International Journal of Parallel Programming, Vol. 30, No. 2, April 2002 (© 2002)

Automatic Intra-Register Vectorization for the Intel® Architecture

Aart J. C. Bik,[1] Milind Girkar,[1] Paul M. Grey,[1] and Xinmin Tian[1]

Received July 17, 2001; revised December 6, 2001

Recent extensions to the Intel® Architecture feature the SIMD technique to enhance the performance of computational intensive applications that perform the same operation on different elements in a data set. To date, much of the code that exploits these extensions has been hand-coded. The task of the programmer is substantially simplified, however, if a compiler does this exploitation automatically. The high-performance Intel® C++/Fortran compiler supports automatic translation of serial loops into code that uses the SIMD extensions to the Intel® Architecture. This paper provides a detailed overview of the automatic vectorization methods used by this compiler together with an experimental validation of their effectiveness.

KEY WORDS: Compilers; instruction-level-parallelism; SIMD; vectorization.

1. INTRODUCTION

Computer designers have always tried to keep up with the demands for high performance. At the semiconductor level, for example, the speed of circuits has been increased and the packaging densities have been enhanced to obtain higher performance. Due to physical limitations on the speed of electronic components, however, other approaches to enhance performance have been taken as well. At an architectural level, many advances have been made that either reduce **latency**, i.e., the time between start and completion of an operation, or increase **bandwidth**, i.e., the width and rate of operations.[1–4]

[1] Intel Corporation, 2200 Mission College Blvd. SC12-301, Santa Clara, California 95052. E-mail: aart.bik@intel.com

0885-7458/02/0400-0065/0 © 2002 Plenum Publishing Corporation

Since the early days of supercomputing, there have been architectural advances that utilize **data parallelism** to improve execution bandwidth. This form of parallelism arises in many numerical applications in science, engineering and image processing where a single operation is applied to multiple elements in a data set, usually a vector or a matrix. One way to utilize data parallelism that has proven its effectiveness in early vector processors (like the Cray 1[5]) is **data pipelining**. In this approach, vectors of data stream directly from memory or vector registers to pipelined functional units (possibly chained) and back. The process of vectorization, i.e., transforming serial code into vector instructions to obtain high performance, can be a cumbersome task for programmers. Therefore, successfully using vector processors generally depends on vectorizing compilers that can do this process with none or little assistance of the programmer. Over the years, compiling for vector processors, with a traditional emphasis on Fortran programs, has become one of the more mature areas of compiling for parallel architectures. A closely related approach to utilize data parallelism is **replication**. Massively parallel supercomputers that are based on the SIMD (single-instruction-multiple-data) technique (like the Maspar MP-1[6]) consists of a single control unit that dispatches instructions to an ensemble of simple processing elements that execute each instruction synchronously on different data items. Rewriting serial code into a form that exploits this kind of lock-step parallelism is usually also referred to as vectorization, although it has the added complexity of efficiently dealing with the private memories of processing elements and the interconnection network.

More recently, extensions to the Intel® Architecture have started to use the SIMD technique as a way to enhance execution bandwidth in mainstream computing. In this approach, multiple functional units operate simultaneously on so-called packed data elements, i.e., relatively short vectors that reside in memory or registers. Since a single instruction processes multiple data elements in parallel, this form of instruction-level-parallelism provides a new way to utilize data parallelism. The Pentium® Processor with 64-bit MMX™ technology,[7,8] for example, supports integer operations on 8 packed bytes, 4 packed words or 2 packed dwords. The Pentium® III Processor and Pentium® 4 Processor introduced 128-bit SSE and SSE2 (Streaming-SIMD-Extensions),[9,10] respectively, providing support for floating-point operations on 4 packed single-precision and 2 packed double-precision floating-point numbers, respectively, as well as integer operations on 16 packed bytes, 8 packed words and 4 packed dwords in SSE2. Because from an architectural point of view, these SIMD extensions differ from the two other approaches described above, we use the term **intra-register vectorization** to specifically refer to the conversion of

serial code into a form that utilizes the MMX™ technology or SSE/SSE2. To date, most intra-register vectorization is done explicitly using assembly, intrinsic functions, or language extensions. Although such explicit methods can be very effective, there are certain advantages to letting a compiler do (at least part of) intra-register vectorization automatically. 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.[11–16] 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 more 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 (cache line size, relative cost of instructions, etc.). In this paper, we discuss the automatic intra-register vectorization methods used by the high-performance Intel® C++/Fortran compiler. Because, as stated above, automatic vectorization is a mature research area that is well described in the literature, we mainly focus on the methods related specifically to intra-register vectorization.

The rest of this paper is organized as follows. In Section 2, some preliminaries are given. Section 3 covers automatic intra-register vectorization in detail. An experimental validation of the resulting performance is given in Section 4. Finally, related work and conclusions are presented in Sections 5 and 6, respectively.

2. PRELIMINARIES

In this section, we briefly introduce the MMX™ technology[7, 8] and SSE/SSE2.[9, 10]

2.1. MMX™ Technology

The first SIMD extensions became available with the 64-bit MMX™ technology on the Pentium® Processor and Pentium® II Processor.[7, 8] This technology consists of the following extensions.

- Eight 64-bit registers: `mm0` through `mm7`.
- Four 64-bit integer data types: 8 packed bytes, 4 packed words, 2 packed dwords, and 1 qword.
- Instructions that operate on the 64-bit integer data types.

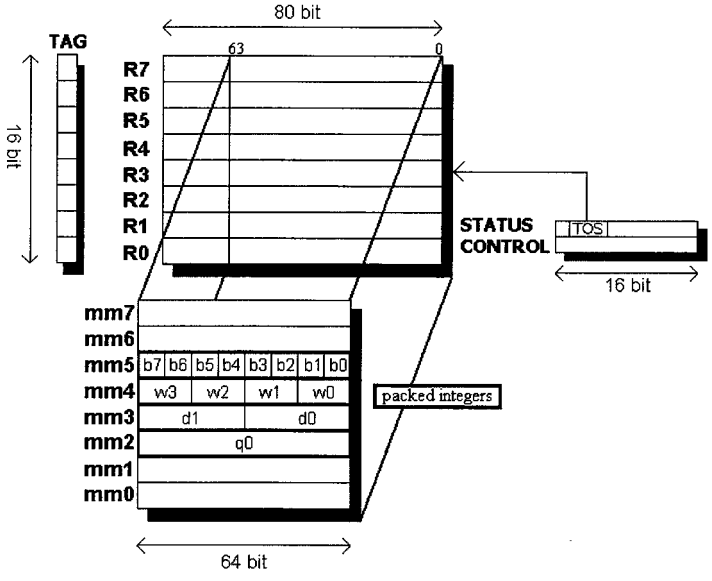

Fig. 1. MMX™ technology.

As illustrated in Fig. 1, the new register set is aliased to the data register stack of the Intel® Architecture FPU (Floating-Point Unit), mainly to keep the MMX™ technology fully transparent to the operating system. As a result, MMX™ code and FPU code cannot be mixed at the instruction level. Each floating-point code section should be exited with an empty FPU stack. Likewise, the instruction `emms` should be executed after each MMX™ code section to empty the FPU tag (all other MMX™ instructions fill the entire FPU tag, which would cause subsequent FPU instructions to produce unexpected results). In the same figure, we illustrate how the 64-bit integer data types are stored using little-endian order. The MMX™ technology supports arithmetic, comparison, conversion, logical, shift, and data movement instructions on the 64-bit integer data types (examples of these appear throughout this paper).

The arithmetic MMX™ instructions simultaneously process the individual data elements using either **wrap-around arithmetic** (where individual elements are truncated to the least significant bits) or **saturation arithmetic** (where individual elements are clipped to the appropriate data-range limit). The instruction `paddusb`, for example, yields 8-way SIMD parallelism by adding eight unsigned bytes in the source operand to eight unsigned bytes in the destination operand, clipping each individual result to `0xFF` on overflow. Finally, note that although the 64-bit MMX™ technology

operates more efficiently on memory addresses at an 8-byte boundary, there is only one 64-bit data movement instruction movq that can be applied to any address.

2.2. Streaming SIMD Extensions

The Pentium® III Processor introduced 128-bit SSE, which supports SIMD instructions on packed single-precision floating-point numbers.[10] The Pentium® 4 Processor further extended this support with 128-bit SSE2, which features SIMD instructions on packed double-precision floating-point numbers and integers.[9, 10] These technologies extend the Intel® Architecture as follows.

- Eight 128-bit registers: xmm0 through xmm7.
- Two 128-bit floating-point and four 128-bit integer data types:
  (a) 4 packed single-precision and 2 packed double-precision floating-point numbers.
  (b) 16 packed bytes, 8 packed words, 4 packed dwords, and 2 packed qwords.
- Instructions that operate on the 128-bit floating-point and integer data types.

