Efficient Exploitation of Parallelism on Pentium® III and Pentium® 4 Processor-Based Systems

Aart Bik, Microcomputer Software Laboratories, Intel Corp.
Milind Girkar, Microcomputer Software Laboratories, Intel Corp.
Paul Grey, Microcomputer Software Laboratories, Intel Corp.
Xinmin Tian, Microcomputer Software Laboratories, Intel Corp.

Index words: compiler optimization, parallelization, vectorization, SIMD, multithreading

ABSTRACT

Systems based on the Pentium® III and Pentium® 4 processors enable the exploitation of parallelism at a fine- and medium-grained level. Dual- and quad-processor systems, for example, enable the exploitation of medium-grained parallelism by using multithreaded code that takes advantage of multiple control and arithmetic logic units. Streaming Single-Instruction-Multiple-Data (SIMD) extensions, on the other hand, enable the exploitation of fine-grained SIMD parallelism by vectorizing loops that perform a single operation on multiple elements in a data set. This paper provides a high-level overview of the automatic parallelization and vectorization methods used by the Intel® C++/Fortran compiler developed at the Microcomputer Software Labs.

INTRODUCTION

The Pentium III and Pentium 4 processors are designed to boost application performance and to provide performance scalability. The rich features of the Intel® microprocessors, such as the streaming SIMD extensions [9,10], enable compilers to exploit fine-grained parallelism by vectorizing loops that perform a single operation on multiple elements in a data set. The performance of the majority of scientific, engineering, and multimedia applications with characteristics such as inherent parallelism, a data independent control flow, regular and re-occurring memory access patterns, and localized re-occurring operations performed on the data can be improved by taking advantage of the streaming SIMD extensions. Dual- and quad-processor systems based on the 32-bit Intel® architecture provide opportunities for the compiler to exploit medium-grained parallelism by generating multithreaded code that uses multiple control and arithmetic logic units.

In this paper, we present the high-level software architecture of the automatic parallelization and vectorization methods used by the Intel C++/Fortran compiler developed at the Microcomputer Software Labs. We describe the static and dynamic analysis technologies implemented to enable the efficient generation of parallel code. We follow this with a description of multithreaded and vector code generation. A number of optimization technologies, such as alignment optimizations, advanced instruction selection, multi-entry threading technique, and Profile-Guided-Optimization (PGO) of parallel code, are also presented. We also discuss the results of experiments with automatic vectorization and parallelization on systems based on the Pentium III and Pentium 4 processors.

COMPILER ARCHITECTURE OVERVIEW

The approach taken by the Intel C++/Fortran compiler to exploit implicit parallelism in serial code is organized into three stages: program analysis, program restructuring, and parallel code generation.

Program Analysis

Program analysis performs a control flow, data flow, and data dependence analysis [1,3,4,11,12] to provide the compiler with useful information on where implicit parallelism in the input 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 [7].

Program Restructuring

Program restructuring focuses on converting the input program into a form that is more amenable to

parallelization. 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 tests to increase the opportunities for exploiting implicit parallelism in a program. An example of this is given below.

```
void init(char *p, char *q) {
    int i;
    for (i = 0; i <= 255; i++) p[i] = q[i];
}
```

Without any further information, the compiler must conservatively assume that the two pointers could refer to overlapping regions in memory. Conversion into multi-version code, however, yields a fully data-independent loop in the true branch that can be optimized accordingly.

```
void init(char *p, char *q) {
  int i;
  if (p+255 < q || p > q+255)
    for (i = 0; i <= 255; i++) p[i] = q[i]; /* dependence free */
  else
    for (i = 0; i <= 255; 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 [1,3]), loop transformations (such as loop interchanging or loop distribution [11,12]), and idiom recognition (such as the detection of reductions or other operations). An example of the latter category is shown below, where converting an if-statement into a "MAX"-operator makes the loop more amenable for analysis and, eventually, parallelization.

```
for (i = 0; i < N; i++) {                for (i = 0; i < N; i++) {
   if (a[i] > x)  x = a[i];       ◊        x = MAX(a[i], x);
}                                        }
```

Parallel Code Generation

Finally, parallel code generation consists of converting serial code into semantically equivalent multithreaded code or SIMD instructions. Both these conversions are outlined in the next sections. An in-depth presentation of vectorization is given in [5].

AUTOMATIC PARALLELIZATION

Automatic parallelization is a promising technique that can take advantage of shared-memory multiprocessors based on the Pentium III and Pentium 4 processors.

These systems can potentially deliver near supercomputer performance to mainstream computing. On a multiprocessor system, however, parallelizing inner loops usually does not provide sufficient granularity of parallelism. Thus, our focus for automatic parallelization is to exploit medium-grained parallelism to utilize a multiprocessor effectively. In this section, we describe the parallelization methods used by the Intel C++/Fortran compiler for automatic multithreaded code generation.

Finding Parallel Loops

Finding effective parallelism is one of the critical steps in generating efficient multithreaded code [6,8,11,12]. Based on the control flow graph, the data flow graph and the symbol table, the loop analyzer takes the following steps:

- Finds all loops within the serial code and builds a loop hierarchy structure. It fills up loop parameters such as trip count, lower bound, upper bound, and pre-header.
- Performs data dependence analysis to classify loops. Loops without loop-carried data dependencies are marked as loops that can be made parallel.
- Performs static or dynamic granularity estimation for each loop that can be made parallel. Multithreaded code for a parallel loop will be generated if and only if parallelization of the loop is profitable.

An example of the optimization is shown below.

