2011 International Conference on Parallel Architectures and Compilation Techniques

An Evaluation of Vectorizing Compilers

Saeed Maleki†, Yaoqing Gao‡, María J. Garzarán†, Tommy Wong‡ and David A. Padua†

†: Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801–2302
Email: {maleki1,garzaran,padua}@illinois.edu

‡: IBM Toronto Laboratory
Markham, Ontario L6G 1C7
Email: {ygao,tomwong}@ca.ibm.com

***Abstract*—Most of today's processors include vector units that have been designed to speedup single threaded programs. Although vector instructions can deliver high performance, writing vector code in assembly language or using intrinsics in high level languages is a time consuming and error-prone task. The alternative is to automate the process of vectorization by using vectorizing compilers.**

This paper evaluates how well compilers vectorize a synthetic benchmark consisting of 151 loops, two application from Petascale Application Collaboration Teams (*PACT*), and eight applications from *Media Bench II*. We evaluated three compilers: GCC (version 4.7.0), ICC (version 12.0) and XLC (version 11.01). Our results show that despite all the work done in vectorization in the last 40 years 45-71% of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers we evaluated.

I. Introduction

The hot spots of many single threaded codes are loops where the same operation is applied across different array elements. Vectorization transforms these loops into instructions that work on multiple data items simultaneously. Typically, vectorization targets either high-end vector processors or microprocessor vector extensions. Vector processors were the main components of the supercomputers of the 1980s and early 1990s. Vector extensions for general purpose microprocessors such as Intel MMX and IBM AltiVec, emerged in the 1990s to support multimedia applications. Vector devices are also found today in video game consoles and graphic cards. Today, vector extensions and vector devices are not only used for multi media applications and video games but also for scientific computations.

The maximum speedup that can be obtained through vector extensions in microprocessors is a function of the width of the vector registers and units. Most of today's machines have vector units that are 128-bit wide, and the vector operations over the equivalent sequential operations are up to 2, 4, 8 and 16 times faster than their scalar counterpart depending on the data type. Thus, vectorization is one of the compiler transformations that can have significant impact on performance.

There are three major ways to make use of vector units: by programming in assembly code, by programming with intrinsic functions in high-level languages such as C, or to use a vectorizing compiler that automatically translates sequences of scalar operations, typically represented in the form of loops, into vector instructions. Programming in assembly language or using intrinsic functions gives the programmer complete control of the low level details at the expense of productivity and portability. On the other hand, vectorizing compilers such as GCC, the Intel C compiler (ICC) and the IBM XLC compiler can automatically generate vector code.

In this paper we study the effectiveness of vectorizing compilers because of its importance for productivity. We first discuss the main obstacles that limit the capabilities of current vectorizers and the available solutions, and then evaluate the vectorization capabilities of GCC, ICC and XLC for different benchmarks, including a set of synthetic benchmarks [5], two applications from the Petascale Application Collaboration Teams *PACT* [3], [14], and the *Media Bench II* [2] applications. Our evaluation methodology was to vectorize the codes using the compilers, and when they failed, do the vectorization by hand. Our results show that after 40 years of studies in auto vectorization, today's compilers can at most vectorize 45-71% of the loops in our synthetic benchmark and only 18-30% in our collection of applications.

As discussed below, we have not found any cases that well-known compiler techniques cannot handle. When vectorization failed, the reason was that the compiler did not implement the necessary techniques, not that automatic vectorization was not possible.

The rest of this paper is organized as follows: Section II describes the benchmarks we used; Section III describes the environment; Section IV discusses the main issues that limit vectorization; Section V presents the evaluation; Section VI presents the related work; and finally Section VII concludes.

II. Benchmark

We used three different sets of benchmarks to evaluate the vectorizing capabilities of the compilers.

1) Test Suite for Vectorizing Compilers: One of the benchmarks for our evaluation is the Test Suite for Vectorizing Compilers (TSVC) developed by Callahan, Dongarra and Levine [5], which contains 135 loops. This benchmark was developed 20 years ago to assess the vectorizing capabilities of compilers. The original benchmark suite was written in Fortran and we translated it into C using `f2c` [9] and manually applied minor transformations to account for the differences between the two languages, such as converting arrays to start at 0 to conform to C convention and aligning all arrays at 16-byte boundaries. We have extended this benchmarks set with 23 additional loops to assess issues not addressed by the old set of loops and removed 7 (obsolete) loops that are formed with

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.68

Machine		double	float	int	short	char
Power 7	Vec	1.18	1.10	1.11	1.16	1.14
	NoVec	1.59	2.91	2.64	5.46	10.49
	Speedup	1.35	2.64	2.37	4.71	9.20
Intel i7	Vec	1.67	1.61	1.67	1.65	1.68
	NoVec	2.37	4.81	4.83	9.61	19.22
	Speedup	1.42	2.99	2.89	5.82	11.44

TABLE II
RUNNING TIMES AND SPEEDUPS FOR THE OPERATION `A[I]=B[I]+1` WITH FIXED ARRAY SIZE 12.5KB.

`gotos` and conditions. The extended version (which contains 151 loops) is publicly available [1].

2) Codes from Petascale Application Collaboration Teams: The goal of the Petascale Application Collaboration Teams (PACT) [4] is to develop codes that meet the performance milestones set for several Blue Waters applications. For this study we used MIMD Lattice Computation (MILC version 7.6.2) [3] and Turbulence problem with Direct Numerical Simulation (DNS) [14] from PACT. Notice that DNS uses FFT which is going to be replaced by a hand tuned, built-in FFT from IBM. Therefore, we did not consider the code segment implementing FFT when we evaluated the compilers.

3) Media Bench II: Media Bench II [2] is a benchmark representing multimedia applications. This benchmark contains encoder/decoder of video/image files. Eight applications from this benchmark were chosen: the encoders and decoders for JPEG, H263, MPEG2, and MPEG4. We did not use the other applications from this benchmark because there is not much room for improvement through vectorization in them.

III. ENVIRONMENTAL SETUP

A. Target Platforms

We used the IBM Power 7 and Intel Nehalem i7 processors for our experiments. Table I describes the platforms used. An estimation of the speedup that we can expect from vectorization for the two machines used in our study is presented in Table II. For the estimation, we compared the execution times of the vectorized version and its *scalar* counterpart (i.e. non-vectorized code) of the simple computation `A[i]=B[i]+1` over a large array. We kept the array size fixed (12.5KB) and changed the data types of the elements. Since we measured the running time of $400,000$ consecutive executions of this vector operation and the arrays fully fit in the L1 and L2 caches, the load latencies can be ignored.

B. Compilers

The compilers used in this experiment were GCC and ICC on the Intel platform and XLC on the IBM platform. The version of each compiler and the command line options are shown in Table III. GCC version 4.7.0 was new at the time of writing of this paper and it still had some bugs. Because of this, we were unable to compile DNS benchmark with it and used GCC 4.4.5 instead.

