programming-based automatic vector tile selection.

ration. Like other JVMs, this version of Jikes RVM uses SSE registers for both scalar single and double precision arithmetic (instead of using the x86 Floating Point unit). The main reason for doing so is the runtime improvements achieved from using the SSE unit instead of the normal floating point unit. There are 8 (XMM0-XMM7) SSE registers that are exposed in a 32-bit configuration, out of which (XMM7) is used as a scratch register. Effectively there are seven XMM registers available for scalar single and double precision arithmetic operations and also, for vector operations.

We extend the Jikes RVM IA32 assembler generator to support generation of SSE instructions used for *swap*, *shuffle*, *horizontal arithmetic*, *packing*, and *unpacking* operations [12] in XMM registers. The alignment offset of each array object is chosen so that element 0 is aligned on a 16-byte boundary.

For the dynamic compilation configuration, we set the initial compiler as the optimizing compiler with `-O2` for the optimization level to enable a standard set of optimizations to be performed prior to vectorization, including Common Subexpression Elimination (CSE), Dead Code Elimination (DCE) and Scalar Replacement (SR) [2].

A set of preprocessing transformations is performed at the bytecode level within the Soot Java Bytecode Transformation Framework [28] prior to program execution and dynamic compilation. These include Loop Unrolling (LU), Code Specialization (CS), Scalar Expansion (SE), and Loop Peeling (LP). The specialized code is ensured to be free of alignment issues, aliasing dependences, and exception problems that include bounds-check and null-pointer exceptions.

Table I summarizes the optimizations that were actually performed on each benchmark.

*Experimental Platform:* All results except Figure 12 were obtained on a Quad-Core Intel E5430 Xeon 2.66GHz system with 8GB memory, running Red Hat Linux (RHEL 5). For Figure 12 we use a Quad-Core Intel E5440 Xeon 2.83GHz system due to the availability of Intel's Fortran compiler on the system. The execution times reported were the best of three

[2]The public release version of Jikes RVM 3.0.0 enables Scalar Replacement transformation at the highest optimization level (-O3). We changed this setting by enabling Scalar Replacement at optimization level -O2 for our framework.

| Instruction | Cost |
|---|---|
| memory move | 4 |
| register move | 1 |
| vector memory move | 4 |
| vector register move | 1 |
| packing instruction | 2 |
| unpacking instruction | 2 |
| shuffle within a vector register | 2 |
| shuffle across vector registers | 5 |
| vector addition | 5 |
| vector subtraction | 5 |
| vector multiply | 10 |
| vector divide | 10 |
| vector addsub | 5 |

Fig. 8. The costs of vector and scalar machine instructions were determined on a x86 machine using microbenchmarks.

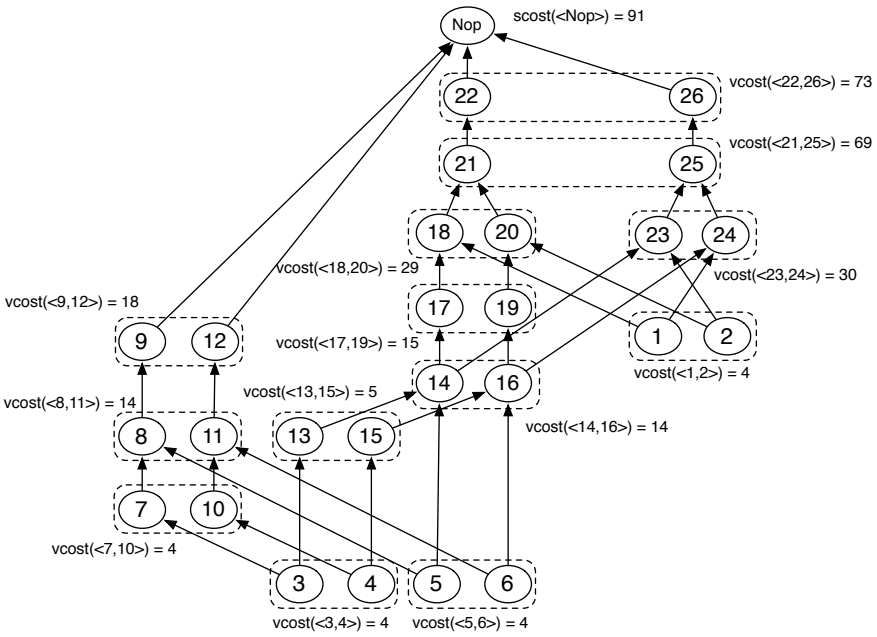


Fig. 9. Final cost estimates obtained by applying method $FindBestCost$ of Algorithm 7 to Example 1(c). We assume that the *mov* instructions on lines 3, 4, 5, and 6 in Figure 1(c) are memory load operations. We also assume that the instructions on lines 1 and 2 are vectorized in a predecessor block and are used in the loop-nest.