The registers form an extension to the state of the Intel® Architecture, which must be explicitly saved and restored by the operating system during a context switch. In Fig. 2, we illustrate how the packed data elements are stored using little-endian order. The SSE/SSE2 instructions can be grouped in arithmetic, comparison, conversion, logical, shift or shuffle, and data movement instructions that operate on the 128-bit floating-point and

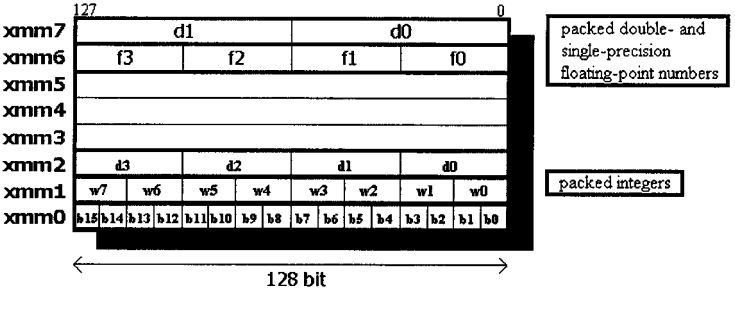

Fig. 2. Streaming SIMD extensions.

integer data types (examples appear throughout the paper). Furthermore, SSE/SSE2 supports some cacheability control instructions, such as prefetching instructions.

The floating-point arithmetic 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 are mostly direct extensions of the MMX™ technology to the wider data format. Finally, note that there are two kinds of 128-bit data movement instructions in SSE/SSE2. 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 at a 16-byte boundary, or else an exception occurs. The less efficient 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, as illustrated below with two instruction sequences that add data from memory to register xmm0.

```
movups xmm1, U_MEM ; add 4 SP          addps xmm0, A_MEM ; add 4 SP
addps  xmm0, xmm1  ; (unaligned)                         ; (aligned)
```

3. AUTOMATIC INTRA-REGISTER VECTORIZATION

Because the automatic detection of vector loops in serial code has been well studied in the literature,[11–16] in this section we mainly focus on the methods that are related specifically to intra-register vectorization, starting with a brief outline of the overall organization.

3.1. Organization of Intra-Register Vectorization

The approach taken by the Intel® C++/Fortran compiler to automatic intra-register vectorization is organized into three phases (a) analysis, (b) restructuring, and (c) vector code generation, with a strong interaction between the first two phases.

Program analysis consists of control flow, data flow and data dependence analysis[14–22] to provide the compiler with useful information on where implicit parallelism in the source program can be exploited. The data dependence analyzer is organized as a series of tests, progressively increasing in accuracy as well as time and space costs. First, the compiler tries to

prove independence between memory references by means of simple, inexpensive tests. If the simple tests fail, more expensive tests are used. Eventually, the compiler resorts to solving the data dependence problem as an integer linear programming problem that is attacked by the powerful but potentially expensive Fourier–Motzkin elimination method.[23, 24]

Program restructuring focuses on converting the source program into a form that is more amenable to analysis and, eventually, vectorization. For example, if static data dependence analysis of a program fails to prove independence, then the Intel® C++/Fortran compiler has the ability to generate **dynamic data dependence testing** to increase the opportunities for exploiting implicit parallelism in a program. Here, multi-version code of a loop is generated that is guarded by an overlap test for each pair-wise combination of accesses that could cause a data dependence. This idea is sketched below. In the resulting code, data independence may be assumed for the first loop.

```
float *p, *q;
...
for (i = L; i <= U; i++) {
  p[i] = q[i];
}
```

→

```
pL = p + 4*L; pH = p + 4*U;
qL = q + 4*L; qH = q + 4*U;
if (pH < qL || pL > qH) {
  /* loop without data dependence */
  for (i = L; i <= U; i++) p[i] = q[i];
}
else {
  /* loop with potential overlap */
  for (i = L; i <= U; i++) p[i] = q[i];
}
```

Other examples of transformations that are done during program restructuring are traditional compiler optimizations (such as constant/copy propagation and constant folding[17, 18, 21, 22]), loop transformations (such as loop interchanging or loop distribution[14–16]), and idiom recognition (such as the detection of reductions or MIN/MAX/ABS operators in user code).

Because vectorization only preserves lexically forward data dependences, lexically backward data dependences and data dependence cycles must be dealt with prior to a translation into vector instructions. The Intel® compiler follows the standard approach to vector code generation.[14–16] First, the statements in innermost loops are reordered according to a topological sort of the acyclic condensation of the data dependence graph, which clusters statements involved in dependence cycles and makes the dependences between all others lexically forward. Second, statements involved in a data dependence cycle are either recognized as certain idioms (such as reductions) or are distributed out into a loop that will remain serial. Finally, vector code is generated for all vectorizable loops.

3.2. Intra-Register Vectorization

Given a vectorizable loop with loop-body B(I), intra-register vectorization consists of strip-mining the loop with a vector length VL and replacing the statements in B(I) by corresponding SIMD instructions that operate on VL data elements in parallel. Below, this is sketched using so-called subscript triplet notation to denote SIMD instructions.

```
DO I = 1, N          DO I = 1, N, VL // assume VL evenly divides N
  B(I)         →       B(I:I+VL-1)
ENDDO                ENDDO
```

Usually, the compiler must also generate some test code and a **serial cleanup loop** to deal with any remaining iterations in case VL does not evenly divide the trip count. For clarity of explanation, however, we omit such implementation details from our code examples. The focus of this paper is on countable loops, i.e., loops for which a runtime expression for the trip count can be constructed. Furthermore, in order to obtain a uniform vector length VL, the Intel® compiler only proceeds with vectorization of a loop *if all unit stride memory references have the same type*, which is referred to as the **loop type**. A statement like da[i] = fa[i], where da and fa are a double- and single-precision floating-point array, respectively, will prohibit vectorization. As we will see in subsequent sections, however, certain mixed-type loops can still be vectorized. Note that, given a loop type, all data dependences with a distance that exceed the vector length of the corresponding packed data type can be pruned from the data dependence graph before deciding whether vectorization is legal. Because for intra-register vectorization the vector lengths are relatively short (ranging from VL = 2 to VL = 16), this implies that some loops that could not be vectorized on traditional vector processors can still be converted into MMX™ or SSE/SSE2 code.

Intra-register vectorization of a loop that operates on merely singly typed, unit stride memory references is straightforward. Each operator or load/store operation is replaced by an equivalent SIMD instruction, where the precision of the individual data elements is defined by the loop-type.

For loops that operate on integer data, the MMX™ technology and SSE2 provide SIMD instructions with 32-bit, 16-bit, and 8-bit precision for the individual data elements (packed dwords, words and bytes, respectively). Vectorization may proceed *if the precision of all final results in the original fragment is preserved.* For example, given a 32-bit and 8-bit implementation of the types int and char, respectively, the C standard[25] dictates that the addition in the following loop should be done in 32-bit precision (after the operands have been promoted accordingly by an implicit

type conversion). Because eventually only an 8-bit value is stored back (again by means of an implicit type conversion), however, the operations may be done in lower precision 8-bit wrap-around arithmetic. When compiling for the 64-bit MMX™ technology, this implies that 8-way SIMD parallelism can be obtained for this loop with loop type char, as illustrated below.

```
char a[N], b[N], c[N];
...
for (i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}
```

```
Back: movq  mm0, _b[ecx] ; load  8 bytes from b
      paddb mm0, _c[ecx] ; add   8 bytes from c
      movq  _a[ecx], mm0 ; store 8 bytes into a
      add   ecx, 8       ;
      cmp   ecx, edi     ;
      jl    Back         ; looping logic
```

Vectorization is disabled, however, if higher order bits of intermediate results can contribute to the precision of the final result. A 32-bit shift-right operator, for instance, cannot be vectorized using only 16-bit precision, because this could result in the loss of higher order bits that are shifted into lower positions. Also, because the MMX™ and SSE/SSE2 instruction sets are not fully orthogonal (there are no byte shifts, for instance), not all integer operations can actually be vectorized.

For loops that operate on floating-point numbers, SSE/SSE2 provides SIMD instructions with 32-bit and 64-bit precision for the individual data elements (packed single-precision and double-precision floating-point numbers, respectively). The binary +, −, *, and /, MIN and MAX and unary SQRT operators are directly supported in hardware. Below, for example, we show how 2-way SIMD parallelism is obtained by intra-register vectorization of a double-precision loop.

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

```
Back: movapd xmm0, _b[ecx] ; load  2 DP from b
      mulpd  xmm0, _c[ecx] ; mult  2 DP from c
      movapd _a[ecx], xmm0 ; store 2 DP into a
      add    ecx, 16       ;
      cmp    ecx, edi      ;
      jl     Back          ; looping logic
```

In addition, the Intel® C++/Fortran compiler provides a "short vector mathematical library" (developed at Intel Nizhny Novgorod Labs) with efficient software implementations for trigonometric, hyperbolic, exponential and logarithmic functions on vectors. This library enables the automatic intra-register vectorization of loops that contain such functions.

Intra-register vectorization of loops containing loop invariant memory references, type conversions, induction variables, reductions, saturation arithmetic, and conditional statements is more elaborate. In the following sections we discuss how each of these constructs is handled.

