EXPERIMENTS WITH AUTOMATIC VECTORIZATION FOR THE PENTIUM® 4 PROCESSOR

Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian

Intel Corporation
Microcomputer Software Laboratories
2200 Mission College Blvd. SC12-301, Santa Clara, CA 95052, USA
aart.bik@intel.com

Abstract

The Pentium® 4 Processor features SIMD extensions to the Intel® Architecture that enhance the performance of computational intensive applications that perform the same operation on different elements in a data set. To take full advantage of these extensions, the high-performance Intel® C++/Fortran compiler developed at the Microcomputer Software Laboratories supports the automatic translation of serial loops into the SIMD instructions of the Pentium® 4 Processor. In this paper, we present an initial assessment of the effectiveness of this approach by studying the performance impact of automatic vectorization on some benchmarks.

I. INTRODUCTION

Recent extensions to the Intel® Architecture exploit SIMD parallelism to enhance execution bandwidth in mainstream computing. These extensions feature instructions that let multiple functional units operate simultaneously on so-called packed data elements, i.e. relatively short vectors that reside in memory or registers. The Pentium® Processor with 64-bit MMX™ technology [7,12], for example, supports integer operations on 8 packed bytes, 4 packed words or 2 packed dwords. The Pentium® III Processor introduced 128-bit SSE (Streaming-SIMD-Extensions) [13], providing support for floating-point operations on 4 packed single-precision floating-point numbers. The Pentium® 4 Processor further extended this support with SSE2 that features operations on 2 packed double-precision floating-point numbers and 128-bit packed integers [11,13]. Because a single instruction processes multiple data elements in parallel, these extensions are useful to utilize SIMD parallelism in numerical and multimedia applications.

The translation of serial code into a form that exploits the SIMD extensions to the Intel® Architecture can be done manually using, for example, in-line assembly, intrinsic functions, or other language extensions, such as the SWARC language [10]. Although such explicit methods can be extremely effective, they also can greatly complicate program development and maintenance. On the other hand, letting a compiler do (at least part of) this translation automatically, a process we will refer to as **intra-register vectorization**, has a number of advantages. First, automatic vectorization has been well studied in the past and many techniques to convert serial code into vector form are described in the literature [2,14,15,17,18,19]. Second, the approach is less error-prone and simplifies the task of the programmer. Third, it enables the vectorization of existing software, which avoids potentially huge investments that would be required to rewrite this code into SIMD form. Finally, the approach is flexible, because one serial program can be mapped to different vectorized versions, each of which is specifically tailored for the peculiarities of a particular target architecture.

In this paper, we study the performance impact of intra-register vectorization for the Pentium® 4 Processor as it is done by the high-performance Intel® C++/Fortran compiler developed at the Microcomputer Software Laboratories. For an in-depth presentation of the automatic vectorization methods used by this compiler we must refer to [5].

II. PRELIMINARIES

This section provides a brief overview of the SIMD features of the Pentium® 4 Processor and the way in which the high-performance Intel® C++/Fortran developed at the Microcomputer Software Laboratories compiler takes advantage of these extensions to the Intel® Architecture.

A. SIMD FEATURES OF THE PENTIUM® 4 PROCESSOR

The Pentium® 4 Processor can exploit SIMD parallelism to improve performance by letting a single instruction process multiple data elements in parallel. The SIMD features consist of SSE and SSE2 (Streaming-SIMD-Extensions), supporting operations on packed single- and double-precision floating-point numbers, as well as on packed integer data elements [13]. All these SIMD features combined yield the following extensions to the Intel® Architecture.

1. Eight 128-bit registers: 'xmm0' through 'xmm7'.

2. Packed floating-point and integer data types:

    a. Two 128-bit floating-point data types: 4 packed single-precision and 2 packed double-precision floating-point numbers.

    b. Four 128-bit integer data types: 16 packed bytes, 8 packed words, 4 packed dwords, and 2 packed qwords.