| Optimizations Performed | FT | CG | MG | SOR | LUFact | MolDyn |
|---|---|---|---|---|---|---|
| Scalar Expansion (Soot) | × | √ | √ | √ | × | √ |
| Loop Unrolling (Soot) | √ | √ | √ | √ | √ | √ |
| Loop Peeling (Soot) | √ | √ | √ | √ | √ | √ |
| Code Specialization (Soot) | √ | √ | √ | √ | √ | √ |
| Common Subexpression Elimination (Jikes) | √ | × | √ | × | √ | √ |
| Dead Code Elimination (Jikes) | √ | √ | √ | √ | √ | √ |
| Scalar Replacement (Jikes) | √ | √ | √ | √ | × | √ |

TABLE I
LIST OF BENCHMARKS AND THE OPTIMIZATIONS APPLIED ON EACH BENCHMARK

runs within a single JVM instance for each benchmark. Since all benchmarks studied in this paper are sequential, the results are not impacted by the number of cores.

*Experimental Results:* We compared our automatic vectorization algorithm (described in Section IV) with a prototype implementation of the superword-level vectorization algorithm from [20][3]. We refer to our approach as `SP` (for Scalar Packing) and superword level vectorization as `SLP`. For both approaches, we include liveness-based dead-code elimination and copy propagation passes (referred to as `Opt`) after register allocation pass to eliminate redundant scalar packing and unpacking instructions. This pass produces tighter vector code. We measured the impact of `Opt` on both `SP` and `SLP` algorithms.

Experimental results are reported for the following five cases:

- `Sequential`: without vectorization, but with unrolling (see below);
- `SLP`: Superword vectorization algorithm from [20];
- `SLP+Opt`: Superword vectorization with register allocator level `Opt` pass;
- `SP`: Our approach described in Section IV;
- `SP+Opt`: Our approach with register allocator level `Opt` pass;

Since loop unrolling is a common compiler transformation for enabling SIMD parallelism, we performed loop unrolling on all innermost loops in all benchmarks and studied its impact for unroll factors 2, 4, 8, and 16.

For all the benchmarks shown in Table I, we measured the speedup relative to the serial execution of the benchmark without vectorization. In Figure 11, we show the speedups of each benchmark relative to non-unrolled sequential execution of the five cases listed above. In `FT`, `SOR`, and `MolDyn` benchmarks, `SP` introduced more opportunities for packing the scalars into

[3]We implemented this superword vectorization algorithm in Jikes RVM 3.0.0 for the purpose of this research.