IV. ISSUES WITH VECTORIZATION AND THEIR SOLUTIONS

To asses how well current compilers vectorize we have taken the benchmarks listed in Section II and vectorized with the compilers described in Section III-B. When the compiler failed, we applied hand vectorization. Our results, reported in Section V, show that the the hand-vectorized codes run significantly faster than the compiler-vectorized codes. After carefully studying the loops that we were able to efficiently vectorize by hand, we have found the main reasons why the compilers fail to generate efficient vector code: i) the lack of accurate analysis, ii) the failure to perform certain compiler transformations to enable vectorization, and iii) the lack of accurate profitability analysis. In this Section, we discuss these issues in detail.

A. Accurate compiler analysis

To generate efficient vector code, the compilers need an accurate interprocedural analysis of the interactions between operations. This analysis in turn enables transformations. In our study, we observed that compilers do not apply three important transformations, discussed in Section IV-B: memory layout change, code replacement, and data alignment. The inability of compilers to do accurate interprocedural pointer disambiguation and interprocedural array dependence analysis is at least in part the reason for their inability to perform these three transformations.

B. Transformations

Compilers need to apply certain transformations to make vectorization possible and/or profitable. In our study, we found that there are some scalar-to-scalar transformations which were not applied by the compiler; however, when the transformation was manually applied to some of the codes not vectorized by the compiler, the compiler was able to automatically vectorize.

In the five subsections below (B1-B5), we discuss the transformations that we applied by hand and outline when they are needed in the context of vectorization. We classify these transformations based on the issue they are trying to address. In Section V we will discuss what compiler transformations were applied for each of the benchmarks we used.

B1) Non-Unit stride accesses Vector loads and stores of current processors can only access elements stored in consecutive locations. Thus, although it is possible to issue several scalar loads/store operations to load/store a vector register and then execute the computations in vector mode, the resulting vector code obtains low speedups, or even slowdowns, with respect to its scalar counterpart. Several transformations can be applied by the compiler to efficiently vectorize loops with non-unit stride accesses.

1) *Memory Layout change.* The compiler can change the memory layout of the data structures. As an example consider the first loop in Figure 1, where the accesses `pt[i].x` are non-unit strides, since variable `pt` has been declared as an array of the `point` structure. To enable efficient vectorization the compiler could transform the memory layout of the variable `pt`, as shown in the second code in Figure 1, where

Platform	Model	Frequency (MHz)	Vector Unit	Vector Length	L3 Cache (MB)	Operating System
IBM	Power7 9179-MHB	3864.00	AltiVec	128 (bit)	32 (4 local)	Linux v2.6
Intel	Intel Core i7 920	2659.964	SSE4.2	128 (bit)	8	Linux v2.6

TABLE I
SPECIFICATION OF EXPERIMENTAL MACHINES

Specification	GCC	ICC	XLC
Version	4.7.0	12.0	11.1
Baseline Optimization	`-O3 -fivopts -funsafe-math-optimizations`	`-O3`	`-O3 -qhot -qarch=pwr7 -qtune=pwr7 -qipa=malloc16`
Vectorization Options	`-flax-vector-conversions -msse4.2`	`-xSSE4.2`	`-qenablevmx -qdebug=NSIMDCOST -qdebug=alwaysspec`
Disable Vectorization	`-fno-tree-vectorize`	`-no-vec`	`-qnoenablevmx`
Vectorization Report	`-ftree-vectorizer-verbose=[n]`	`-vec_report[n]`	`-qreport`

TABLE III
COMPILERS SPECIFICATIONS

```
1 typedef struct{int x,y,z} point;
2 point pt[LEN];
3 for (int i=0; i<LEN; i++)
4   pt[i].x *= scale;
```

```
1 int ptx[LEN], int pty[LEN], int ptz[LEN];
2 for (int i=0; i<LEN; i++)
3   ptx[i] *= scale;
```

Fig. 1. An array of a struct versus an array for each field

```
1 for (ctr = DCTSIZE-1; ctr >= 0; ctr--) {
2   tmp0 = dataptr[0] + dataptr[7];
3   tmp7 = dataptr[0] - dataptr[7];
4   // Repeats with tmp1...tmp6
5   dataptr[0] = f0(tmp0,...,tmp7);
6   dataptr[4] = f1(tmp0,...,tmp7);
7   // Repeats with dataptr[1..3] and dataptr[5..7]
8   dataptr += 8;
9 }
```

Fig. 2. Function `jpeg_idct_islow` form `CJPEG`. All the computations in this loop are vectorizable, but memory accesses require non-unit strides.

there is a separate array for each field of the `point` structure. Although both codes can be vectorized, the vectorization of the first code is inefficient, while vectorizing the second code obtains an speedup of 3.7 versus the sequential code in the Intel platform. ICC and GCC do not apply this transformation in part because they lack interprocedural strategies to propagate the new layout across different procedures in the application. XLC can apply this transformation, but it is very conservative, and as a result, it rarely applies it. Furthermore, we did not apply this transformation to any benchmark by hand since it would need global changes to a program, although applying it to some cases in our benchmark would improve the performance of the vector code.

2) *Data Copying (DC).* When interprocedural analysis is not available and the compiler finds vectorizable loops that operate on data with non-unit stride, the compiler can copy the data to local buffers where data are consecutive in memory, and then copy the results back, after the loop has been vectorized. Of course, the compiler has to determine if the overhead of this transformation is amortized by the benefit of the vectorization. For that the compiler needs a cost model as will be discussed in Section IV-C.

An example where data copying can be used, is shown in Figure 2 which accesses the vector `dataptr` with stride 8. `f0,...,f7` are costly macro functions; therefore, there are many vectorizable computations if the data is reorganized. To obtain unit stride accesses we have manually transposed the $8 \times$ `DCTSIZE` matrix of data referenced by `dataptr` before and after the loop so that we can use the transposed data inside the loop without affecting the rest of the program. Our implementation of transposition takes advantage of the microprocessor vector extensions by efficiently using vector permutation instructions. Then, the code is automatically vectorized by the compiler. This transformation could be easily implemented by a compiler, since it is a local transformation. However, the compiler will need the appropriate cost model to determine when the overheads of copying the data in and out can be amortized by the speedups obtained.

Another example where data copying is useful, is shown in Figure 3-(a), where the index to access the array `iclp` is not a linear function of the loop variable and `f0,...,f7` are costly macro functions. A possible solution to this problem is to store the values which are used as indices for array `iclp`, into a temporary array to make the main loop auto vectorizable. This approach requires another loop after the main loop which goes through the temporary array and stores the correct result back into the output array, as shown in Figure 3-(b). As in the previous example this solution has the data copying overhead and so the cost model needs to determine the profitability of this transformation.