3. New SIMD instructions that operate on the packed data types.

The floating-point instructions perform IEEE-754 compliant operations in an SIMD fashion. For instance, the instruction 'addps' yields 4-way SIMD parallelism by adding 4 individual single-precision floating-point numbers in the source operand to 4 individual single-precision floating-point numbers in the destination operand. The integer instructions perform SIMD integer operations that use either wrap-around or saturation arithmetic for the individual data elements. The instruction 'paddusb', for example, adds 16 unsigned bytes in the source operand to 16 unsigned bytes in the destination operand, saturating the value of individual data elements to '0xFF' on overflow.

The Pentium® 4 Processor features two kinds of 128-bit data movement instructions. Aligned moves (like 'movaps') transfer 128-bit packed data between memory and a 128-bit register, or between two 128-bit registers. The memory address must be aligned on a 16-byte boundary, or else an exception occurs. Unaligned moves (like 'movups') must be used if this assumption on the alignment cannot be made. Also, aligned memory operands can be used directly as operands of most SIMD instructions, whereas unaligned memory operands must be first loaded into one of the 128-bit registers. Because aligned data movement instructions execute much more efficiently than their unaligned counterparts, a substantial part of effectively exploiting the SIMD features of the Pentium® 4 Processor consists of enforcing the appropriate alignment of all data operated upon.

B. INTRA-REGISTER VECTORIZATION

The SIMD instructions of the Pentium® 4 Processor can be used to speedup the execution of serial code. The following C-loop, for instance, can be implemented as shown at the right-hand-side. Here we assume that the compiler has aligned the start of each array 'a', 'b' and 'c' at 16-byte boundaries in memory to enable the use of efficient aligned data movement instructions.

```
double a[100], b[100], c[100];
. . .
for (i = 0; i < 100; i++) {
   a[i] = b[i] * c[i];
}
```

```
L: movapd xmm0, b[ecx] ; load     2 DP from b
   mulpd  xmm0, c[ecx] ; multiply 2 DP from c
   movapd a[ecx], xmm0 ; store    2 DP into a
   add    ecx, 16      ;
   cmp    ecx, 800     ;
   jl     L            ; looping logic
```

The Intel® C++/Fortran compiler performs intra-register vectorization by following the standard approach to the vectorization of inner loops [2,14,15,17,18,19]. First, opportunities to exploit implicit parallelism in the loops of a serial program are found using data/control flow and data dependence analysis [1,3,4,9,16]. Subsequently, statements in a loop are reordered according to a topological sort of the acyclic condensation of the data dependence graph for this loop. Then, statements involved in a data dependence cycle are either recognized as certain idioms that can be vectorized, or distributed out into a loop that will remain serial. Finally, vectorizable loops are translated into the SIMD instructions of the Pentium® 4 Processor. This translation is accompanied by a number of alignment optimizations used to increase the number of aligned data movement instructions. If the lower bound of the loop shown above is changed into '1', for instance, the compiler will statically peel off a single iteration to obtain aligned access patterns on the three arrays.

```
for (i = 1; i < 100; i++)
   a[i] = b[i] * c[i];
```

→

```
a[1] = b[1] * c[1];
for (i = 2; i < 100; i++)
   a[i] = b[i] * c[i];
```

More advanced dynamic alignment optimizations are used to deal with situations where alignments cannot be determined at compile-time and where different access patterns in a loop have different initial (un)-alignments. Although we will provide examples of such (and other) optimizations later in this paper, for a more detailed presentation of the intra-register vectorization methods used by the Intel® C++/Fortran compiler, we must refer to [5].

III. EXPERIMENTAL RESULTS

In this section, we study the performance impact of automatic intra-register vectorization on a number of micro-kernels and benchmarks. All experiments have been conducted on a 1.5 GHz. Pentium® 4 Processor with 256MB of memory.