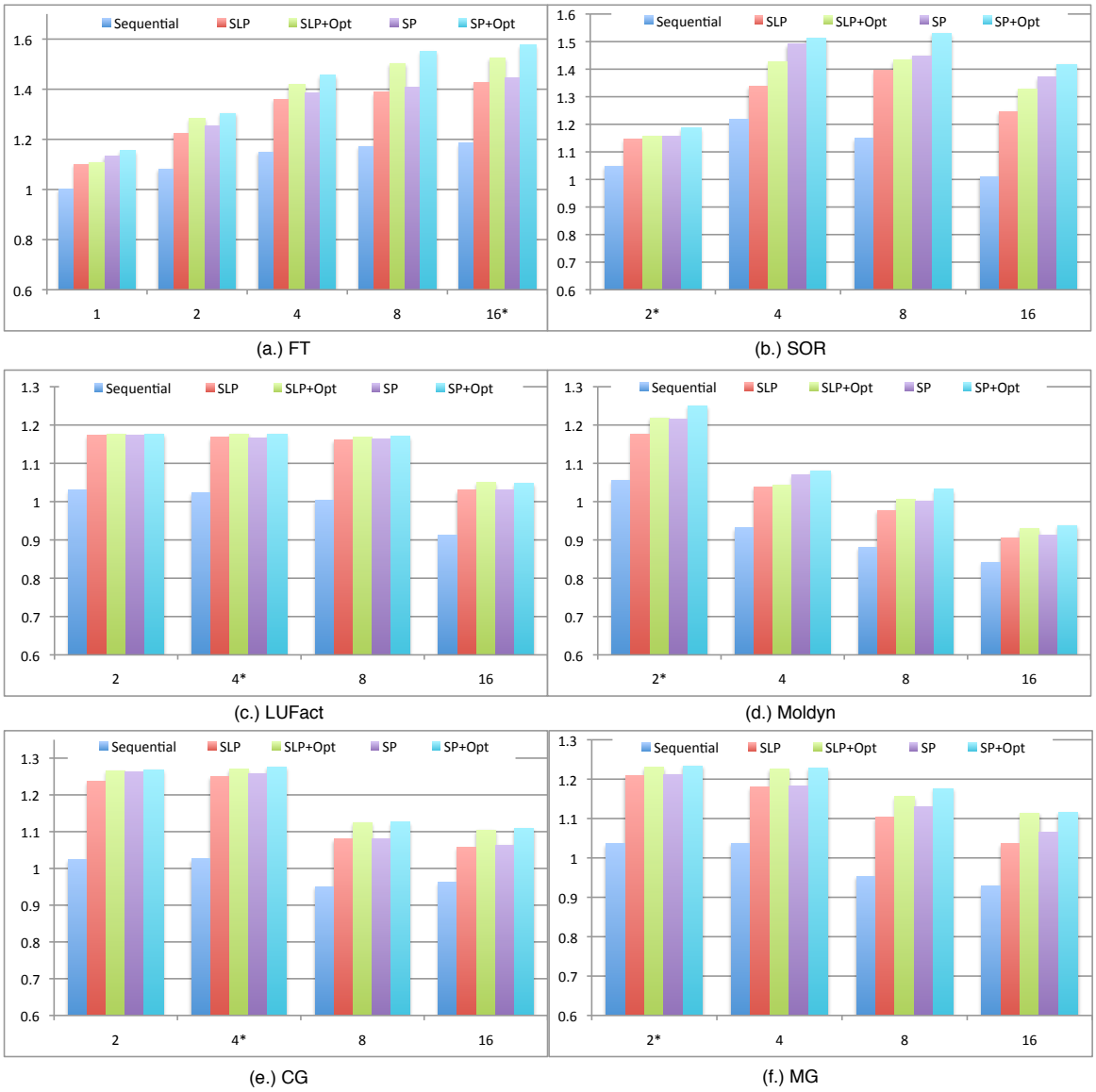


Fig. 11. Speedups as a function of unroll factor for the five cases, relative to the performance of non-unrolled non-vectorized code. X-axis denotes the unroll factors and Y-axis shows the speedups. The asterisk (*) next to an unroll factor denotes the unroll factor that produced best cost using our cost model presented in Section IV.

vector than `SLP` (The `FT` benchmark uses complex numbers and can benefit from vectorization even when the unroll factor is 1). For `FT`, the `SP` algorithm introduces vector swap and horizontal computations that result in fully vectorizing the innermost loop body using SSE instructions, as discussed in Section II. For benchmarks that contain regular patterns for vector code generation (e.g. `LUFact`, `CG`, `MG`), both `SP` and `SLP` show similar speedups. The `Opt` post-pass provides a performance improvement in all cases, since it eliminates most of the redundant scalar variable packing and extraction instructions after the register allocator assigns physical vector registers to the operands. These redundant instructions may include vector register moves, e.g., moves between IA32 SSE registers. The copy propagation pass eliminates any remaining redundant copy statements in the generated vector code.

To summarize individual benchmark performance improvements of `Sp+Opt` compared to `Sequential` versions, for FT, we observe a maximum improvement of 57.71% (using unroll factor 16). For SOR, we observe a maximum improvement of 52.91% (using unroll factor 8). For LUFact, we observe a maximum improvement of 17.60% (using unroll factor 4). For Moldyn, we observe a maximum improvement of 25.06% (using unroll factor 2). For CG, we observe a maximum improvement of 27.63% (using unroll factor 4). Finally for MG, we observe a maximum improvement of 23.41% (using unroll factor 2). When comparing `SP+Opt` with `SLP`, we achieve a maximum speedup of 13.78% (in SOR using unroll factor 16). When comparing `SP+Opt` with `SLP+Opt`, we achieve a maximum speedup of 6.74% (in SOR using unroll factor 8).