3.3. Scalar Expansion

It is well known that before a *loop invariant memory reference* (like a constant, scalar or array with loop invariant subscripts) can be used at a right-hand side in a vector loop, it must be expanded into a vector in a prelude.[15] Below, we show how the punpcklbw instruction is used to expand a byte value in the least significant byte of integer register eax into the MMX™ register mm0.

```
movd      mm0, eax ; |00|00|00|00|--|--|--|al|
punpcklbw mm0, mm0 ; |--|--|--|--|--|--|al|al|
punpcklbw mm0, mm0 ; |--|--|--|--|al|al|al|al|
punpcklbw mm0, mm0 ; |al|al|al|al|al|al|al|al|
```

Scalar expansions of a single-precision floating-point variable sp and a double-precision floating-point variable dp into the SSE/SSE2 register xmm0 are shown below.

```
movss  xmm0, _sp     ; |00|00|00|sp|        movsd    xmm0, _dp  ; |00|dp|
shufps xmm0, xmm0, 0 ; |sp|sp|sp|sp|        unpcklpd xmm0, xmm0 ; |dp|dp|
```

Similar sequences can be given for all other packed data types. Furthermore, note that directly moving a compile-time expanded vector of constants from a read-only data segment in memory into the appropriate register can be used as a more efficient alternative to expand a constant.

A scalar that is defined at a left-hand side and possibly used later at a right-hand side, exists in expanded form during execution of the vector loop, followed by a last value assignment in a postlude (which may be omitted if the scalar is dead on exit of the loop). A last value assignment simply consists of moving the most significant data element back into the appropriate scalar. In the example shown below, for instance, shuffling the most significant element (denoted by x@63) in register xmm0 back into memory restores the last value of x.

```
                                mov    eax, -256          ;
float a[64], b[64], x;     Back:                        ;
...                             movaps xmm0, _a[eax+256]  ; load  4 SP from a
for (i = 0; i < 64; i++) {      movaps _b[eax+256], xmm0  ; store 4 SP into b
  x = a[i];                     add    eax, 16            ;
  b[i] = x;                     jne    Back               ; looping logic
}                                                        ; |x@63|x@62|x@61|x@60|
... use of x ...                shufps xmm0, xmm0, 3      ; |x@60|x@60|x@60|x@63|
                                movss _x, xmm0           ; store into x
```

When a scalar is assigned *conditionally* in a loop, it is generally infeasible to generate the last value assignment. Likewise, a scalar that is used before it is defined in a loop causes a *carry-around* behavior that may be hard to vectorize. Vectorization of such construct only proceeds under certain circumstances, as further discussed in Section 3.8.

3.4. Type Conversions

Given a loop that operates on integer data, all integer type conversions that keep the data at least as wide as the precision of the loop type can simply be ignored during intra-register vectorization (provided that higher order bits of intermediate results cannot contribute to the precision of the final results, as stated earlier). One example was actually already given at the beginning of Section 3.2. Another example is shown below. Here, given a 16-bit implementation of the type short, the compiler can simply use the 16-bit scalar expansion sequence for the 32-bit variable x.

```
                                movd      xmm0, _x         ; load 32-bits of x
                                punpcklwd xmm0, xmm0       ; and expand lower
                                pshufd    xmm0, xmm0, 0    ; 16-bits into 8 words
short a[N], b[N], c[N];    Back:                           ;
int x;                          movdqa    xmm1, _b[ecx*2]  ; load  8 words from b
...                             pmullw    xmm1, xmm0       ; mult  8 words
for (i = 0; i < N; i++) {       paddw     xmm1, _c[ecx*2]  ; add   8 words from c
  a[i] = x * b[i] + c[i];       movdqa    _a[ecx*2], xmm1  ; store 8 words into a
}                               add       ecx, 8           ;
                                cmp       ecx, eax         ;
                                jl        Back             ; looping logic
```

Integer type conversions with data that is narrower than the precision of the loop type are only allowed if the conversion can be safely done during a scalar expansion. The loop above still vectorizes, for instance, if the type of x is changed into char (using instruction movsx prior to the scalar expansion to implement sign-extension).

For loops operating on floating-point numbers, vectorization proceeds in either a relaxed mode (the default) or a conservative mode (when the compiler switch /Op for exactly preserving precision is given). In the relaxed mode, arbitrary mixes of single-precision and double-precision floating-point operators, constants, scalars and even simple integer sub-expressions are allowed at right-hand sides. In the conservative mode, on the other hand, only loops containing operations of one precision (either single-precision or double-precision floating-point) are vectorized. Vectorization of the following loop, for instance, only proceeds in the relaxed mode.

```
                                fld QWORD PTR _d          ; load   DP
float a[N], b[N];               fst DWORD PTR [esp+8]     ; store  SP
double d;                       movss     [esp+8]         ;
...                             shufps    xmm0, xmm0, 0   ; expand 4 SP
for (i = 0; i < N; i++) {   Back:                         ;
  a[i] = d + b[i];              movaps    xmm1, _b[ecx*4] ; load   4 SP from b
}                               addps     xmm1, xmm0      ; add    4 SP
                                movaps    _a[ecx*4], xmm1 ; store  4 SP into a
                                add       ecx, 4          ;
                                cmp       ecx, eax        ;
                                jl        Back            ; looping logic
```

We stated earlier that vectorization only proceeds if all unit stride memory references in a loop have the same type in order to obtain a uniform vector length. There a several obvious exceptions to this rule, however. Integer type conversion arising from the use of *signed* and *unsigned* data with the same precision can be ignored, provided that all operations can be implemented with appropriate SIMD instructions. This is illustrated below, where an *arithmetic* and *logical* shift are used to implement the signed and unsigned shift-right operation, respectively, as dictated by the C standard.[25]

```
signed   short a[N];    Back: movdqa xmm1, _b[eax*2] ; load  4 dwords from b
unsigned short b[N];          movdqa xmm0, _a[eax*2] ; load  4 dwords from a
...                           psrld  xmm1, 2         ; shift 4 dwords by 2
for (i = 0; i < N; i++) {     psrad  xmm0, 2         ; shift 4 dwords by 2
  a[i] = (b[i] >> 2)          paddd  xmm1, xmm0      ; add   4 dwords
       + (a[i] >> 2);         movdqa _a[eax*2], xmm1 ; store 4 dwords into a
}                             add    eax, 8          ;
                              cmp    eax, ecx        ;
                              jl     Back            ; looping logic
```

Type conversions due to mixing 32-bit integer and single-precision floating-point unit stride memory references allow for a conversion between vectors of the same length using the SIMD instructions **cvtdq2ps** and **cvtps2dq**. An example of such mixed-type loop vectorization is shown below.

```
float a[N], b[N];       Back:                          ;
int   c[N];                  movdqa   xmm0, _c[ecx*4] ; load    4 dwords from c
...                          cvtdq2ps xmm1, xmm0      ; convert 4 dwords into SP
for (i = 0; i < N; i++) {    mulps    xmm1, _b[ecx*4] ; mult    4 SP
  a[i] = b[i] * c[i];        movaps   _a[ecx*4], xmm1 ; store   4 SP into a
}                            add      ecx, 4          ;
                             cmp      ecx, ebp        ;
                             jl       Back            ; looping logic
```

3.5. Induction Variables

Typically, statements that implement induction variables[17] are removed from the vector loop and replaced by last value assignments in the postlude. Induction variables that appear at the right-hand side of other statements must be dealt with explicitly, however. Consider a right-hand expression that can be expressed as follows, where a and b denote arbitrary loop invariant expressions and I denotes the normalized loop index that iterates from 0 up to the trip count.

$$a \cdot I + b$$

To implement the first vector iteration, the compiler generates an instruction sequence (similar to the sequences presented for scalar expansion) that constructs the following vector (in little-endian order).

$$| \ldots | \ldots | 2 \cdot a + b | a + b | b |$$

For subsequent iterations, this vector is kept up-to-date by adding the scalar expanded value $a \cdot VL$. A simple MMX™ example for bytes is shown below (viz. a = 1, b = 0, VL = 8). The initialization in the prelude consists of moving a compile-time expanded vector of constants from a read-only data segment in memory into two MMX™ registers. Subsequently, these registers are used to implement SIMD induction in the loop-body.

```
                                    movq  mm1, _cnst$1      ; |8|8|8|8|8|8|8|8|
char a[128];                        movq  mm0, _cnst$2      ; |7|6|5|4|3|2|1|0|
...                                 mov   eax, -128         ;
for (i = 0; i < 128; i++) {    Back:                        ;
  a[i] = i;                         movq  _a[eax+128], mm0 ; store 8 bytes into a
}                                   paddb mm0, mm1          ; add   8 bytes
                                    add   eax, 8            ;
                                    jne   Back              ; looping logic
```

Floating-point induction is implemented similarly. Furthermore, the relaxed mode even allows integer induction within floating-point expressions. In this case, the SIMD induction is performed directly in floating-point arithmetic, as illustrated below (viz. a = 24, b = 25, VL = 2).

```
double a[N]; int x = 0;             movapd  xmm1, _cnst$1 ; |48.0|48.0|
...                                 movapd  xmm0, _cnst$2 ; |49.0|25.0|
for (i = 0; i < N; i++) {      Back:                      ;
  x = x + 4;                        movapd  _a[ecx], xmm0 ; store 2 DP into a
  a[i] = 6*x+1;                     addpd   xmm0, xmm1    ; add   2 DP
}                                   add     ecx, 16       ;
                                    cmp     ecx, edi      ;
                                    jl      Back          ; looping logic
```

A wrap-around induction variable is a special case of a scalar with *carry-around* behavior. For the loop shown below, for instance, all but the first iteration see the value i-1 for the wrap-around induction variable j. Vectorization of such loops may proceed after loop peeling[15] has been applied to peel off at least one iteration, after which all uses of the wrap-around induction variable can be replaced with the value i-1.

```
for (i = 0, j = N; i < N ; i++) { ... uses of j ... j = i; }
```

3.6. Reductions

A reduction is an operation that computes a scalar value from a set of values. If a reduction is implemented by means of a loop, then this gives rise to loop-carried data dependences that prohibit straightforward vectorization. Some reductions into a scalar variable can still be vectorized, however, by means of a slightly more elaborate scheme.