A. FLOATING-POINT KERNELS: SDOT/DDOT AND SAXPY/DAXPY

First we will study four important floating-point kernels: SDOT, DDOT, SAXPY and DAXPY. The first two kernels compute the dot product of two single-precision or double-precision floating-point arrays, respectively, as shown below for a C implementation of SDOT.

```
float SDOT(float *dx, float *dy, int n) {
  int i;
  float d = 0.0f;
  for (i = 0; i < n; i++)
    d += dx[i] * dy[i];
  return d;
}
```

If both incoming arrays 'dx' and 'dy' are aligned at a 16-byte boundary, then the most straightforward way to implement this reduction on the Pentium® 4 Processor is as follows. After the loop, the accumulator 'xmm0' contains 4 partial sums that must be added into the final scalar result 'd'. Also, a serial cleanup has to be generated to deal with iterations that remain in case the trip count cannot be divided evenly by the vector length (see [5] for the exact code sequences).

```
        pxor      xmm0, xmm0        ; initialize accumulator | 0  0  0  0 |
L:      movaps    xmm1, [ebp+eax*4] ; load       4 SP from dx
        mulps     xmm1, [ecx+eax*4] ; multiply   4 SP from dy
        addps     xmm0, xmm1        ; accumulate 4 partial sums
        add       eax, 4            ;
        cmp       eax, esi          ;
        jl        L                 ; looping logic
```

Because the alignments of parameters 'dx' and 'dy' are not known at compile-time, however, the compiler must be ready to somehow deal with all possible alignments with respect to 16-byte boundaries. In this case there are 4x4=16 possibilities if we assume that float array elements are at least aligned at 4-byte boundaries, or 16x16=256 otherwise. This number increases even further if more arrays are referenced in the loop or if we are interested in the alignment relative to cache lines to enable the generation of special instruction sequences for accesses that split a cache line [5]. This causes a trade-off between code size and testing overhead on one hand and vector loop performance on the other hand (using sub-optimal implementations for certain alignment combinations vs. generating the best possible implementation for each of the relevant alignment combinations).

The Intel® C++/Fortran compiler deals with this trade-off by the use of **dynamic loop peeling alignment strategies**, where first a few iterations are executed serially until access to a certain array (preferable one with many occurrences) becomes 16-byte aligned. At this point, the compiler has the choice between (a) generating a single vector loop in which the alignment of this array is known and unaligned data movement instructions are used for all other array references, or (b) testing the resulting alignment of another array to decide between multiple versions of the vector loop that are optimized accordingly. The compiler selects this latter alignment strategy (b) for SDOT, as illustrated below. Here the first vector loop is implemented as shown above and an unaligned data movement instruction is used to load 'dx' in the second.[1]

```
p = dy & 0x0f;
if (p != 0) { /* serial peeling loop (assumes dy & 0x03 is nonzero) */
  p = (16 - p) / 4;
  for (i = 0; i < min(p,n); i++) d += dx[i] * dy[i];
}
if (((dx + p) & 0x0f) == 0) { /* vector loop with both dx and dy aligned */
  for (i = p; i < n; i++) d += dx[i] * dy[i];
}
else { /* vector loop with dy aligned and dx unaligned */
  for (i = p; i < n; i++) d += dx[i] * dy[i];
}
```

In Table 1, we present the performance of SDOT in MFLOPS (counting one addition and one multiplication per iteration) of a serial implementation (SEQ) and automatically vectorized implementations using the two alignment strategies (VEC-A and VEC-B, respectively) in case the incoming arrays of varying length are aligned at a 16-byte boundary (all numbers in this section are obtained by running the kernel many times and dividing the execution time accordingly, which implies that for smaller data sets effectively 'warm cache behavior' is measured). In the same figure, the corresponding speedups with respect to serial execution are plotted (S-A and S-B). This experiment clearly reveals that using the slightly more elaborate alignment strategy (b) clearly pays of in situations where all incoming data is aligned. Since both alignment strategies will execute similar vector code in case the alignment of the incoming arrays differ, the Intel® C++/Fortran compiler defaults to alignment strategy (b). Only if the size of the loop-body exceeds a certain threshold, alignment strategy (a) is used to prevent an excessive growth in code size.