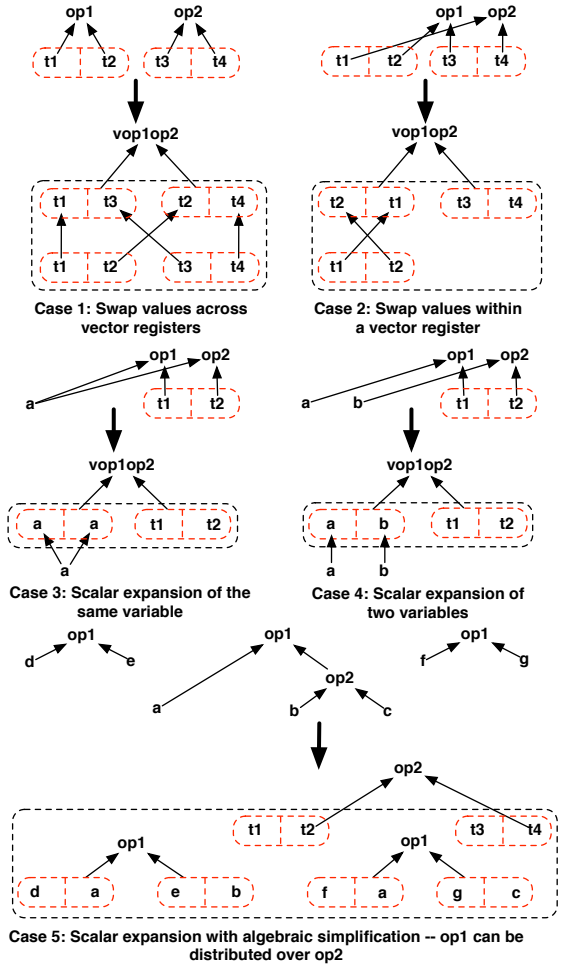


Fig. 10. Various scenarios for expanding opportunities for vector instruction selection.

*Automatic Selection of Unroll Factor:* In addition to performance speedups for benchmarks, we use the cost model to predict the unroll-factor that gives the best performance for a benchmark within the dynamic compiler. For every unroll factor of the vectorizable innermost loop-nest in a specialized method, we create a clone of the *IR*. The cloned copies of the *IR* subsequently undergo the same standard compiler optimizations including our vector instruction selection pass. Our vectorization algorithm produces a unique cost for each such cloned copy of the *IR*. Based on the best cost, we choose the corresponding *IR* for execution and discard the others. Note that our cost model captures both the register pressure and relative cost of both scalar and vector instructions.

The best unroll factors according our cost model are indicated by "*" in Figure 11. Except for `SOR` benchmark, we are able to predict the correct unroll factor within the Jikes RVM dynamic compiler. The reason for mis-prediction in `SOR` benchmark can be attributed to the cache memory effects on the cost model, which is a subject of future work.

*Compile-time Overhead:* `SP` is implemented within the Jikes RVM dynamic compilation system, which means that the vectorization is invoked during runtime. In this section, we report the compile-time overhead for dynamic compilation. Table II presents the compile-time for the vector instruction selection algorithm described in Section IV (`Vec`) and the total compilation time (`Total`) for each benchmark using different unroll factors. For the non-unrolled version of FT, the compile-time overhead of vectorization is `23 msec` and the total compilation time is `480 msec`. The cost of compilation is correlated with the increase in code. Hence, an unroll factor of 16 results in larger compilation time than smaller unroll factors. In all the cases we studied, the compile-time overhead of `SP+Opt` ranges from `0.87%` to `9.92%` of the total compilation time. This modest increase in compilation time indicates that a dynamic compiler can use our dynamic programming based vector selection algorithm in order to achieve the runtime benefits shown in Figure 11.