Summing all elements of an array into a scalar accumulator, for instance, is an example of a reduction defined by the operator +. Such a reduction can be vectorized by resetting an SIMD register in a prelude, adding partial results to this register in the loop using an SIMD instruction (e.g., paddw for words), and combining these partial results into the final result in a postlude. Below, we show how this final step can be implemented for some packed data types.

```
Add words in mm0:   Add dwords in mm0: Add SPs in xmm0:       Add DPs in xmm0:
*************************************************************************
movq  mm1, mm0      movq  mm1, mm0     movap   xmm1, xmm0     movaps   xmm1, xmm0
prslq mm1, 32       prslq mm1, 32      movhlps xmm1, xmm0     unpckhpd xmm1, xmm0
paddw mm0, mm1      paddd mm0, mm1     addps   xmm0, xmm1     addsd    xmm0, xmm1
movq  mm1, mm0                         movaps  xmm1, xmm0
prslq mm1, 16                          shufps  xmm1, xmm0, 1
paddw mm0, mm1                         addss   xmm0, xmm1
```

Figure 3 illustrates the accumulation of packed words into the lower part of a MMX™ register. After each of these sequences, the least significant part of the register must be added to the scalar accumulator. Similar sequences can be given for all other packed data types.

The Intel® compiler provides similar support for the operators +, -, *, MAX, MIN, &, and | with only trivial changes to the initial value assigned to the SIMD register in the prelude and the operations in the vector loop and postlude. Idiom recognition and some other supporting transformations

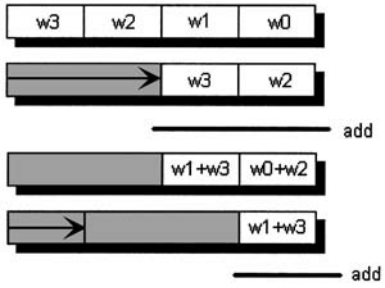

Fig. 3. Accumulation of packed words.

(such as the elimination of coupled reductions) help the compiler to vectorize more reductions. Note that because vectorizing a reduction may change the order in which partial results are computed, a different round off error may result in the final answer for floating-point numbers. Therefore, floating-point reductions are not vectorized in the conservative mode (under /Op).

In the example shown below, idiom recognition first converts the if-statement into the assignment statement x = MAX(a[i], x);. Subsequently, this reduction is vectorized.

```
double a[N], x;                 movsd     xmm0,  _x         ;
...                             unpcklpd  xmm0,  xmm0       ; expand x into 2 DP
for (i = 0; i < N; i++) {  Back:                            ;
  if (a[i] > x) {               movapd    xmm1,  _a[eax*8]  ; load 2 DP from a
    x = a[i];                   maxpd     xmm0,  xmm1       ; max  2 DP
  }                             add       eax,   2          ;
}                               cmp       eax,   ecx        ;
                                jl        Back              ; looping logic
                                                            ;
                                movapd    xmm1,  xmm0       ;
                                unpckhpd  xmm1,  xmm0       ;
                                maxsd     xmm0,  xmm1       ; compute final maximum
                                movsd     _x,    xmm0       ; and store into x
```

In general, we require that the type of the scalar used for the reduction is equal to the loop type. An exception is formed by the accumulation of the product of two 16-bit signed integer expressions into a 32-bit signed integer accumulator. This mixed-type reduction can be easily implemented using the MMX™ or SSE2 instruction pmaddw, as will be shown in Section 4.1. Likewise, accumulating the absolute differences of unsigned bytes is recognized as a special idiom that can use the SSE/SSE2 instruction psadbw.

3.7. Saturation Arithmetic

Advanced instruction selection is used to vectorize particular operations that can be efficiently mapped onto SIMD instructions that perform saturation arithmetic. Consider, for example, the following loop (the suffix letter u denotes an unsigned constant).

```
unsigned char x[N];
...
for (i = 0; i < N; i++) {
  x[i] = (x[i] >= 20u) ? x[i] - 20u : 0u;
}
```

The Intel® C++/Fortran compiler recognizes the saturation arithmetic performed in this code fragment (if the subtraction would yield a negative value, the result is saturated to 0x00) and converts the serial loop into the following SIMD instructions.

```
      movdqa   xmm0, _cnst$1 ; |20u|20u|...|20u|20u|
Back:                        ;
      movdqa   xmm1, _x[eax] ; load    16 bytes from x
      psubusb  xmm1, xmm0    ; sat-sub 16 bytes
      movdqa   _x[eax], xmm1 ; store   16 bytes into x
      add      eax, 16       ;
      cmp      eax, ecx      ;
      jl       Back          ; looping logic
```

The automatic detection of saturation idioms in user code can be quite elaborate. A detailed overview of how the Intel® compiler recognizes saturation and related idioms is given in Ref. 26.

3.8. Conditional Statements

The Intel® compiler supports vectorization of singly nested conditional statements by a technique called **bit masking**. Given an if-statement in a loop that is under control of a condition C(I), vectorization proceeds by generating code that constructs a bit mask for VL successive values of the condition that has an all-ones bit mask for a TRUE value and an all-zeros bit mask for a FALSE value. Statements in the if-statement are subsequently vectorized according to the pattern below.

```
DO I = 1, N                  DO I = 1, N, VL
  IF (C(I)) THEN      →        G = BIT_MASK( C(I:I+VL-1) )
    A(I) = B(I)                A(I:I+VL-1) = (A(I:I+VL-1) & !G) | (B(I:I+VL-1) &  G)
  ELSE                         X(I:I+VL-1) = (X(I:I+VL-1) &  G) | (Y(I:I+VL-1) & !G)
    X(I) = Y(I)              ENDDO
  ENDIF
ENDDO
```

For the true-branch, the final values consist of a bit-wise or of the original values and new values that are masked by the negation of the bit mask and by the bit mask itself, respectively. For the false-branch, the opposite bit masking is performed. Note that because both branches are evaluated in the resulting code, benefits from vectorization could be nullified by this additional computational overhead.[2] Therefore, vectorization is disabled if the ratio of masked statements versus unmasked statements exceeds a certain threshold.

There are some obvious ways to improve the performance of the masked code. If A and X denote the same array, then the following optimized pattern can be used.

```
DO I = 1, N, VL
  G = BIT_MASK( C(I:I+VL-1) )
  A(I:I+VL-1) = (Y(I:I+VL-1) & !G) | (B(I:I+VL-1) & G)
ENDDO
```

Within a masked expression, the following two rules (and similar variants obtained by the commutativity of operators) are applied to hoist common sub-expressions out of the logical operations for any of the operators $\oplus \in \{+, -, *, /\}$ for the first, and $\oplus \in \{+, -\}$ for the second.

$$((E \oplus X)\ \&\ G)\,|\,((E \oplus Y)\ \&\ !G) \rightarrow E \oplus ((X\ \&\ G)\,|\,(Y\ \&\ !G))$$

$$((E \oplus X)\ \&\ G)\,|\,(E\ \&\ !G) \rightarrow E \oplus (X\ \&\ G)$$

In combination with traditional compiler optimizations (like replacing x&0 with 0), the expressions that typically arise from bit masking can be implemented quite efficiently. The MMX™ technology and SSE/SSE2 support the construction of bit masks for conditions that are formed by any of the relational operators (although some rewriting may be required to construct a condition that is directly supported by an SIMD instruction). The instructions pand and pandn implement direct bit masking and negated bit masking, respectively. The following loop, for example, can be converted into rather compact MMX™ instructions by bit masking the negation of a comparison with the value 10 (expanded into mm2) with the value 1 (expanded into mm3).