3) *Recomputation (RC).* We also found loops with vectorizable computations that contain non-unit stride accesses to a lookup table, where the value of the index to access the table is statically unknown. One of the reasons that the programmers use these lookup tables is to save the values

```
1 for (i=0;i<8;i++){
2   x1 = block[8*1+i]; x7 = block[8*7+i];
3   // Repeats with x0,x2...,x6
4   block[8*0+i] = iclp[f0(x1,...,x7)];
5   // Repeats with block[8*1+i..8*7+i]
6 }
```

(a) The indirect access to array `iclp`.

```
1 for (i=0;i<8;i++){
2   x1 = block[8*1+i]; x7 = block[8*7+i];
3   // Repeats with x0,x2...,x6
4   temp[8*0+i] = f0(x1,...,x7);
5   // Repeats with temp[8*1+i..8*7+i]
6 }
7 for (i=0;i<8;i++){
8   block[8*0+i] = iclp[temp[8*0+i]];
9   // Repeats with block[8*1+i..8*7+i]
10 }
```

(b) Data copying is applied.

```
1 void Initialize_Fast_IDCT()
2 {
3   int i;
4   iclp = iclip+512;
5   for (i= -512; i<512; i++)
6     iclp[i]=(i<-256)?-256:((i>255)?255:i);
7 }
```

(c) The initialization for array `iclp`.

Fig. 3. Function `idctcol` from MPEG2. Non-unit stride accesses to a lookup table.

of expensive mathematical functions, such as `sqrt` or `sin`. However, in some cases it is actually faster to recompute the functions than to use the lookup table. With the appropriate interprocedural analysis the compiler could replace the lookup table with the actual computation used to initially fill the table. Lin and Padua studied how to calculate the reaching definition of subscript arrays and apply forward substitution [15].

As an example consider again the loop in Figure 3-(a). We found that array `iclp` is a read-only array and it is only initialized once, as shown in Figure 3-(c). Therefore, a different possible solution to the one previously described for the loop in Figure 3-(a) is to replace `iclp` with the actual computation in terms of `x7` and `x1` and let the compiler vectorize it.

4) *Loop Interchange.* Loop interchange can be applied when traversing a matrix along the dimensions with non-unit strides so that the access moves along consecutive locations. Compilers need to apply data dependence analysis to determine if the transformation is valid.

B2) Data Alignment. Accesses to vectors whose addresses are aligned on certain boundaries (which is 16-byte for our two experimental machines) can in some cases be significantly faster than the unaligned ones. To perform a vector load/store of a non-aligned memory address, previous generations of Intel processors such as Core 2 have an unaligned vector load/store instruction that incurs a significantly higher latency than the aligned one. However, the Intel Nehalem processor supports an unaligned load/store instruction that executes almost as fast as the aligned load/store. IBM does not have an unaligned

```
1 for (int i = 1; i < LEN-1; i++){
2   a[i] = b[i-1] + c[i];
3   b[i] = d[i  ] + e[i];
4 }
```

Fig. 4. Acyclic backward data dependence.

vector load, but supports it with higher latency using two loads of consecutive 128-bit of data and shift operations. To address this issue, compilers need to apply the following transformations:

1) *Padding.* Data allocations need to be aligned. Compilers can make sure that the memory allocation of an array is aligned by padding i.e. allocating a larger chunk of memory than needed and using this extra space to guarantee data alignment. This type of transformation is not always applied automatically by the compiler and so programmers need to manually apply it. The compilers could easily apply this transformation, although care needs to be taken when a program allocates many arrays of small sizes.

2) *Loop Peeling.* Peeling off iterations from the beginning and/or the end of the loop is sometimes needed to generate vector code with aligned vector loads/stores.

Notice that the compiler needs to know the alignment of the pointer (which could be an input parameter to the function containing the loop) so it can determine which vector load/store instructions to use or how many iterations to peel off. For this purpose, the compiler needs an accurate interprocedural analysis. Eichenberger et al. presented a compilation scheme that systematically vectorizes loops in the presence of misaligned memory references [8].

B3) Data Dependence based Transformations. The correctness of certain compiler transformations is determined by data dependences. Below we outline some of these transformations.

1) *Statement reordering.* Compilers need to apply this transformation to enable vectorization of codes with lexicographically backward dependences within a loop body. As an example, consider the loop in Figure 4 which has an acyclic backward dependence. This loop is vectorized by ICC and XLC, but not by GCC. XLC vectorizes it perfectly by reordering the statements. The new ICC compiler (v12.0) can vectorize it, while the previous one (v11.1) could not. However, ICC v12.0 only vectorizes it partially because it distributes the loop over the statements and reorders the loops and vectorizes them as opposed to reordering the statements, strip-mining, and vectorizing. This introduces overhead and less locality for array `b`. GCC vectorizes neither the original loop nor the reordered one.

2) *Algorithm and Reduction Substitution.* Algorithm and reduction substitution can be used to vectorize computations that form a cycle of flow dependences in the dependence graph. Although algorithm substitution encompasses reduction substitution, we deal with reductions separately since these seem to be by far the most frequent kernels that compilers must replace for vectorization.

2.1) *Algorithm Substitution.* An example where substitution

```
1 for (int i = 1; i < LEN-1; i++)
2   A[i] = A[i-1] + B[i];
```

Fig. 5. Prefix sum

of an algorithm that is not a simple reduction, is necessary for vectorization, is shown in Figure 5. For substitutions of all classes of algorithms, a typical strategy is for the compiler to isolate the cycle and then pattern match the computation in the cycle to one of the frequently occurring kernels. In numerical computations these kernels include simple computations such as the loop shown in Figure 5 which is also known as *Prefix Sum*, and can include several classes of reductions and even complex kernels such as matrix-matrix multiplication. Prefix sum is a kernel computation that occurs with some frequency and its vector implementation has been extensively studied. For example, Chen and Kuck studied algorithms to parallelize and vectorize these types of computations [12], [6]. However, none of the compilers we evaluated vectorize this loop. We implemented a manually vectorized prefix sum for the `float` data type and an array of size $32,000$ that runs 4.61 and 1.46 faster on our experimental IBM and Intel platforms, respectively, than the code generated by the compiler for the loop in Figure 5.

2.2) *Reduction Substitution (RE).* Compilers are capable of recognizing reductions and substituting them with vector versions. Common examples of reductions are the `sum` or `max` of an array. The compilers recognize many of them, but not all.

In the cases where reduction is not recognized automatically by the compilers, we found that if we first apply scalar expansion, then the compiler can automatically vectorize them. For example, the loop in Figure 6-(a) is vectorized by ICC, but not by XLC or GCC. By manually applying scalar expansion, as shown in Figure 6-(b), the compilers can vectorize the inner loop, as there are no dependences. Notice that in this example there is some extra code before and after the loop, but it is not shown to keep the example short.