| | Factor 2 | | Factor 4 | | Factor 8 | | Factor 16 | |
|---|---|---|---|---|---|---|---|---|
| | Vec | Seq | Vec | Seq | Vec | Seq | Vec | Seq |
| FT | 140 | 188 | 220 | 316 | 380 | 572 | 700 | 1082 |
| CG | 676 | 723 | 730 | 855 | 849 | 1063 | 1241 | 1676 |
| MG | 399 | 471 | 446 | 593 | 645 | 906 | 1077 | 1497 |
| SOR | 39 | 57 | 57 | 74 | 91 | 142 | 190 | 234 |
| LUFact | 193 | 198 | 209 | 221 | 317 | 332 | 342 | 358 |
| Moldyn | 302 | 318 | 596 | 625 | 1247 | 1299 | 2570 | 2586 |

TABLE III
STATIC NUMBER OF X86 INSTRUCTIONS GENERATED.

*Static Code Size Reduction:* By applying vectorization, the machine code size can also be reduced, since more instruction level parallelism (ILP) are employed. Table III compares the static number of x86 instructions generated by the `SP+Opt` vectorization algorithm (`Vec`) and the sequential execution (`Seq`) for each benchmark using different unroll factors. The data listed in this table shows the total number of instructions for only those methods that are vectorized by `SP+Opt` in each benchmark (i.e., the specialized methods). The reductions in the number of instructions range from `0.627%` (`MolDyn` with unroll factor 16) to `35.3%` (`FT` with unroll factor 16). For the non-unroll FT, the size of vectorized code is `100` and non-vectorized version is `118`.

*Comparison with Intel Compiler auto-vectorization:* We now report on an investigation of the SIMD parallelization performed by v11.1 of the Intel Fortran Compiler (IFC) on three NAS parallel benchmarks for which Fortran versions are available (`MG`, `CG`, `FT`). We compiled and executed each benchmark in the `SER` version of the NPB benchmark suite. Figure 12 reports on the absolute performances of these benchmarks and compares them with their corresponding Java versions executed using Jikes RVM on an Intel Xeon E5440 2.83GHz system. IFC-Seq and IFC-Vec refer to the versions compiled with the `-O3 -no-vec` and `-O3` options respectively. The performance gaps between the Fortran and Java versions are clearly visible in Figure 12. However, the FORTRAN benchmarks surprisingly did not show any noticeable performance improvements due to vectorization, whereas the Java versions showed improvements of 17.5% for `MG`, 9.6%

| | Unroll Factor 2 (msec) | | Unroll Factor 4 (msec) | | Unroll Factor 8 (msec) | | Unroll Factor 16 (msec) | |
|---|---|---|---|---|---|---|---|---|
| | Vec | Total | Vec | Total | Vec | Total | Vec | Total |
| FT | 24 | 480 | 26 | 472 | 37 | 523 | 79 | 796 |
| CG | 29 | 556 | 29 | 556 | 31 | 579 | 38 | 640 |
| MG | 5 | 112 | 6 | 108 | 6 | 114 | 9 | 119 |
| SOR | 5 | 94 | 5 | 94 | 6 | 100 | 10 | 111 |
| LUFact | 23 | 477 | 23 | 489 | 23 | 506 | 33 | 605 |
| Moldyn | 12 | 237 | 12 | 247 | 13 | 313 | 16 | 1608 |

TABLE II
COMPILE-TIME IN MILLI SECONDS FOR `SP+Opt` USING UNROLL FACTORS 2, 4, 8, AND 16.

for `CG`, and 15.54% for `FT`, when using our approach. A vectorization report obtained by using the `-vec-report(3)` option showed that many loops were vectorized by IFC, but they unfortunately did not contribute to a measurable speedup.