[1] This code fragment assumes that float array elements are at least aligned at 4-byte boundaries. If initially "dy & 0x3" is nonzero, then the access to this array cannot be made 16-byte aligned. The code generated by the Intel® C++/Fortran compiler defaults to serial execution in such cases.

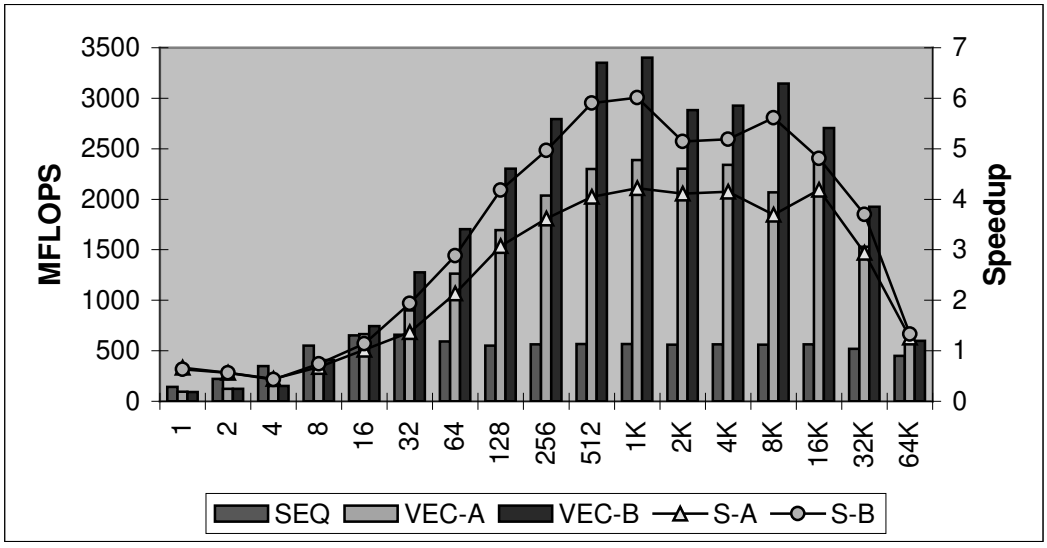

Table 1.Performance and Speedup for SDOT (16-byte boundary alignment).

The kernel DDOT is vectorized in a similar manner. In Table 2, we compare the execution times in nanoseconds of a serial (SEQ), vectorized (VEC) and an assembly implementation (ASM) for several small array lengths and different incoming alignment properties (where "a/b" denotes that 'dx&0xf == a' and 'dy&0xf == b'). The assembly version (courtesy Henry Ou) has been hand-optimized for the Pentium® 4 Processor to efficiently deal with all possible incoming alignments. Although the assembly version performs slightly better, the execution time trends of the hand-code and automatically generated vector code are quite similar. In Table 3, we present the performance in MFLOPS of these three different implementations in case the incoming arrays of varying length are aligned at a 16-byte boundary. In addition, the corresponding speedups with respect to serial execution are shown (S and S-ASM, respectively).