3) *Node Splitting.* This transformation can break cycles in the dependence graph where not all of the dependences are flow-dependences. This transformation is similar to data copying except it breaks dependences. In Section V, we grouped node splitting and algorithm substitution together since both resolve the issue with cyclic dependences.

4) *Loop Peeling (LP).* Besides its use for alignment, loop peeling can also be used to remove dependences. Thus, peeling off first and last few iterations is helpful to break dependences which only exists for those iterations.

5) *Symbolic Resolution (SR).* The compilers does not always generate multiple versions of a loop for the cases where a data dependence is assumed due to a symbolic variable such as `for(i=...)` {`A[i]=A[i+k]+...;`}. In this example, the compiler can vectorize the loop in a simple manner for positive values of `k` but for negative values either more complicated transformations are needed or the loop cannot be

```
1 int index = 0;
2 float min = A[0];
3 for (int i = 0; i < LEN; i++)
4   if (A[i] < min){
5     min = A[i];
6     index = i;
7   }
```

(a) Finding maximum element with its index in an array (MaxLoc).

```
 1 int index[64];
 2 float min[64];
 3 ...
 4 for (int i=0; i<LEN; i+=64)
 5   for (int ii=i,j=0; ii < i+64; ii++,j++)
 6     if (A[ii] < min[j]){
 7       min[j] = A[ii];
 8       index[j] = i;
 9     }
10 ..
```

(b) Scalar expanded MaxLoc.

Fig. 6. MaxLoc Reduction

speeded up with vectorization. Therefore, they need to add a runtime check to run a vectorized code for positive values of `k` and a scalar code for negative values.

6) *Recognition of Induction Pointers (IP).* An induction pointer, is a pointer that is increased by a constant value in each iteration. As an example, assume that pointer `A` is an induction pointer whose dereferenced value `*A` is used inside a loop and it is incremented by 1 in each iteration `A++`. Instead, `A[i]` can be used, where `i` is the loop induction variable. Compilers do not always recognize induction pointers as array accesses and programmers need to change them manually into their array counterpart.

7) *Wrap Around Variable Detection.* The compilers need to express wrap around variables in terms of the loop induction variable to enable vectorization. This transformation may need peeling off a few iterations.

8) *Loop Distribution.* Distributing a loop over the statements inside it can enable vectorization of some of the statements in the loop body. It is useful for those cases where one (or more) of the statements are vectorizable, but the other (or others) are not. This transformation may result in limited performance benefit (or even slowdowns) if the loop distribution decreases cache and/or register locality. Profitability analysis, which is discussed later, should determine when this transformation should be applied.

9) *Loop Interchange.* This transformation can enable vectorization when dependences are carried across the iterations of the inner loop, but not of the outer one. Since loop interchanging may change the access pattern, care must be taken to not introduce non-unit stride access.

10) *If-Conversion.* If-conversion is a technique used to replace control dependences by data dependences. The compiler converts programs with conditional branches to code containing predicated instructions. Instructions previously guarded by a branch are guarded by a predicate. Therefore, the statements inside a branch are now executed regardless of the outcome

```
1 for (int i = 0; i < LEN; i++)
2   if (A[i] < B[i]){
3     C[i] = D[i]+E[i];
4   }
```

(a) A loop containing conditional statements

```
1 for (int i = 0; i < LEN; i++)
2   C[i] = (A[i]<B[i]) ? (D[i]+E[i]) : C[i];
```

(b) Converted version

Fig. 7. If-Conversion example

```
1 for (int i=0; i<N; i++)
2   if (C[i] == 0)
3     A[i] = FAST_FUNCTION(B[i]);
4   else
5     A[i]=C[i]*SLOW_FUNCTION(D[i])
6         +FAST_FUNCTION(B[i]);
```

(a) Short circuiting to prevent from computing the slow function.

```
1 for (int i=0; i<N; i++)
2     A[i] = FAST_FUNCTION(B[i]);
3 #pragma novector
4 for (int i=0; i<N; i++)
5   if (C[i] != 0)
6     A[i]=C[i]*SLOW_FUNCTION(D[i])+A[i];
```

(b) Removing the short circuit to generate an efficient vector code.

Fig. 8. A loop containing conditional statement

of the branch and the predicate is used to store the correct result back into memory by using bitwise mask operations such as logical and/or/not operations. Figure 7-(a) shows a loop containing conditional statements and Figure 7-(b) shows its converted version. Clearly this transformation may execute unnecessary instructions depending on values of the input, but it makes the loop vectorizable. Therefore, compilers need to use profiling information to estimate its profitability.

An example of a loop containing branches in our benchmark set is shown in Figure 8-(a). The branch to prevent the computation of the slow function in order to improve the scalar performance is added because it is known that for many iterations `C[i]` is zero. To vectorize this loop there are three possible options: i) execute both parts of the branch in vector fashion and only store the appropriate one; ii) remove the condition and the first statement and vectorize only the second statement. This option does not take advantage of the fact that `C[i]` can be 0 for many iterations but it is better than the first one; iii) apply loop distribution across the statements, vectorize the first loop and run the second loop in scalar mode, as shown in Figure 8-(b). As it will be discussed later in Section V, some functions in Media Bench II applications have this type of computation. We refer to this type of loops by *Fast-Slow Function (FSF)*.

To determine the best way to vectorize a loop with if conditions the compiler needs profile information, as the decision on which is the best option, is input data dependent.

B4) Generation of efficient vector code. Some transformations are required to increase the amount of parallelism or to minimize the overheads of the vectorization.

1) *Data Permutations (DP).* This transformations is necessary when a vector operation on two vector registers is such that the corresponding elements of the operation are not aligned. In that situation, permutation instructions need to be used. It is important to minimize the number of permutations instructions, as they are expensive, and not doing so, can result in the vector code running slower than the original sequential code. Although several works have been published on this topic such as [10], [17], [18], current compilers do not apply this optimization efficiently.

As an example, consider the code in Figure 9-(a) from MILC. In this application the data types of the arrays are `double` and the elements of the matrices are complex numbers stored in an array of pairs representing complex numbers. The compiler can easily vectorize the computation of `a0r*b0r,a0r*b0i` and `a0i*b0i,-a0i*b0r` which are the first and the second terms, respectively, of the two statements on line 8 and 11, but need permutation operations to add the results of the vector multiplication, as the data do not have the right alignment in the vector registers. To minimize the number of permutations, the compiler needs to carefully regroup the additions of the results that have the same alignment (this makes permutations unnecessary), as shown in the computations of variables `cpr,cpi` and `cmi,cmr` in Figure 9-(b). In this computation, a single permutation is performed to compute the sum in lines 7 and 8. In addition, we avoid individual multiplications by -1, by factoring out the minus in the computation of `cmi`, as shown in line 5 of Figure 9-(b). Another solution to this problem is to change the layout of the complex structure as discussed earlier. In other words, if the real and imaginary parts of the complex numbers of the matrices had been stored in different matrices, vectorizing this code would have been straight forward. But as it was discussed before, changing the layout of a structure requires global changes. Permutation minimization was required in several loops in DNS and MILC.