[2] Another issue is that if the condition protects situations in which exceptions will occur (such as an integer division by zero), masking may move exceptions into the execution path. For the operations handled by the Intel® compiler, however, this problem does not occur.

```
int a[N];                Back: movq    mm1, _a[eax*4] ; load  2 dwords from a
...                            movq    mm0, mm2       ; copy  2 dwords with 10
for (i = 0; i < N; i++) {      pcmpgtd mm0, mm1       ; GT    2 dwords
  if (a[i] < 10)               pandn   mm0, mm3       ; mask  2 dwords with 1
    a[i] = 0;                  movq    _a[eax*4], mm0 ; store 2 dwords into a
  else                         add     ecx, 2         ;
    a[i] = 1;                  cmp     ecx, eax       ;
}                              jl      Back           ; looping logic
```

In general, when a scalar is assigned conditionally in a loop, efficient vectorization may become infeasible. Vectorization of conditionally assigned scalars only proceeds if (1) the scalar is **private** in the loop, i.e., each use of the scalar is dominated by a single definition in the loop-body and the scalar is dead on exit of the loop, or (2) the scalar is involved in a reduction. In the first situation, the scalar is merely used to carry a temporary value into another computation, and vectorization can proceed as if the scalar assignment is done unconditionally. The second situation is handled by applying the bit masking technique to the computation of partial results in the loop-body.

3.9. Static Alignment Optimizations

Since SSE/SSE2 distinguishes between aligned and unaligned data movement instructions, the compiler uses alignment information of data structures together with loop bound information to determine the initial alignment of all unit stride memory references in a vector loop. For example, if in the example shown below, the base of the 2-dimensional array `a` is allocated at a 16-byte boundary, then the compiler is able to determine that the references to this array in the j-loop always start at a 16-byte boundary, viz. each initial address is ***a+512*i+16***.

```
float a[128][128];              // base of a allocated at 16-byte boundary
for (i = 0; i < 128; i++)
  for (j = 4; j < n; j++)       // in this loop, array references
    a[i][j] = 0;                //    start at 16-byte boundary
```

Because SSE/SSE2 operations are 16 bytes wide, vectorized memory references that start at a 16-byte boundary remain aligned in all vector iterations, as illustrated with the access pattern at the top in Fig. 4. Such references can be implemented efficiently as aligned loads and stores (`movaps`, `movapd`, and `movdqa`) or as direct memory operands of SSE/SSE2 instructions. If all memory references have the same initial misalignment with respect to a 16-byte boundary, then the compiler uses loop peeling[15] to enforce aligned access patterns within the vector loop. For example, if in the Fortran example shown below the single-precision

floating-point arrays A and B are both allocated at a 16-byte boundary, then 3-fold loop peeling of the original loop converts the initial misalignment 4 with respect to a 16-byte boundary for A(I) and B(I) into aligned access patterns. Alignment properties of loop invariant memory references (such as A(1) in this example) are simply ignored, because such references are involved in scalar expansions.

```
REAL A(1:1000)                    A(2) = A(1) + B(2)
REAL B(1:1000)                    A(3) = A(1) + B(3)
...                               A(4) = A(1) + B(4)
DO I = 2, 1000                    DO I = 5, 1000
  A(I) = A(1) + B(I)      →         A(I) = A(1) + B(I)
ENDDO                             ENDDO
```

In general, an initial misalignment x with respect to a 16-byte boundary for a memory reference with an element size of b bytes can be resolved by $(16-x)/b$-fold loop peeling, assuming that arrays are aligned at least at an element size boundary. Unfortunately, loop peeling cannot help if *different* initial misalignments occur. In such cases, the unaligned memory references cannot be used directly as an operand of SSE/SSE2 operations and must be implemented using the less efficient unaligned loads and stores (movups, movupd, and movdqu). For example, if the single-precision floating-point array A is allocated at a 16-byte boundary, straightforward translation of the following example yields the vector code shown below.

```
                          Back: movups xmm0, _a[eax*4+4]  ; load  4 SP from A (+1)
REAL A(1:N)                     movups xmm1, _a[eax*4+24] ; load  4 SP from A (+6)
...                             mulps  xmm0, xmm1         ; mult  4 SP
DO I = 1, N-6                   movaps _a[eax*4], xmm0    ; store 4 SP into A
  A(I) = A(I+1) * A(I+6)        add    eax, 4             ;
ENDDO                           cmp    eax, edx           ;
                                jl     Back               ; looping logic
```

An additional performance penalty must be paid for unaligned memory references that cross a cache line boundary. Given a cache line size of 32 bytes, for example, there are three possible misaligned access patterns that can occur after vectorization of a single-precision floating-point memory reference in case arrays are at least allocated at an element size boundary. As illustrated in Fig. 4, every other iteration of the vector loop will cause a cache line split of $+4$, $+8$, or $+12$ bytes into the next cache line.

Some preliminary experiments revealed that movups xmm1, mem instructions that cause such cache line splits can be implemented more efficiently using one of the instruction sequences shown below. Similar sequences can be given for a cache line split of $+8$ for the movupd and movdqu instructions.

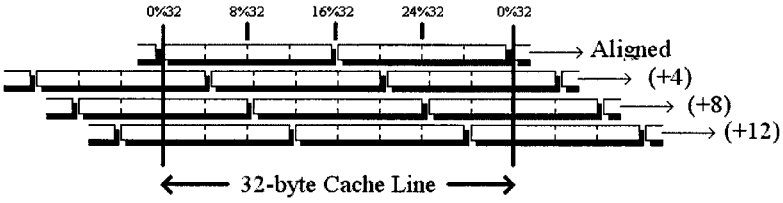

Fig. 4. Cache line splits in vector access patterns.

```
(+4):                          (+8):                    (+12):
movlps xmm1, mem               movlps xmm0, mem         movss  xmm1, mem
movss  xmm0, mem+8             movhps xmm0, mem+8       movhps xmm1, mem+4
movhps xmm0, mem+12                                     movlps xmm0, mem+8
shufps xmm1, xmm0, 0x84                                 shufps xmm1, xmm0, 0x48
```

Therefore, in order to alleviate the performance penalties of cache line splits, the compiler proceeds as follows in case some of the memory references in a vector loop have a known initial misalignment with respect to a cache line boundary that cannot be resolved with loop peeling. Given a cache line size of C bytes, first $C/16$-fold loop unrolling[15] is applied to the vector loop, which unifies the chunks of data handled in each iteration with the cache line size. Subsequently, the compiler uses one of the instruction sequences discussed above for each unaligned access in the unrolled version that actually crosses a cache line. Finally, the compiler uses aligned instructions for memory references at a 16-byte boundary, and uses the unaligned load and stores for all remaining memory references with either an unknown or harmless (mis)alignment. This scheme yields the following more efficient code for the example shown above.

```
back:  movups  xmm1, _a[eax*4+4]  ; + first unrolled vector iteration
       movlps  xmm0, _a[eax*4+24] ; + A(I+1) within cache line
       movhps  xmm0, _a[eax*4+32] ; + A(I+6) causes cache line split +8
       mulps   xmm1, xmm0         ; +
       movaps  _a[eax*4], xmm1    ; +
       movlps  xmm1, _a[eax*4+20] ; * second unrolled vector iteration
       movss   xmm0, _a[eax*4+28] ; * A(I+1) causes cache line split +4
       movhps  xmm0, _a[eax*4+32] ; * A(I+6) within cache line
       shufps  xmm1, xmm0, 0x84   ; *
       movups  xmm0, _a[eax*4+40] ; *
       mulps   xmm1, xmm0         ; *
       movaps  _a[eax*4+16], xmm1 ; *
       add     eax, 8             ;
       cmp     eax, edx           ;
       jl      back               ; looping logic
```

Note that although the original 64-bit MMX™ technology does not distinguish between aligned and unaligned load and stores, still performance penalties may arise for unaligned movq instructions. In such cases, loop peeling may be useful to enforce access patterns that start at an 8-byte boundary, and $C/8$-fold loop unrolling unifies the chunks of data handled in each iteration with a cache line size C, which enables the generation of special instruction sequences for cache line splits.

3.10. Dynamic ALignment Optimizations

Even with aggressive analysis, the compiler may still fail to obtain alignment information for all memory references, for example due to the use of pointers. In such cases, the compiler must be ready to deal with each possible alignment. The number of possible alignment combinations clearly increases with the number of memory references in the loop-body and the cache line size (in case special instructions for cache line splits have to be generated). 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 memory references, or (b) testing the resulting alignment of another memory reference to decide between multiple versions of the vector loop that are optimized accordingly.

We illustrate these strategies with an example. Suppose that in the following fragment, the alignments of the memory locations pointed to by dx and dy are not known at compile-time.

```
float *dx, *dy;
...
for (i = 0; i < n; i++)
   d += dx[i] * dy[i];
```

Under strategy (b), the following code is generated where aligned data movement instructions can be used for both variables in the first vector loop, whereas an unaligned data movement instruction has to be used to

load the elements of dx in the second.[3]

```
p = dy & 0x0f;
if (p != 0) { /* serial dynamic peeling loop (assumes dy & 0x03 is zero) */
   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];
}
```

Under strategy (a), only the false-branch of the if-statement that test the resulting alignment for dx would be generated. Since both alignment strategies will execute similar vector code in case the alignments of the incoming arrays differ, the Intel® C++/Fortran compiler defaults to alignment strategy (b). Only if code for the MMX™ technology is generated (where there is only one data movement instruction) or if the size of the loop-body exceeds a certain threshold, alignment strategy (a) is used to prevent an excessive growth in code size.