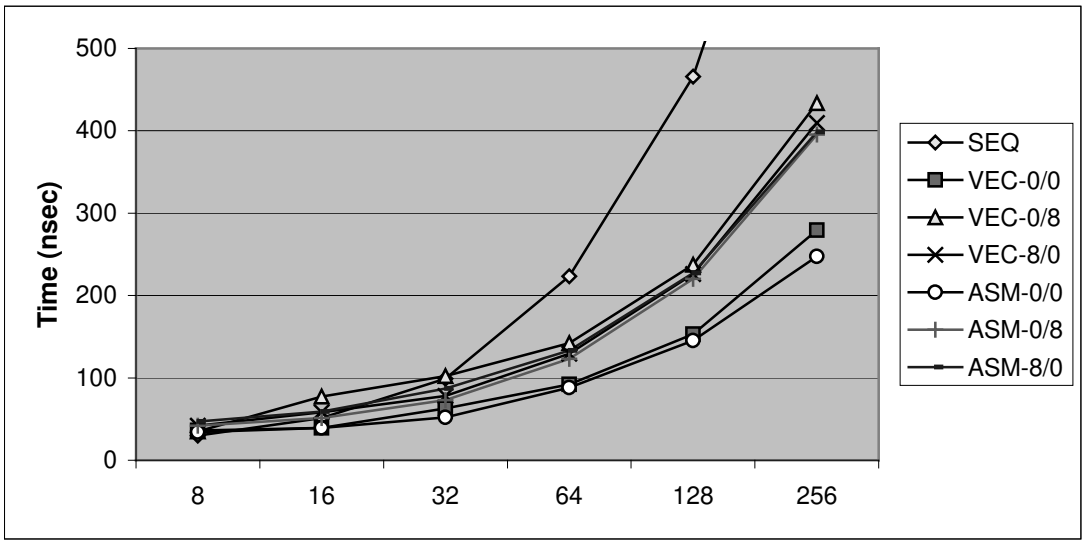

Table 2.Execution Time of DDOT (varying alignments).

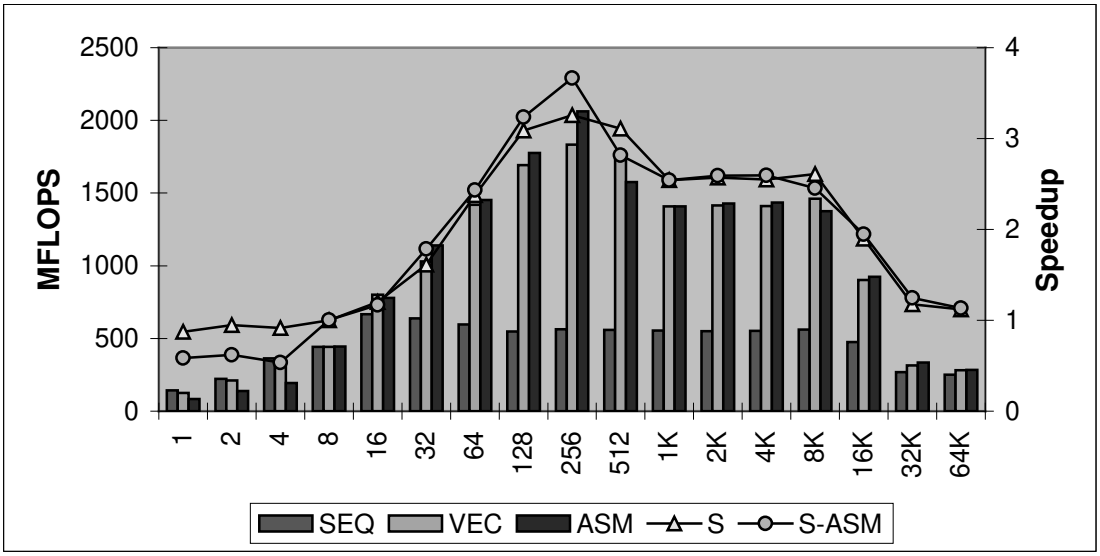

Table 3.Performance and Speedup for DDOT (16-byte boundary alignment).