2) *Data Type Promotion (DTP).* On integer codes it is not uncommon for programmers to cast operations in a loop on a data type to a larger data type, to avoid overflows. However, sometimes the programmer cast to a larger type than necessary, which results in less parallelism for vectorization. Since casting to different data types requires shuffling of vector registers, we can also classify it as data permutation.

As an example consider the code in Figure 10. In this particular example, since $16 \times (2 \times \texttt{MAX\_CHAR}) < \texttt{MAX\_SHORT}$, it is possible to perform the computation in the loop temporarily as 16-bit short, instead of using the 32-bit integer data type specified by the programmer and then add the result back to `sum`. As we will see later in Section V, there are many occurrences of this type of computation in the Media Bench II applications.

3) *Loop Invariant Recognition (LIR).* We have observed that in some cases compilers fail to recognize a loop invariant that can be moved outside of the loop. In particular, we have observed that XLC and GCC failed to recognize that a constant

```
 1 for (int i=0; i<3; i++){
 2   a0r=a->e[0][i].real;
 3   a0i=a->e[0][i].imag;
 4   b0r=(*b)->h[0].c[0].real;
 5   b0i=(*b)->h[0].c[0].imag;
 6   ...
 7   c->h[0].c[i].real=
 8      a0r*b0r+a0i*b0i+a1r*b1r
 9     +a1i*b1i+a2r*b2r+a2i*b2i;
10   c->h[0].c[i].imag=
11      a0r*b0i-a0i*b0r+a1r*b1i
12     -a1i*b1r+a2r*b2i-a2i*b2r;
13 }
```

(a) Original code without regrouping.

```
1 for (int i=0; i<3; i++){
2   cpr=a0r*b0r+a1r*b1r+a2r*b2r;
3   cpi=a0r*b0i+a1r*b1i+a2r*b2i;

5   cmi=-(a0i*b0r+a1i*b1r+a2i*b2r);
6   cmr= (a0i*b0i+a1i*b1i+a2i*b2i);
7   c->h[0].c[i].real=cpr+cmr;
8   c->h[0].c[i].imag=cpi+cmi;
9 }
```

(b) Same code with regrouping.

Fig. 9. A loop from a function in MILC that requires optimizing data permutations

```
1 //int sum;
2 //unsigned char* pix1,pix2;
3 for (int i=0; i<16; i++)
4   sum += abs(pix1[i] - pix2[i]);
```

Fig. 10. A common type of computation in Media Bench II

vector that was obtained from replicating a constant could be moved outside the loop. We manually moved them out for these cases.

B5) Other transformations

1) *Reroll (RR).* Rerolling manually unrolled loops in some cases enables vectorization. Thus, unless the compiler can vectorize basic blocks, fully unrolled loops cannot be vectorized and partially unrolled loops could lead to inefficient vectorization. Of course, vectorization of basic blocks is in some way a superset of loop rerolling.

2) *Vectorization of while loops.* `While` loops where the condition of the loop is not predictable at compile/run time, such as `while(a[i]<b[i])`, or `for` loops that have a `break` instruction, are not vectorized by the compilers we evaluated. We vectorized these loops manually by checking the conditional expression of `while(a[i]<b[i])` for consecutive iterations in a vector manner and doing an `AND` reduction. If the outcome of the reduction was true, a vector code for the body of the loop runs; otherwise, a scalar code runs.

C. Profitability

Compilers use cost models to determine whether a transformation is profitable, i.e., whether the transformed code runs faster than the original. To determine if vectorization is profitable, the compiler might need information about the input data, the number of iterations of the loop, or whether the overheads associated with the extra code added to enable vectorization can be amortized with the benefit of the vectorization.

In some cases profiling information can be important to guide the compiler. For example, profiling information can be used to determine the most likely outcome of a branch. In fact, loops with if conditions can be vectorized using the if-conversion method discussed before. However, vectorizing loops with if statements may result in slowdowns. Thus, the profitability of vectorization of loops with if statements is usually input data dependent, and only profile information can be used to determine whether they should be vectorized.

V. Evaluation of Vectorizing Compilers

A. Methodology

In this section, we evaluate the vectorization capabilities of XLC, ICC and GCC. All arrays in the codes used in this study are 16-byte aligned and contain the restrict attribute and alignment assertions. One intention was to provide the compiler with as much information as possible. The restrict attribute for a pointer such as `float* __restrict__ A` tells the compiler that no other pointer will access the same memory addresses that `A` accesses. Alignment assertions are instructions that inform the compiler about the alignment of a pointer. We believe that in many cases the restrict attribute and the alignment assertion could be automatically inserted by the compiler but we have not studied this issue.

To obtain our experimental results we first compiled the codes with vectorization disabled to generate scalar versions (baseline) and with vectorization enabled to generate vector codes (see compiler flags in Table III). Scalar and vector codes were timed. When the obtained speedups were lower than the expectation according to Table II, we transformed it manually. First, we transformed the code at the source level and then recompiled with vectorization options; if that also failed, we wrote vector code using vector intrinsics. We will use the following terminology for the evaluation:

- **Speedup:** Speedup of a code is the ratio of the running time of the baseline, which is the original code compiled with vectorization disabled, over the running time of the code (untransformed or transformed) compiled with the vectorization options in Table III.
- **Auto Vectorized:** A code is auto vectorized when it is automatically vectorized by the compiler and the speedup is at least 1.15.
- **Perfectly Auto Vectorized:** A code is perfectly auto vectorized when it is auto vectorized by the compiler and we could not obtain a higher speedup by hand.
- **Not Vectorizable:** A code is not vectorizable when neither the compiler nor the manual transformations are helpful in obtaining a speedup of greater than 1.15 due to vectorization. In our evaluation, the classification of a loop as not vectorizable is done based on the target platforms (Intel or IBM), and not on the compilers.

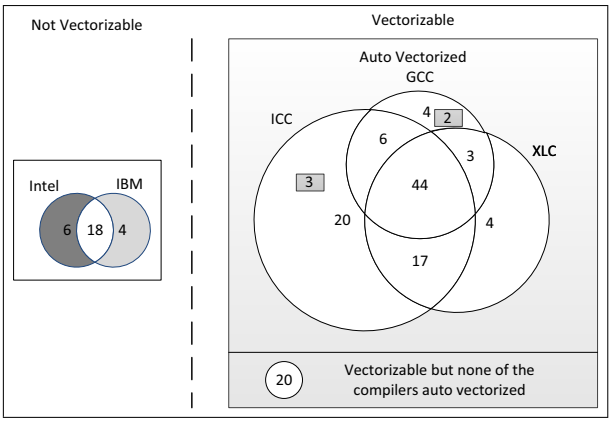

Fig. 11. Compilers evaluation for TSVC.