4. EXPERIMENTAL RESULTS

In this section we provide an experimental validation of the performance improvements obtained by intra-register vectorization. First, we study the performance impact on some small integer and floating-point kernels. Subsequently, we present improvements obtained for a number of benchmarks. Some preliminary experiments were already discussed in Ref. 27.

4.1. Integer Kernels

The first benchmark considered is an integer dot product kernel IDOT that is central to many signal-processing algorithms.[7] Below we show a function in C that computes the dot product of two 16-bit integer arrays into a 32-bit accumulator. When compiling for the MMX™ technology, this particular mixed-type reduction is vectorized as shown below. Alignment strategy (a) is used to make the access pattern of incoming array q

[3] This code fragment assumes that array elements are aligned at least at a 4-byte boundary. If initially dy & 0x3 is nonzero, then the access to the array cannot be made 16-byte aligned. The code should default to serial execution in such cases.

aligned at an 8-byte boundary for efficiency (the code for this loop peeling as well as the code for the final summation of the partial sums are omitted).

```
int IDOT(short *p, short *q, int n) {
   int x = 0, i;
   for (i = 0; i < n; i++)
     x += p[i] * q[i];
   return x;
}
```

```
pxor     mm0, mm0
Back:
movq     mm1, [ebp+ecx*2]
pmaddwd  mm1, [esi+ecx*2]
paddd    mm0, mm1
add      ecx, 4
cmp      ecx, edi
jl       Back
```

When compiling specifically for the Pentium® III Processor, the Intel® C++/Fortran compiler further optimizes this code by adding prefetching instructions to an unrolled version of the vector loop. In Fig. 5, we show the performance (in million operations per seconds) for 8-byte aligned arrays of varying length of a serial version (SEQ) and the vector versions compiled for the MMX™ technology (VEC-mmx) and the Pentium® III Processor (VEC-PIII) on a 500MHz Pentium® III Processor. The corresponding speedups (S-mmx and S-PIII, respectively) are also depicted in this figure. Here we see that arrays with lengths exceeding about 8 to 16 start to exhibit a speedup that is nicely sustained for most practical lengths. Timings were obtained by calling the kernel many times and dividing the total execution time accordingly. Therefore, the performance behaves regularly for the data sets that fit in cache. Cache effects are responsible for the performance anomalies that are observed for larger data sets.

When compiling for the Pentium® 4 Processor, the loop is vectorized similarly in SSE2 using alignment strategy (b). Because the Pentium® 4

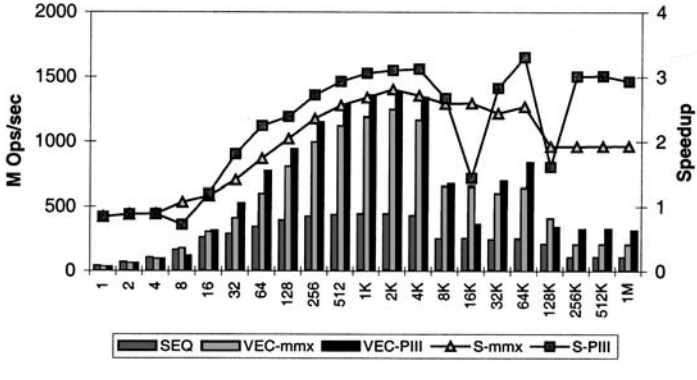

Fig. 5. Performance of IDOT on 500MHz Pentium® III Processor (aligned arrays).

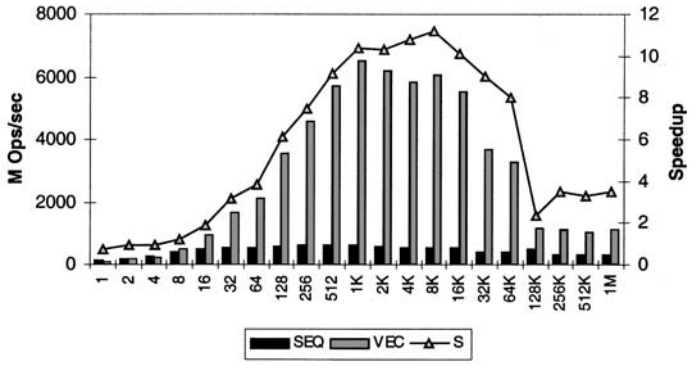

Fig. 6. Performance of IDOT on 1.5 GHz Pentium® 4 Processor (aligned arrays).

Processor features hardware prefetching,[9] no prefetching instructions are generated in this case. In Fig. 6, we show the performance of a serial (SEQ) and vector version (VEC) of IDOT and corresponding speedup (S) on a 1.5 GHz Pentium® 4 Processor for 16-byte aligned arrays of varying length. Again, a nice speedup that is sustained for most practical lengths is observed.

In the experiments considered so far, the peeling loop is never executed because all incoming arrays are aligned (the testing overhead is negligible though). The usefulness of an alignment strategy, however, is illustrated in Fig. 7. Here we show the speedup obtained on a 1.5 GHz Pentium® 4

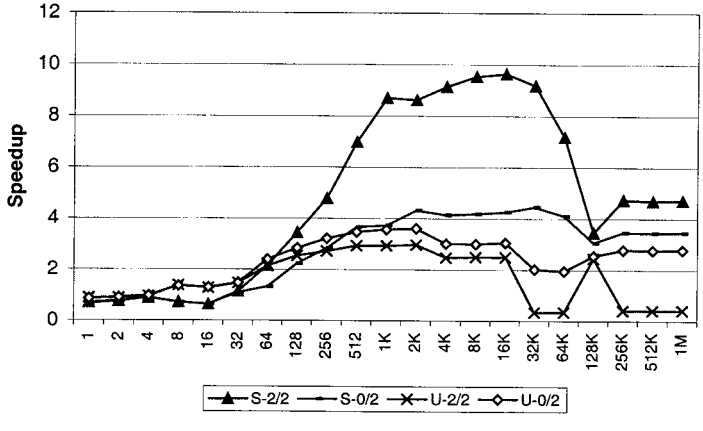

Fig. 7. Performance of IDOT on 1.5 GHz Pentium® 4 Processor (unaligned arrays).

Processor in case the incoming arrays p and q have an initial misalignment 2 with respect to a 16-byte boundary (S-2/2) and in case array p is aligned and q has an initial misalignment 2 with respect to a 16-byte boundary (S-0/2). In the first case, peeling off 7 iterations will make both access patterns aligned. In the second case, one of the access patterns remains unaligned. For comparison, we also show the speedup that would be obtained without an alignment strategy (U-2/2 and U-0/2), i.e., simply using unaligned data movement instructions for both arrays. Clearly, although peeling requires slightly longer arrays to become beneficial, combining intra-register vectorization with an alignment strategy eventually obtains higher performance.

4.2. Floating-Point Kernels

In this section, we study the performance impact of intra-register vectorization on four important floating-point Level 1 BLAS routines,[28, 29] namely 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 illustrated below with a C version of SDOT. We also show a straightforward way to vectorize this kernel in case both incoming arrays dx and dy are aligned at a 16-byte boundary. After the loop, the accumulator xmm0 contains 4 partial sums that must be added into the final scalar result d.

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

```
        pxor    xmm0, xmm0
Back:
        movaps  xmm1, [ebp+eax*4]
        mulps   xmm1, [ecx+eax*4]
        addps   xmm0, xmm1
        add     eax, 4
        cmp     eax, esi
        jl      Back
```

Because the alignments of the incoming arrays dx and dy are not known at compile-time, however, the Intel® C++/Fortran compiler uses alignment strategy (b) to deal with all possible alignments with respect to a 16-byte boundary. The kernel DDOT is vectorized similarly.

The kernels SAXPY and DAXPY perform single-precision or double-precision floating-point array updates, respectively, as shown below with a C implementation of SAXPY. A straightforward vector implementation in case both incoming arrays dx and dy are aligned is also given, where we assume that the scalar s has been scalar expanded into register xmm1.

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

```
Back:
     movaps xmm0, [ebp+eax*4]
     mulps  xmm0, xmm1
     addps  xmm0, [ecx+eax*4]
     movaps [ecx+eax*4], xmm0
     add    eax, 4
     cmp    eax, esi
     jl     Back
```

To deal with the unknown initial alignment of the incoming arrays, however, alignment strategy (b) is used. Automatic intra-register vectorization of this kernel also has to deal with a complication caused by the use of pointers. Without further points-to information, the compiler must conservatively assume that data dependences are carried by the loop, which prevents vectorization. As explained in Section 3.1, the Intel® C++/Fortran compiler solves this complication with dynamic data dependence testing, where an initial test for overlap decides between vector and serial execution of the loop.