The kernels SAXPY and DAXPY perform single-precision or double-precision floating-point array updates, respectively, as shown below for SAXPY.

```
void SAXPY(float *dx, float *dy, float s, int n) {
  int i;
  for (i = 0; i < n; i++)
     dy[i] += s * dx[i];
}
```

Automatic intra-register vectorization of this kernel has to deal with an additional complication caused by the use of pointers. Without any further points-to information [16], the compiler must conservatively assume that data dependences are carried by the loop, which prevents vectorization. Our compiler solves this complication by the use of **dynamic data dependence testing**, where an initial test for overlap decides between vector and serial execution of the loop, as illustrated below.

```
xH = dx + n; yH = dy + n;
if (xH <= dy || dx >= yH) {
  for (i = 0; i < n; i++)     /* vector loop */
     dy[i] += s * dx[i];
}
else {
  for (i = 0; i < n; i++)     /* serial loop */
     dy[i] += s * dx[i];
}
```

In Table 4, we present the performance in MFLOPS of the vectorized versions of SAXPY (VEC-SP) and DAXPY (VEC-DP) together with the speedups (S-SP and S-DP) with respect to the serial implementations of these kernels. Here we see that despite the runtime overhead of dynamic dependence testing, speedup is already obtained at length 8 (in fact, avoiding the runtime test by using a compiler hint "#pragma ivdep" in the original loop has no significant impact on the reported performance). To save on die size and power cost, only a single FP multiplier is used on the Pentium® 4 Processor to multiply two double-precision numbers in a single SIMD instruction [11]. This clearly limits the speedup of DAXPY.

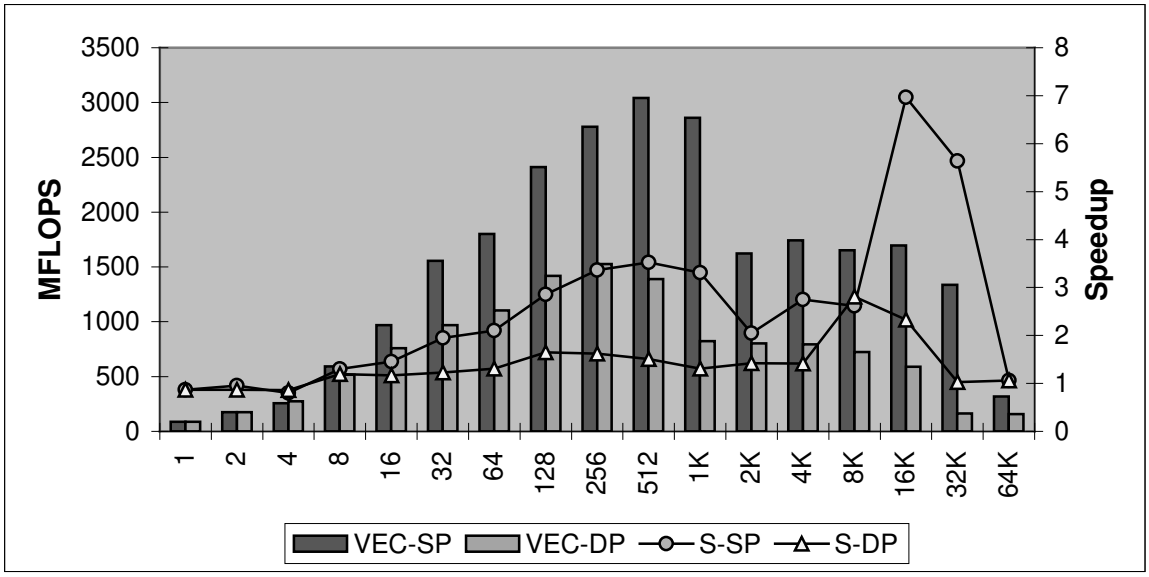

Table 4.Performance and Speedup for SAXPY and DAXPY (16-byte boundary alignment).

B. LINPACK BENCHMARK

In this section, we present the performance reported by a LINPACK [8] benchmark (available at http://www.netlib.org/benchmark/) consisting of a single- or double-precision linear equation solver. A substantial part of the runtime of this benchmark results from repetitively calling the Level 1 BLAS routine SAXPY or DAXPY for different sub-columns of the coefficient matrix during factorization, causing different alignment properties of the incoming arrays in these calls. Because the Level 1 BLAS routines are vectorized as discussed in the previous section, however, each call is executed with the appropriate SIMD instructions.