Notice that none of the evaluated compilers used profiling to improve their capabilities in vectorization, except in one case where XLC used the profiling data to apply symbolic resolution transformation. The use of profiling changed the performance of both the scalar and vectorized code (sometimes increased and sometimes decreased). As a result, we have not used the profile information for our evaluation. Next, we present the evaluation for each of the benchmarks described in Section II.

B. Test Suite for Vectorizing Compilers

Figure 11 shows a Venn diagram that graphically illustrates the loops that are auto vectorized by each compiler and the loops that are not vectorizable on the Intel or the IBM platform. The diagram also shows graphically the number of loops that are vectorized by all the compilers, the ones vectorized by only one compiler, or the ones vectorized by two of the compilers, but not the other one. The numbers show that ICC is the compiler that auto vectorizes a higher number of loops, 90, versus the 68 of XLC or the 59 of GCC. The numbers 3 and 2 in the shaded rectangles in the auto vectorized sets correspond to the loops that we classified as not vectorizable for the IBM platform, but they are auto vectorized on the Intel platform. There are 18 loops that cannot be vectorized for any of the platforms because most of them require sophisticated gather or scatter hardware support to efficiently vectorize them. The number 20 which is not in any of these sets, shows the loops that were not vectorized by any of the compilers but are vectorizable by hand on both platforms.

Table IV shows how much programmer effort is required to vectorize the extended TSVC loop collection [1] described in Section II, based on the compiler (XLC or ICC) used. In particular the table shows how many loops were perfectly auto vectorized by the XLC or ICC compiler, by applying transformations at the source level or by writing vector code using intrinsics. The transformations that we used are the ones discussed in Section IV. We did not consider GCC for this study, as our results show that GCC fails to vectorize many more code snippets than ICC. Notice that the diagram in Figure 11 shows the *auto vectorized* loops, while Table IV shows the loops that were *perfectly auto vectorized*.

Method	XLC	ICC
Vectorizable	124(82.12%)	127(84.11%)
Perfectly Auto Vectorized	66(43.71%)	82(54.31%)
Source Level Transformation	42(27.82%)	38(25.17%)
Intrinsics	16(10.6%)	7(4.64%)

TABLE IV
LOOP CLASSIFICATION BASED ON THE METHOD USED TO ACHIEVE THE BEST SPEEDUP.

Table VI shows the transformations we had to apply to perfectly auto vectorize the loops in TSVC. Table VII shows the number of loops that were perfectly auto vectorized by one compiler, but not by the other one, and the transformation that was required. For instance, the column XLC-ICC indicates the loops vectorized by XLC, but not by ICC. This column shows that XLC vectorized 3 loops that were not vectorized by the ICC compiler where statement reordering was required. When comparing ICC versus XLC we can observe that ICC vectorized 9 loops that were not vectorized by XLC that required if-conversion. The reason is that in most cases, XLC only vectorizes loops with conditional statements when the programmer has applied the if-conversion transformation manually. Finally, the other 4 loops that XLC did not vectorize but ICC did, are loops with non-unit stride accesses that we vectorized using the data copying transformation. ICC vectorized these loops because Intel platforms can use scalar loads to build a vector register in case of non-unit stride accesses. Table VIII shows the average speedup obtained based on the vectorization method used over all loops in this benchmark (speedup 1.0 is considered for the loops which were not vectorized).

We also compared the compilers used in this study with the compilers used in the evaluation made in 1991 by Callahan, Dongarra, and Levine TSVC [7]. Since the machines used in their study had support for vector instructions not available today in the microprocessor vector extensions, such as gather/scatter, and the collection we used had more loops than the original collection, we removed the loops which did not exist in the original collection and the loops which were not vectorizable on neither Intel platform nor IBM platform. We ended up with 85 loops and Table V how many loops were vectorized by each compiler. In this table, a loop is considered vectorized by a today's compiler if a speedup of at least 1.15 is obtained (auto vectorized by our definition) and considered vectorized by a compiler from Callahan's collection if the loop was either vectorized or partially vectorized according to their model described in section 7 of [7]. Also, we refer to XLC, ICC and GCC as New Compilers and the compilers from Callahan's collection as Old Compilers. As it is clear, only ICC is comparable with the old compilers and that XLC and GCC vectorized significantly fewer loops.

Overall, compilers have much room for improvement in vec-

Compilers		Vectorized
New	XLC	45 (52.94%)
	ICC	61 (71.77%)
	GCC	38 (44.71%)
Old	fc 6.1	65 (76.47%)
	cft77 4.0.1.1	53 (62.35%)
	CF77 4.0	70 (82.35%)
	FORTRAN V5.5	54 (62.53%)
	f77 4.3	67 (78.82%)
	Fortran77EX	60 (70.59%)
	fort77	62 (72.94%)
	VS FORTRAN 2.4.0	66 (77.65%)
	f77sx 010	63 (74.12%)

TABLE V
COMPARISON BETWEEN NEW AND OLD COMPILERS

Transformation Required	XLC	ICC	GCC
Total	58	45	70
if-conversion	10	0	14
Vectorization of `While` and `GOTO` loops	3	0	0
Algorithm substitution and Node Splitting	9	8	6
Statement Reordering	0	3	3
Data Copying	9	8	12
Wrap Around Variable Detection	5	2	5
Reduction	4	5	10
Rerolling	0	4	1
Symbolic Resolution	3	3	3
Loop Interchanging	3	2	5
Loop Distribution	1	1	2
Other	11	9	9

TABLE VI
TRANSFORMATIONS REQUIRED TO PERFECTLY AUTO VECTORIZE THE LOOPS IN TSVC.

torizatoin with regard to our results from TSVC benchmark. There are three main reasons why today's compilers fail in vectorizing the loops in the TSVC benchmark. The first reason is the hardware limitations of the current vector extensions. For example, non-unit stride accesses are more challenging for today's compilers than they were for the compilers of 20 years ago, as the machines back then had hardware support that is not available in today's microprocessors. The second reason is that today's compilers are not designed to support some programming patterns such as loops with wrap around variables. The third reason is that compilers fail in the presence of data dependences graphs that require statement reordering or algorithm substitution, but we do not really know how common they are today. We believe that this is because the compilers are now concerned with a different set of patterns.