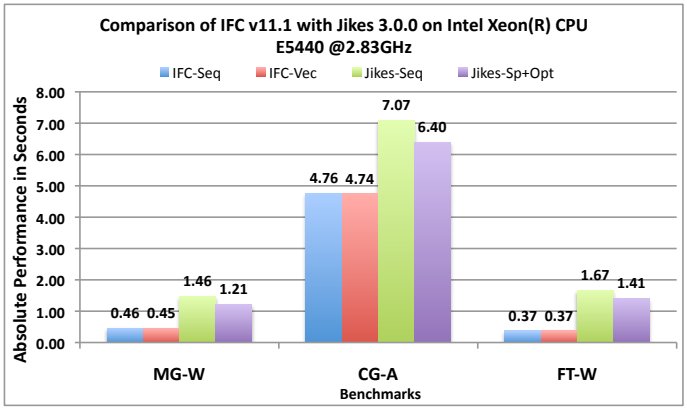


Fig. 12. Comparison of vectorization in Intel Fortran Compiler v11.1 and Jikes RVM 3.0.0 (SP+Opt)

## VI. RELATED WORK

Most past techniques for automatic vectorization are either based on traditional loop-based vectorization [14], [18] or use instruction packing to explore data parallelism within basic lock level. Several compilers employ loop-level vectorization techniques to perform automatic vectorization for innermost loop-nest, *e.g.,* Intel Compiler [5], IBM XL compiler [8] and GNU Compiler [2]. This paper focuses on straight-line code vectorization in the back-end of a dynamic compiler.

Larsen and Amarasinghe [20], [21] introduced the Superword-Level Parallelism (SLP) algorithm which initiates the vectorization process by identifying memory operations that operate on contiguous addresses, forms groups, and then packs the relevant instructions and merges small groups until each group's size reaches the size of SIMD vector. To improve the vectorization, they also suggested techniques to increase the number of contiguous memory operations in [22]. This superword technique shows good scalability to adapt to different SIMD widths. One of the key limitations of this algorithm is that it may not uncover vectorization opportunities for complicated interleaved data accesses, which our paper addresses. Additionally, their algorithm can not vectorize pure scalar code, which our algorithm can by using *scalar packing*.

Kudriavtsev and Kogge [19] proposed a generalized approach for handling the permutation issue during vectorization. The foundation of this method is again driven by contiguous memory accesses as in [20]. It performs bi-directional search on a data flow graph (DFG) to pack instructions into SIMD groups. One direction starts from store operations and another from load operations. Once it identifies mismatched SIMD groups, it enumerates all permutation solutions and uses an Integer Linear Program approach to select the best one. This technique can solve the problem of vector instruction selection with permutation, but it can't handle scalars in the case when no contiguous memory operations exist in the data flow graph. Also, this bi-directional approach may generate some redundant mismatch operations compared to the single-direction packing. Integer Linear Programming also incurs significant compile-time overhead making it impractical for use in dynamic compilation.

In [24], Leupers introduced vector instruction selection based on the Data Flow Tree (DFT) and Integer Linear Programs (ILP). For each DFT, the parser generates a set of DFT covers that includes both SIMD and non-SIMD covers. The ILP approach is applied to choose the best cover for generating SIMD instructions. This approach also has difficulty in handling permutations, and is not practical for dynamic compilation.

The authors in [17], [16] propose a vectorization technique based on rewriting rules using the dependence graph that requires depth-first search, backtracking, and branch-and-bound techniques. In contrast, our paper introduces a cost-driven dynamic programming algorithm for vector instruction selection which obtains a solution with best cost that includes register pressure and peephole optimization considerations during instruction selection. Additionally, our approach only requires three passes over the dependence graph and avoids the need for any search, back-tracking, and branch-bound techniques. Our approach is compile-time efficient since it increases the compile-time by less than 10%. Searching, backtracking, and branch-bound techniques will undoubtedly incur significant compile-time overheads.

The main focus of [26] is to perform interprocedural compiler analysis to determine alignment of pointers during

compile-time. They also describe a vector code generation algorithm that is tightly integrated with scheduling. The key difference between our approach and their vector code generation algorithm is that their approach combines multiple scalar operations using a heuristic which is based on both the number of resulting vector instructions and the number of adjacent subwords in the vector instruction, where as our approach uses a profitability based approach that uses estimated instruction costs to perform vector instruction selection. Their approach does not necessarily yield the best cost vector code as scalar operations are combined on-the-fly and scheduled during a single pass of the dependence graph.