In Fig. 8, we show the speedup obtained by automatic intra-register vectorization of the Level 1 BLAS routines on a 1.5 GHz Pentium® 4 Processor. For DDOT, we also show the speedup obtained by an assembly version (courtesy Henry Ou) that has been hand-optimized for the Pentium® 4 Processor. Here we see that despite the runtime overhead of dynamic data 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 performance). Furthermore, we see that the performance of compiler-generated code can

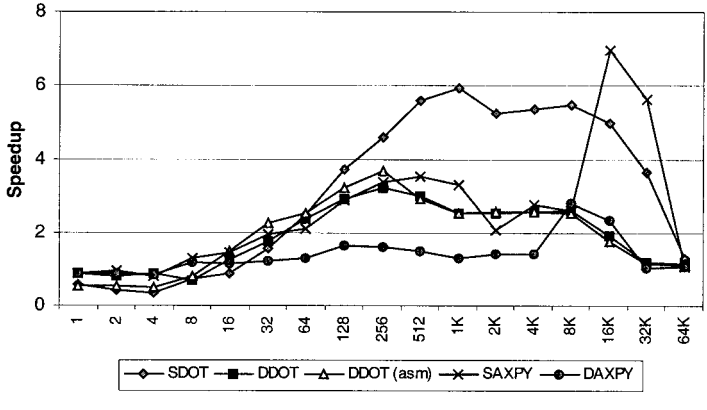

Fig. 8. Speedup of floating-point kernels on 1.5 GHz Pentium® 4 Processor (aligned arrays).

be quite close to the performance of hand-optimized assembly. Finally, note that, in order 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.[9] This clearly limits the speedup of DAXPY.

4.3. Benchmarks

Al Aburto's FLOPS benchmark consists of eight independent routines (flops1 through flops8), each of which tests the performance for a particular mix of floating-point instructions. Currently the main loops of all but routine flop2 are vectorized by the Intel® C++/Fortran compiler. In Fig. 9 we show the performance (in MFLOPS) obtained on a 2 GHz Pentium® 4 Processor for a serial version (SEQ) and vector version (VEC) of this benchmark in C. The corresponding speedups are also plotted in this figure. Here we see that five out of the eight routines substantially benefit from automatic intra-register vectorization. Only the performance of flops2 (not vectorized) and flops1 and flops7 (dominated by floating-point divisions) remains virtually unaffected.

The Callahan–Dongarra–Levine Fortran test suite (available at http://www.netlib.org/benchmark) consists of a variety of loops, intended to test the analysis capabilities of a vectorizing compiler. The Intel® C++/Fortran compiler currently vectorizes 59 out of the 135 test loops (the number goes up to 73 if elaborate instructions sequences may be used to implement non-unit stride references, but this method usually does not yield much speedup).

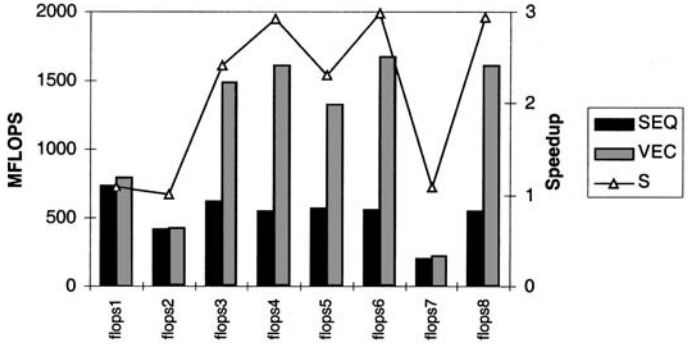

Fig. 9. Performance of FLOPS benchmark on 2 GHz Pentium® 4 Processor.

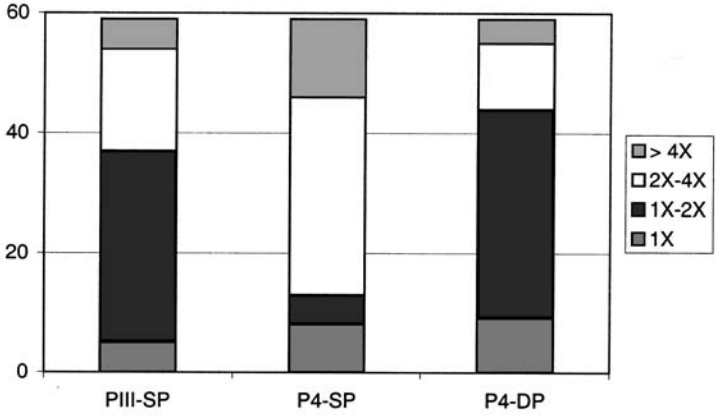

Fig. 10. Speedup classification for loops in Callahan–Dongarra–Levine Fortran test suite.

In Fig. 10, we summarize some performance results for a single-precision and double-precision floating-point version of this benchmark on a 500MHz Pentium® III Processor and 2 GHz Pentium® 4 Processor for an array length $N = 1000$. The table classifies the speedup of the 59 vector loops compared to serial execution of the loops in one of the following four categories: slight slowdown or no speedup (1×), moderate speedup (1×–2×), good speedup (2×–4×) and beyond (> 4×). Clearly, a majority of the loops exhibit a speedup of around 2.

The Linpack[28] benchmark (available at http://www.netlib.org/benchmark in both C and Fortran) reports the performance of a single-precision and double-precision linear equation solver that uses routines SGEFA/DGEFA and SGESL/DGESL for the factorization and solve phase, respectively. Most of the runtime of this benchmark results from repetitively calling the Level 1 BLAS routine SAXPY/DAXPY for different subcolumns of the coefficient matrix during factorization. Table I illustrates alignment properties of SAXPY's incoming arrays dy and dx during factorization of a single-precision 100×100 matrix (with leading dimensions LDA = 201 and LDA = 200 as defined in the benchmark). This table presents the number of times each incoming array is aligned with offset 0, 4, 8, or 12 with respect to a 16-byte boundary (denoted by x%16). The number of times both incoming arrays have the same or a different (mis)alignment is shown in the last two columns. This data suggests that the best performance for the single-precision floating-point version can be expected for LDA = 200, since in this case dynamic loop peeling can make both access patterns aligned.

Table I. Alignment Properties of Incoming Arrays of SAXPY (During Factorization of a 100 × 100 Matrix)

		0%16	4%16	8%16	12%16	SAME	DIFFERENT
LDA = 201	dx	25	2575	25	2524	1250	3899
	dv	1374	1250	1275	1250		
LDA = 200	dx	1324	1300	1275	1250	5149	0
	dv	1324	1300	1275	1250		

In Table II, we present the single-precision and double-precision performance (in MFLOPS) reported by the Linpack benchmark for serial (SEQ) and vector (VEC) execution on a 2 GHz Pentium® 4 Processor. The results are identical for Fortran and C, even though the C version requires dynamic data dependence testing to enable intra-register vectorization (In Fortran, the compiler may safely assume that aliases arising from parameter passing may not be modified). Note that the relatively poor performance of the serial single-precision version causes a rather unrealistic speedup. For the double-precision version, intra-register vectorization yields a speedup of around 2.

Finally, we report some performance results for the industry-standardized computationally intensive benchmark suite SPEC CPU2000 (see http://www.spec.org/). This suite consists of 14 floating-point and 12 integer benchmarks written in the languages C, C++ and Fortran, all derived from real-life applications. Here, we focus on how much intra-register vectorization can contribute to improving the performance of each of these benchmarks as a whole. Because the maximum obtainable speedup is limited by the fraction of execution time that is actually spent in vectorizable code (Amdahl's law), the resulting improvements will be much more moderate than for kernels where the main loop can be vectorized.

In Figs. 11 and 12, we show the speedup obtained on a 2 GHz Pentium® 4 Processor by full optimization (including advanced features like whole-program and profile-guided optimization, as well as optimizations specific to the Pentium® 4 Processor) relative to default optimization

Table II. Linpack Performance on 2 GHz Pentium® 4 Processor

	LDA = 201		LDA = 200	
MFLOPS	SEQ	VEC	SEQ	VEC
SP	64.4	965.1	64.3	1430.7
DP	525.1	1012.5	523.5	1011.0

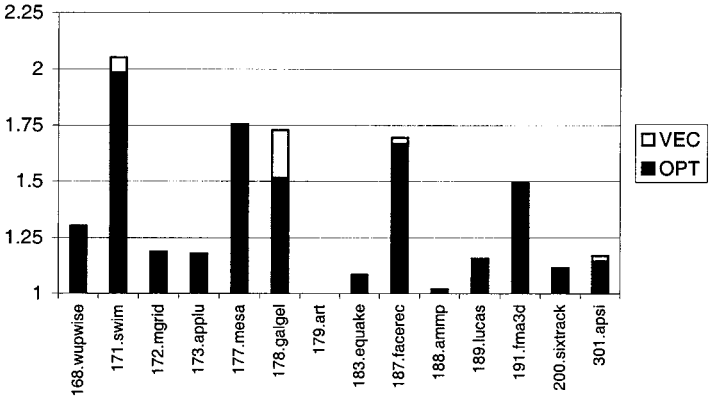

Fig. 11. Speedup for SPEC2000 (FP) on 2 GHz Pentium® 4 Processor.