The performance in MFLOPS for a serial version (SEQ) and a version obtained by automatic intra-register vectorization (VEC) of this benchmark are shown in Table 5 for a 100x100 matrix (with leading dimensions 201 and 200).

MFLOPS	LDA=201		LDA=200	
	SEQ	**VEC**	**SEQ**	**VEC**
SP	53.3	670.2	53.3	955.5
DP	589.5	716.3	585.3	710.7

Table 5.Linpack Performance (single- and double-precision).

C. SUNSET BENCHMARK

This section discusses the performance of a rough water surface image simulation benchmark SUNSET. Based on the idea of Dmitry Abrosimov and further improved by Andrey Naraikin at Intel Nizhny Novgorod Labs (INNL) in Russia, this benchmark provides a real-time graphical display of the sun setting in a sea. The main algorithm of SUNSET (written in Fortran) spends a substantial part of its computational time executing two operations for every pixel in the picture (1) applying the SIN function to an array, and (2) computing the dot product of two arrays.

The first operation is vectorized using a 'short vector mathematical library' (also developed at INNL) that provides efficient software implementations for (inverse) trigonometric, (inverse) hyperbolic, exponential and logarithmic functions on (sub)arrays. This library, that is part of the Intel® C++/Fortran compiler, enables the automatic vectorization of loops containing such functions. The second operation is vectorized similar to the kernels discussed earlier. In Table 6, we report the performance in frames-per-second for different picture sizes. These numbers clearly illustrate the huge performance improvements that can be obtained on the Pentium® 4 Processor using the high-performance Intel® C++/Fortran compiler. In fact, the performance obtained by automatic vectorization of a pure Fortran implementation of this benchmark is currently only about 1.5 times slower than a fully manually tuned version that uses inline assembly.

Frames/sec	SEQ	VEC
100x100	16.4	100.1
200x200	4.2	31.8
400x400	1.0	8.0

Table 6.SUNSET Performance.

D. SPEC CPU2000 BENCHMARKS

Finally we report some preliminary results for a few individual benchmarks of the industry-standardized CPU-intensive benchmark suite SPEC CPU2000 (see http://www.spec.org/). The benchmarks report the performance metrics as the ratio of elapsed execution time with respect to a reference machine. In Table 7, we show some results for the benchmarks 164.gzip (data compression), 171.swim (weather prediction), 178.galgel (computational fluid dynamics), 187.facerec (image processing), and 200.sixtrack (nuclear physics). The reported ratios are normalized with respect to the default optimizations (O2) of the Intel® C++/Fortran compiler. The column marked with OPT shows the normalized ratios obtained with all optimizations specific to the Pentium® 4 Processor enabled (including advanced optimizations such as Profile-Guided-Optimization), but without vectorization. The column marked with OPT+VEC shows the normalized ratio for these optimizations combined with intra-register vectorization.

Ratio	OPT	OPT+VEC
164.gzip	1.20	1.24
171.swim	2.12	2.30
178.galgel	1.34	1.43
187.facerec	2.61	3.20
200.sixtrack	1.02	1.06

Table 7.SPEC CPU2000 Benchmarks Performance (ratios estimated with under-development-software).

In the integer benchmark 164.gzip, about 8% of the execution time is spent in code that has the following form for an unsigned short array 'head'.

```
for (n = 0; n < HASH_SIZE; n++) {
    m = head[n];
    head[n] = (Pos)(m >= 32768 ? m-32768 : 0);
}
```

The Intel® C++/Fortran compiler recognizes such saturation idioms and implements the loop-body with a single SIMD instruction 'psubusw', which boosts the performance of the loop substantially, contributing to an extra improvement of the benchmark as a whole. The speedup of the other floating-point benchmarks generally results from intra-register vectorization of loops similar to the kernels discussed in previous sections.