Our final observation is that if there was an imaginary compiler which could vectorize all the cases that XLC, ICC and GCC auto vectorized, it would auto vectorize about 83.05% of the loops which can take advantage of the vector instructions of both platforms. In contrast XLC, ICC and GCC can separately auto vectorize 54.84%, 70.87% and 46.46% of the vectorizable loops in each platform. This number proves that compiler technology to vectorize most of the loops already exists, but the different compilers are developed by following different directions.

	XLC-ICC	ICC-XLC	ICC-GCC	GCC-ICC	XLC-GCC	GCC-XLC
Total	9	25	36	13	19	12
Statement Reordering	3	0	1	2	2	0
Loop Interchange	0	1	3	0	2	0
Rerolling	3	0	0	2	1	0
Reduction	1	2	6	1	4	0
if-conversion	0	9	14	0	9	3
Wrap Around Variable Detection	0	3	3	0	0	0
Data Copying	0	4	4	3	0	3
Other	2	6	5	5	1	6

TABLE VII
COMPILER COMPARISON BASED ON THE TRANSFORMATIONS REQUIRED TO PERFECTLY AUTO VECTORIZE THE TSVC LOOPS.

Method	XLC	ICC	GCC
Auto Vectorization	1.66	1.84	1.58
Transformations	2.97	2.38	
Intrinsics	3.15	2.45	

TABLE VIII
THE AVERAGE SPEEDUPS OBTAINED BY EACH METHOD.

C. PACT and Media Bench II Applications

Table X shows results for the functions in the PACT and Media Bench II benchmark that consume most of the execution time of the applications. We found these functions by using gprof [11]. For each application, the table shows the name of the function (the number that appears in front of some of them represents the loop number in that function, since some of them have multiple loops); the percentage of time the application spent in each function (Perc); the speedups for each compiler when the loops where automatically vectorized (Auto columns) and the speedup when the code was manually transformed or manually vectorized with intrinsics (Manual columns). The numbers in bold in the Auto columns indicate that the compiler perfectly auto vectorized this code. Notice that Table X shows all speedups not just those above 1.15. Finally, the column Transformation indicates the transformation (described in Section VI) that we had to apply to perfectly auto vectorize a loop.

The DNS and MILC part of the table X shows the importance of the optimization of the data permutations. As it is clear from the table, XLC only obtains 1.02x speedup in DNS application, while 6.98x can be obtained with manual transformations; ICC does not obtain any speedup for MILC, while manual transformations obtained 1.23x. GCC obtains speedup of 1.25 for DNS which improves to 1.96 after manual transformations. GCC does not obtain any speedup for MILC before the manual transformations. Therefore, compilers need to improve their capabilities in vectorizing cases where data permutations are necessary. The imaginary compiler that we mentioned above, can auto vectorize (with threshold 1.15) 10 out of 13 loops (76.92%). XLC auto vectorized 6 out of 13 loops (46%), as well as ICC.

Overall, the compilers vectorized few loops in the appli-

Method	XLC	ICC	GCC
Auto Vectorization	1.154	1.279	1.232
Manual	2.101	2.743	2.692

TABLE IX
THE AVERAGE SPEEDUP OBTAINED WITH AUTO VECTORIZATION AND MANUAL VECTORIZATION FOR PACT AND MEDIA BENCH II.

cations in PACT and Media Bench II. XLC, ICC and GCC only auto vectorized 6 (18.18%), 10 (30.30%) and 7 (21.21%) loops, respectively, out of total 33 loops. However, the imaginary compiler could vectorize 16 (48.48%) loops. Notice that the data in Table X shows that the main transformations compilers need to apply to avoid data dependence issues in PACT and Media Bench II are recognition of induction pointers, reductions, symbolic resolution, and loop peeling which are simple to apply since they need only local information. However, the other transformations are the ones that deal with non-unit stride accesses including recomputation and data copying which requires interprocedural data and profitability information. Also, the number of loops that require shuffling transformations including data permutation and data type promotion, shows that the compilers fail in efficiently using vector permutations. Finally, Table IX shows by how much the speedup obtained by each compiler improves after manual transformations.

VI. RELATED WORK

Many papers were published in the 70's, 80's, and 90's about vectorization. Due to lack of space we do not discuss them here. Apart from those papers, the closest work to our work is that by Callahan, Donagarra and Levine where they designed a benchmark (TSVC benchmark in our study) to assess vectorizing compilers [5]. This benchmark only contains synthetic loops and the compilers they used are now 20 years old. Another similar empirical study on vectorization was done by Ren et al. [16] where they evaluated how well ICC (version 8.0) vectorized the Berkeley Multimedia Workload suite [19]. That benchmark only has integer operations, so the capabilities of the compiler to vectorize floating point operations was not assessed in that study. In this paper, in addition to ICC we also evaluated XLC and GCC. Finally, there is also a similar study by Knuth where he studied the impact on performance of several optimizations that he applied by hand [13]. That study did not consider loop vectorization.

VII. CONCLUSION

Our study shows that some of today's best compilers fail to vectorize many patterns. It is also clear that it is possible to attain significant improvements. In fact, XLC and ICC only auto vectorized 45-71% of our synthetic benchmarks and only about 18-30% of the loops extracted from the PACT and Media Bench II codes. However, at the same time these compilers can collectively vectorize about 83.05% of the synthetic benchmark and 48.49% of the PACT and Media Bench II. Our hand analysis indicates that well-known techniques would be successful on many of the loops that could not be vectorized by these compilers.

One reason for the current situation seems to be lack of resources. Compilers include a limited number of analyses and transformations. Just adding several optimization modules to compilers should lead to significant improvements. However, the uneven behavior of the compilers across the benchmarks indicates a second serious limitation: there is no universal criteria to determine what are the important patterns. Given limited resources the ability to identify what transformation should be given priority is of course of great importance, but there does not seem to exist a good methodology to asses the importance of patterns and therefore no methodology to determine what compiler transformations are the most important ones.

ACKNOWLEDGMENT

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation. This research was supported in part by the National Science Foundation under Award CCF 0702260.