## VII. Conclusion and Future Work

In this paper, we presented the design and implementation of an auto-vectorization framework integrated in the IA32 back-end of the Jikes RVM dynamic compiler. Our approach includes a novel dynamic programming based vector instruction selection algorithm that is compile-time efficient. Our approach also expands opportunities for generating vector code in the following ways: (1) *scalar packing* explores opportunities for packing multiple scalar variables into short vectors; (2) judicious use of *shuffle* and *horizontal* vector operations; and (3) *algebraic reassociation* expands opportunities for vectorization by algebraic simplification. For the six benchmarks studied in this paper, our results show performance improvement of up to 57.71% (in the `FT` benchmark using an unroll factor of 16), compared to the non-vectorized code, with a modest increase in compile time in the range from 0.87% to 9.992%. An investigation of the SIMD parallelization performed by v11.1 of the Intel Fortran Compiler (IFC) on three benchmarks shows that our system achieves speedup with vectorization in all three cases and IFC does not. Finally, a comparison of our approach with an implementation of the Superword Level Parallelization (SLP) algorithm from [20], shows that our approach yields a performance improvement of up to 13.78% relative to SLP.

For future work, we plan to port our framework to the PPC back-end in the Jikes RVM compiler, and evaluate our approach on the new VSX extensions in the POWER7 architecture. We would also like to study the impact of various scheduling algorithms, such as trace and modulo scheduling, on vectorization. Early experimental results for loops with conditionals show promise in exploring scheduling and vectorization beyond a single basic block.

## Acknowledgments

We would like to thank members of the Habanero and PACE projects at Rice for valuable discussions related to this work. This research is partially supported by the Center for Domain-Specific Computing (CDSC) funded by the NSF Expedition in Computing Award CCF-0926127. This work is also partially funded by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or Rice University.

## References

[1] Jikes RVM. http://jikesrvm.org/.

[2] GCC compiler. http://gcc.gnu.org/, 2004.

[3] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 207–217, New York, NY, USA, 1975. ACM.

[4] *Intel Advanced Vector Extensions Programming Reference.*, 2008. Available at http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf/.

[5] Aart J. C. Bik. *Applying Multimedia Extensions for Maximum Performance*. Intel Press., 2004.

[6] John Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, 1976.

[7] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[8] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. *SIGPLAN Not.*, 39(6):82–93, 2004.

[9] D. H. Bailey et al. The NAS parallel benchmarks, 1994.

[10] M.G. Burke et. al. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

[11] Free Software Foundation. *Auto-vectorization in GCC*. GCC, 2004.

[12] *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1997. Available at http://developer.intel.com/design/PentiumII/manuals/.

[13] The Java Grande Forum benchmark suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[14] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[15] David Ryan Koes and Seth Copen Goldstein. Near-optimal instruction selection on dags. In *CGO '08*, pages 45–54, New York, NY, USA, 2008.

[16] S. Kral. *FFT Specific Compilation on IBM Blue Gene*. PhD thesis, Vienna University of Technology, June 2006.

[17] S. Kral, F. Franchetti, J. Lorenz, and C. W. Ueberhuber. Fft compiler techniques. In *Proceedings of International Conference on Compiler Construction (CC) 2004*, pages 217–231. LNCS.

[18] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *Int. J. Parallel Program.*, 28(4):347–361, 2000.

[19] Alexei Kudriavtsev and Peter Kogge. Generation of permutations for SIMD processors. In *LCTES '05*, pages 147–156, New York, NY, USA, 2005. ACM.

[20] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 145–156, New York, NY, USA, 2000. ACM Press.

[21] Samuel Larsen, Rodric Rabbah, and Saman Amarasinghe. Exploiting vector parallelism in software pipelined loops. In *MICRO 38*, pages 119–129, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and detecting memory address congruence. In *In International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, 2002.

[23] Bachega et. al. Leonardo. A high-performance SIMD floating point unit for Bluegene/L: Architecture, compilation, and algorithm design. In *PACT '04*, pages 85–96, Washington, DC, USA, 2004. IEEE Computer Society.

[24] Rainer Leupers. Code selection for media processors with SIMD instructions, 2000.

[25] Alex Peleg and Uri Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996.

[26] Ivan Pryanishnikov, Andreas Krall, and Nigel Horspool. Compiler optimizations for processors with SIMD instructions. *Software—Practice and Experience*, 37(1):93–113, 2007.

[27] Vivek Sarkar, Mauricio J. Serrano, and Barbara B. Simons. Register-sensitive selection, duplication, and sequencing of instructions. In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pages 277–288, New York, NY, USA, 2001.

[28] R. Vallée-Rai et al. SOOT - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.