IV. CONCLUSIONS

In this paper, we have presented an initial assessment of the effectiveness of the intra-register vectorization methods used by the high-performance Intel® C++/Fortran compiler developed at Microcomputer Software Laboratories. This compiler supports the automatic translation of serial loops into the SIMD instructions of the Pentium® 4 Processor. Because the alignment of access patterns can have a big impact on performance, the compiler has the disposal of a variety of alignment optimizations, including the dynamic loop peeling strategies presented in this paper. Dynamic data dependence testing is used to allow the vectorization of loops in which pointer analysis has failed to proof data independence between array references. The experiments clearly reveal that intra-register vectorization can obtain good speedup on a Pentium® 4 Processor without the need for any source code modifications. Hence, such an automatic exploitation of implicit parallelism provides a convenient way for programmers who are not familiar with the Intel® Architecture to boost the performance of their applications. In addition, it may even assist expert programmers by minimizing the number of loops that have to be hand-optimized to exploit all available SIMD parallelism. Finally, the approach allows the automatic vectorization of existing serial software, which avoids potentially huge investments that would be required to hand-optimize this code.

Future research will focus on a more extensive validation of the effectiveness of automatic intra-register vectorization and the combination of this technique with automatic parallelization for shared-memory multiprocessors [6]. More information on high-performance compilers for the Intel® Architecture can be found at our website http://developer.intel.com/software/products/.

Acknowledgments The authors would like to thank the other members of the Intel® C++/Fortran compiler team who helped develop the compiler. We thank Andrey Naraikin for his extremely valuable feedback on performance issues for intra-register vectorization.

REFERENCES

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

2. J.R. Allen and K. Kennedy. *Automatic Translation of Fortran Programs to Vector Form*. ACM Transactions on Programming Languages and Systems, 9:491—542, 1987.

3. Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

4. Utpal Banerjee. *Dependence Analysis*. Kluwer, Boston, 1997. A Book Series on Loop Transformations for Restructuring Compilers.

5. Aart J.C. Bik, Paul M. Grey, Milind Girkar, and Xinmin Tian. *An Auto-vectorizing Compiler for the Intel® Architecture*. Submitted to ACM Transactions on Programming Languages and Systems, 1999.

6. Aart J.C. Bik, Paul M. Grey, Milind Girkar, and Xinmin Tian. *Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems*. Intel Technology Journal, 2001, http://intel.com/technology/itj/.

7. David Bistry et.al. *The Complete Guide to MMX™ technology.* McGraw-Hill, Inc. New York, 1997.

8. J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving linear systems on vector and shared memory computers.* SIAM, Philadelphia, PA, 1991.

9. C.N. Fisher and R.J. LeBlanc. *Crafting a Compiler.* Benjamin-Cummings, Menlo Park, California, 1988.

10. R.J. Fisher and H.G. Dietz. *Compiling for SIMD within a Register.* 1998 Workshop on Languages and Compilers for Parallel Computing, University of North Carolina at Chapel Hill, North Carolina, August 7-9, 1998.

11. Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. *The Microarchitecture of the Pentium® 4 Processor.* Intel Technology Journal, 2001, http://intel.com/technology/itj/.

12. Intel Corporation. *Intel Architecture MMX™ technology – Programmer's Reference Manual.* Intel Corporation, Order No. 243007-003, available at http://developer.intel.com/, 1997.

13. Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.* Intel Corporation, available at http://developer.intel.com/, 2001.

14. David J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, New York, 1978, Volume 1.

15. John M. Levesque and Joel W. Williamson. *A Guidebook to Fortran on Supercomputers.* Academic Press, Inc., San Diego, 1991.

16. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman Publishers, 1997.

17. Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer, Boston, 1988.

18. Michael J. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, Redwood City, California, 1996.

19. Hans Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.