REFERENCES

[1] Extended Test Suite for Vectorizing Compilers. http://polaris.cs.uiuc.edu/~maleki1/TSVC.tar.gz.
[2] Media Bench II. http://euler.slu.edu/~fritts/mediabench/.
[3] MIMD Lattice Computation (MILC) Collaboration. http://physics.indiana.edu/~sg/milc.html.
[4] Petascale Application Collaboration Teams. http://www.ncsa.illinois.edu/BlueWaters/pacts.html.
[5] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: a test suite and results. In *Supercomputing*, Supercomputing '88, pages 98–105. IEEE Computer Society Press, 1988.
[6] S.-C. Chen and D. Kuck. Time and parallel processor bounds for linear recurrence systems. *IEEE Trans. on Computers*, pages 701–717, 1975.
[7] D. L. David, D. Callahan, and J. Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17:1223–1244, 1991.
[8] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.
[9] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. Availability of f2c-a fortran to c converter. *SIGPLAN Fortran Forum*, 10:14–15, July 1991.
[10] F. Franchetti and M. Pschel. Generating simd vectorized permutations.
[11] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39:49–57, April 2004.
[12] W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.
[13] D. E. Knuth. An empirical study of fortran programs. *Software: Practice and Experience*, 1(2):105–133, 1971.
[14] S. Kurien and M. Taylor. Direct numerical simulation of turbulence: Data generation and statistical analysis. *Los Alamos Science*, 29:142–151, 2005.
[15] Y. Lin and D. A. Padua. A simple framework to calculate the reaching definition of array references and its use in subscript array analysis. In *Proc. of the 11 IPPS/SPDP'99 Workshops*, pages 1036–1045, 1999.
[16] G. Ren, P. Wu, and D. Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *IPDPS*, page 89.2. IEEE Computer Society, 2005.
[17] G. Ren, P. Wu, and D. Padua. Optimizing data permutations for simd devices. In *PLDI*, pages 118–131, New York, NY, USA, 2006. ACM.
[18] L. Shen, L. Huang, N. Xiao, and Z. Wang. Implicit data permutation for simd devices. In *Embedded and Multimedia Computing, 2009. EM-Com 2009. 4th International Conference on*, pages 1 –6, dec. 2009.
[19] N. T. Slingerland and A. J. Smith. Design and characterization of the berkeley multimedia workload. *Multimedia Syst.*, 8:315–327, July 2002.

App	Function	Perc	XLC Auto	ICC Auto	GCC Auto	XLC Manual	ICC Manual	GCC Manual	XLC Transf	ICC Transf	GCC Transf
DNS	multadd	26.5%	-	-	-	1.86	1.85	1.86	DP	DP	DP
	outerproduct3	16.7%	1.17	**2.20**	1.51	1.27	-	2.16	LIR	-	DP
	axpy	15.1%	**1.50**	**2.41**	**2.51**	-	-	-	-	-	-
	axpy2	20.3%	-	**1.19**	**2.06**	1.15	-	-	LIR	-	-
	vorticity_x	7.4%	-	1.59	-	1.64	1.68	2.60	DP	DP	DP
	vorticity_y	7.4%	-	1.59	-	1.63	1.69	2.05	DP	DP	DP
	vorticity_z	6.5%	-	1.91	-	2.63	2.16	2.13	DP	DP	DP
	Application Speedup		1.02	1.21	1.25	6.98	1.26	1.96	-	-	-
MILC	mult_su3_nn	26.6%	**2.26**	-	-	-	1.11	1.13	-	DP	DP
	add_lathwvec_proj	18.2%	-	-	-	-	1.20	1.12	-	DP	DP
	mult_su3_na	29.9%	**2.49**	-	-	-	1.13	1.21	-	DP	DP
	fieldlink_lathwvec	4.0%	**1.15**	-	-	-	1.84	1.26	-	DP	DP
	sitelink_lathwvec	4.1%	1.05	-	-	1.16	1.54	1.09	DP	DP	DP
	mult_su3_an	2.1%	**2.46**	-	-	-	1.63	1.43	-	DP	DP
	Application Speedup	-	1.46	-	-	1.46	1.23	1.14	-	-	-
JPEG Encoder	forward_DCT 1	38.5%	-	-	-	1.13	1.67	-	RR	RR	-
	forward_DCT 3		-	1.74	-	3.59	3.70	3.23	FSF	FSF	FSF
	jpeg_fdct_islow 1	30.8%	-	**1.32**	**1.63**	-	-	-	-	-	-
	jpeg_fdct_islow 2		-	-	**2.58**	-	2.49	-	-	IP	-
	grayscale_convert	2.9%	-	1.59	-	13.60	16.08	15.22	SR	SR	SR
	Application Speedup		-	1.33	1.15	1.39	2.13	1.79	-	-	-
JPEG Decoder	jpeg_idct_islow 1	62.1%	-	-	-	-	1.27	-	-	FSF-IP	-
	jpeg_idct_islow 2		-	-	-	-	1.26	1.20	-	DC-RC-IP	DC-RC
	Application Speedup		-	-	-	-	1.14	1.03	-	-	-
H263 Encoder	SAD_Macroblock	86.5%	-	-	-	1.26	3.18	2.64	DTP-RR	DTP-RR	DTP-RR
	Application Speedup	-	-	-	-	1.25	2.28	1.98	-	-	-
H263 Decoder	conv420to422	44.4%	-	-	-	2.2	3.67	3.60	RC	RC	RC
	conv422to444	44.4%	-	-	-	1.73	1.87	2.99	LP-RC	LP-RC	LP-RC
	Application Speedup	-	-	-	-	1.313	1.45	1.51	-	-	-
MPEG2 Encoder	dist1	77.3%	-	-	-	1.15	2.57	2.45	DTP-RR	DTP-RR	DTP-RR
	Application Speedup	-	-	-	-	1.06	1.96	1.86	-	-	-
MPEG2 Decoder	conv422to444	17.61%	-	-	-	2.19	1.38	2.82	LP-RC	LP-RC	LP-RC
	conv420to422	14.81%	-	-	-	1.99	3.64	3.74	RC	RC	RC
	Saturate	9.84%	-	-	2.04	1.56	2.10	2.29	RE	DP	RE
	idctcol	9.30%	-	-	-	1.97	3.23	3.43	RC	RC	RC
	Application Speedup	-	-	-	1.13	1.37	1.45	1.63	-	-	-
MPEG4 Encoder	pix_abs16_c	34.7%	-	-	-	2.11	3.42	3.01	DTP-RR	DTP-RR	DTP-RR
	pix_abs16_xy2_c	7.4%	-	-	-	2.73	4.89	3.15	DTP-RR	DTP-RR	DTP-RR
	pix_abs16_y2_c	3.0%	-	-	-	1.95	3.48	3.59	DTP-RR	DTP-RR	DTP-RR
	pix_abs16_x2_c	2.6%	-	-	-	1.64	4.00	3.28	DTP-RR	DTP-RR	DTP-RR
	Application Speedup	-	-	-	-	1.43	1.81	1.65	-	-	-
MPEG4 Decoder	v_resample	19.3%	-	**3.66**	2.34	2.34	-	3.37	IP-LIR	-	DP-IP
	Application Speedup	-	-	1.15	-	1.12	-	-	-	-	-

TABLE X

RESULTS FOR PACT AND MEDIA BENCH II APPLICATIONS AND TRANSFORMATION APPLIED TO VECTORIZE IT. THE DESCRIPTION FOR ALL OF THE ACRONYMS CAN BE FOUND IN SECTION VI.