(which yields generic code that executes on any Intel® Architecture). The contribution of intra-register vectorization is shown separately (VEC). For the floating-point benchmarks, four benchmarks benefit from automatic intra-register vectorization: an additional 20% for the computational fluid dynamics benchmark **178.galgel**, 6% for the weather prediction benchmark **171.swim**, and 2% for the benchmarks **187.facerec** and **301.apsi**. For the integer benchmarks, only one benchmark slightly benefits from intra-register vectorization: an additional 6% for **164.gzip** results from vectorization of loops that perform saturation arithmetic. For all other benchmarks, the Intel® C++/Fortran compiler currently fails to detect loops that benefit from automatic intra-register vectorization.

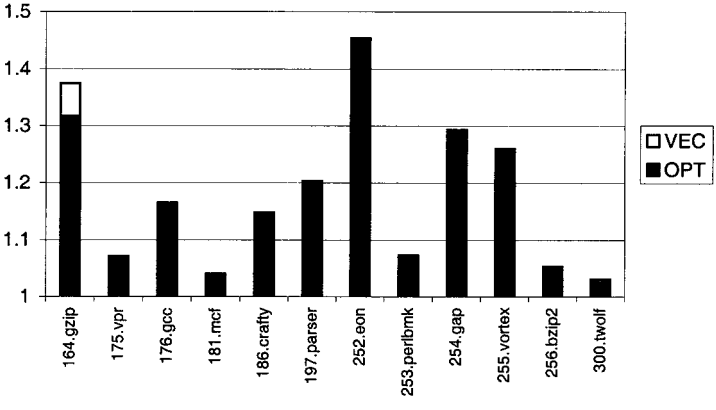

Fig. 12. Speedup for SPEC2000 (INT) on 2 GHz Pentium® 4 Processor.

5. RELATED WORK

To date, only a few production compilers are available that support automatic intra-register vectorization for the Intel® Architecture. The Portland Group (see http://www.pgroup.com/) offers the PGI® Workstation Fortran/C/C++ compilers that support automatic usage of SSE for the Pentium® III Processor. The company Codeplay™ (see http://www.codeplay.com/) offers the C compiler VectorC that automatically vectorizes standard C code to take advantage of the latest features of—amongst others—the Pentium® III Processor and Pentium® 4 Processor. Although a full comparison with these compilers would be beyond the scope of this paper, we have observed the following differences. The compilers of the Portland Group vectorize loops mostly by replacing frequently used operations into calls to optimized library routines for these operations. For example, in the Callahan–Dongarra–Levine test suite discussed earlier, the version 3.2-4 Fortran compiler pgf77—fast—Mvect=sse reports 24 replacements into library calls and 8 direct conversions into SSE instructions. VectorC focuses on replacing operations on consecutive data items (like members in a structure or elements in an array) into SIMD instructions. Loop unrolling is used to line up the number of operations with the vector length. These compilers currently do not seem to perform dynamic data dependence testing or any of the alignment optimizations discussed in this paper.

Examples of research compilers are the Scc compiler for the SWARC (SIMD-within-a-register) language,[30] which is a C-like language that provides a portable programming model for the SIMD extensions of a variety of processors, and the SLP (superword-level-parallelism) compiler,[31] which targets the detection of parallelism in basic blocks rather than loops. Other interesting work in progress is presented in Ref. 32, which describes a preprocessor for multimedia instructions that can be easily retargeted to deal with different multimedia instructions sets that vary frequently from one processor family to another.

6. CONCLUSIONS

Recent extensions to the Intel® Architecture have introduced the SIMD (single-instruction-multiple-data) technique as a way to enhance execution bandwidth in mainstream computing. Both the MMX™ technology and SSE/SSE2 (Streaming-SIMD-Extensions) support operations on packed data elements, which are relatively short vectors residing in registers or memory. To distinguish the conversion of serial code into a form that utilizes these extensions from other approaches to exploit data parallelism

(such as data pipelining in vector processors or replication of processing elements in massively parallel SIMD supercomputers), we referred to this conversion as **intra-register vectorization**. In this paper we presented an overview of the intra-register vectorization methods used by the Intel® C++/Fortran compiler. Obvious advantages of letting a compiler do the conversion are that the task of the programmer is greatly simplified and that existing software can be recompiled with relatively little effort to take full advantage of SIMD extension to the Intel® Architecture, possibly parameterized with certain peculiarities of the target architecture.

The Intel® C++/Fortran follows the standard approach to vectorization, where statements in an innermost loop are reordered and possibly distributed out according to the data dependence graph for the loop. The mapping of scalar operations to equivalent SIMD instructions is usually straightforward and we have specifically discussed how loops containing loop invariant memory references, type conversions, induction variables, reductions, saturation arithmetic, and conditional statements can be implemented using the MMX™ technology or SSE/SSE2. Furthermore, because the alignment of memory references can have a substantial impact on performance, several alignment optimizations have been incorporated in the compiler. Dynamic data dependence testing is used to allow the compiler to proceed with vectorization in situations where analysis has failed to prove independence statically.

The results of a number of experiments have been included in the paper to validate the effectiveness of the intra-register vectorization methods. For kernels and small computationally intensive applications (such as a linear solver), automatic intra-register vectorization can boost the performance substantially. For larger applications, where speedup is bound by the fraction of execution time actually spent in vectorizable code, more moderate improvements are obtained. The Intel® compiler also performed well in a qualitative experiment to test the analysis capabilities of the compiler: vector code can be generated for 73 loops in a test suite of 135 loops.

Future research will focus on increasing the effectiveness of automatic intra-register vectorization for real-life applications and the combination of this technique with automatic parallelization for shared-memory multiprocessors.[33] More information on high-performance compilers for the Intel® Architecture can be found at the website http://developer.intel.com/software/products/.

ACKNOWLEDGMENTS

The authors would like to thank other members of the Intel® C++/Fortran compiler team for their contributions. In particular, we would like

to thank Zia Ansari, Michael Gerlek, Michael Julier, David Kreitzer, Andrey Naraikin, Dale Schouten, Maxim Shutov, Kevin Smith, and German Voronov.

REFERENCES

1. Michael J. Flynn, *Computer Architecture*, Jones and Bartlett Publishers, Boston, Massachusetts (1995).
2. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, San Mateo, California (1990).
3. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to Parallel Programming*, The Benjamin/Cummings Publishing Company, Redwood City, California (1994).
4. Dezsö Sima, Terence Fountain, and Péter Kacsuk, *Advanced Computer Architectures—A Design Space Approach*, Addison-Wesley, Harlow England (1997).
5. R. M. Russel, The CRAY-1 Processor System, *Comm. ACM* **21**(1):63–72 (January 1978).
6. T. Blank, The Maspar MP-1 Architecture, *Proc. IEEE Compcon Spring* (February 1990).
7. David Bistry *et al.*, *The Complete Guide to MMX™ Technology*, McGraw-Hill, New York (1997).
8. Intel Corporation, *Intel Architecture MMX™ Technology—Programmer's Reference Manual*, Intel Corporation, Order No. 243007-003, available at http://developer.intel.com (1997).
9. 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/.
10. Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, available at http://developer.intel.com/ (2001).
11. 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).
12. David J. Kuck, *The Structure of Computers and Computations*, John Wiley and Sons, New York (1978), Vol. 1.
13. John M. Levesque and Joel W. Williamson, *A Guidebook to Fortran on Supercomputers*, Academic Press, San Diego (1991).
14. Constantine D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer, Boston (1988).
15. Michael J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, California (1996).
16. Hans Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York (1990).
17. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley (1986).
18. Andrew Appel, *Modern Compiler Implementation in C*, Cambridge University Press (1998).
19. Utpal Banerjee. *Dependence Analysis*, Kluwer, Boston, 1997. A Book Series on Loop Transformations for Restructuring Compilers.
20. Michael Burke and Ron Cytron, Interprocedural dependence analysis and parallelization, *Proceedings of the Symposium on Compiler Construction*, pp. 162–175 (1986).
21. C. N. Fisher and R. J. LeBlanc, *Crafting a Compiler with C*, Benjamin-Cummings, Menlo Park, California (1991).

22. Steven S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman Publishers (1997).
23. George B. Dantzig and B. Curtis Eaves, Fourier–Motzkin Elimination and Its Dual, *J. Comb. Theory* **14**:288–297 (1973).
24. Alexander Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons, Chichester, England (1986).
25. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1988).
26. Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian, *Automatic Detection of Saturation and Clipping Idioms for the Intel® Architecture*, manuscript in preparation (2001).
27. Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian, Experiments with automatic vectorization for the Pentium® 4 Processor, *Proceedings of the 9th Workshop on Compilers for Parallel Computers*, pp. 1–10 (June 2001).
28. 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).
29. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic Linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software* **5**:308–323 (1979).
30. 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).
31. Samuel Larsen and Saman Amarasinghe, Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *Proceeding of the SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, B.C. (June 2000).
32. Gilles Pokam, Julien Simonnet, and François Bodin, A Retargetable Preprocessor for Multimedia Instructions, *Proceedings of the 9th Workshop on Compilers for Parallel Computers*, pp. 291–301 (June 2001).
33. Aart J. C. Bik, Milind Girkar, Paul M. Grey, 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/.