```
for (k=0; k < 1000; k++) {
     x[k] = k;
     w    = x[k];
     y[k] = w + x[k];
}
```

Parallel loop detection marks this loop as follows.

```
parallel for (k=0; k < 1000; k++) {
      private (k, w), shared (x, y)
      x[k] = k;
      w    = x[k];
      y[k] = w + x[k];
}
```

In this example, the loop is marked as a loop that can be made parallel, and the variables "k" and "w" are marked as **private**. The arrays "x" and "y" are marked as **shared**. In the next section, we discuss variable classification based on liveness analysis.

Variable Classification

Liveness analysis [1,3] is well known and used in many optimizations and transformations. We use liveness analysis to classify the variables in the lexical extent of a loop that can be made parallel.

The **private**, **firstprivate**, and **lastprivate** attributes of variables direct the multithreaded code generator to implement privatization accordingly.

The **shared** attribute of a variable tells the multithreaded code generator to generate code that shares the memory location of this variable amongst multiple threads.

The following compilation rules are used to classify all variables referenced in a parallel loop:

1. A variable is marked **private** if and only if it is not live-in and not live-out on the current loop.
2. A variable is marked **firstprivate** if and only if it is live-in and not live-out on the current loop.

3. A variable is marked **lastprivate** if and only if it is live-out and not live-in on the current loop.
4. A variable is marked **shared** if and only if it is live-in and live-out on the current loop.

For the following example, the loop can be made parallel. Liveness analysis yields var-set = {a, b, c, k, n, x}, live-in-set = {a, b, c, n}, and live-out-set = {a, c, x}.

```
int foo(int b, int n, float c[]) {
    int   x = 101, k, a = 10;
    for (k=0, k < n; k++) {
        x = 5;
        c[k] =  x + a – b * k
    }
    return (a + x + c[0]);
}
```

Using compilation rules 1-4, variables "n" and "b" are marked **firstprivate**. Variables "a" and "c" are marked **shared**. Variable "x" is marked **lastprivate**. Variable "k" is a special form of a **private** variable: it is an induction variable. The data-race condition introduced by such variables is removed by induction variable privatization.

Static Granularity Estimation

Parallelizing a loop can result in slower execution if the overhead of dispatching/scheduling threads and sharing resources is significant compared to the total workload performed by the loop. The Intel C++/Fortran compiler handles this by examining all the operations in the loop body, estimating the grain-size per loop iteration on the targeted microarchitecture, and multiplying this by the loop trip count to arrive at an estimate of the total workload of the loop.

For loops with known trip counts, this value is compared, at compile time, to an experimentally determined profitable workload threshold to see if the loop should be multithreaded. Loops with a workload exceeding this profitable workload threshold will normally speed up when executed in parallel threads. For loops with unknown trip counts, the workload is expressed as a function of the trip count, and the compiler generates code to dynamically evaluate this expression to determine whether the loop should be executed with multiple threads.

Note that this solution avoids all dispatching/scheduling overhead and sharing of resources, if multithreaded execution is not profitable.

For the following example, the compiler generates an expression "(upper - lower) * grain-size" to compute the workload at runtime, based on the lower and upper bound and estimated grain-size.

```
void  foo(int lower, int upper) {
        int i;
        for (i=lower; i<upper; i++) {
          /* grain-size (in units of ops) */
        }
}
```

The granularity estimation has the following form.

```
trip_count  = upper - lower;
workload   =  trip_count * grain-size;

if (workload > (profit_probability *
                PROFIT_WORKLOAD_THRESHOLD) {
    /* multithreaded execution of the loop */
}
else {
    /* serial execution of the loop */
}
```

The profitable workload threshold (expressed in units of ops) is a global constant applicable to all loops. The threshold comparison can be modified with a command line option that sets the probability of profitable parallel execution ("profit_probability"). The workload is then compared to the experimentally determined profitable workload threshold multiplied by this probability. The value "0.0" causes the loop to be always executed as a multithreaded loop, whereas the value "1.0" causes multithreading to be used only if the workload exceeds the profitable workload threshold. The user can use any intermediate value to cause multithreaded execution of loops with low workloads that may still benefit from being made parallel.

Profile-Guided Granularity Estimation

Beyond the static granularity estimation, in the PGO mode of our compiler, we have implemented profile-guided granularity estimation to evaluate the workload, based on the execution count of basic blocks and branch probability. It is well known that compilers are often able to generate better code with the knowledge of likely execution paths. It is even more important for a parallelizing compiler to have the knowledge of the most frequently executed regions in a program, in order to determine if generating multithreaded code is profitable or not. Suppose that for the following code sample, we have the train data set "lower = 0" and "upper = 100." The profiler computes a "branch-taken" probability of "0.98" on the true branch and "0.02" on the false branch. The execution count of the loop pre-header (viz "i = lower") is "1", and the execution count of the loop header is "100."

```
void  foo(int lower, int upper) {
        int i;
        for (i=lower; i<upper; i++) {
          if (i>lower+1) {
              /* TRUE-grain-size (in units of ops) */
          }
          else {
              /* FALSE-grain-size (in units of ops) */
          }
        }
}
```

When these gathered execution measurements are fed back into the second pass of PGO compilation, the compiler compares "100 * (TRUE-grain-size * 0.98 + FALSE-grain-size * 0.02)" with the profitable workload threshold at compile time. Multithreaded code will not be generated if the comparison shows that parallelization is

not profitable. If, for example, the expression "TRUE-grain-size" is very small, PGO may avoid the slowdown introduced by parallelization.

MULTI-ENTRY THREADING TECHNIQUE

The conventional technology for generating multithreaded code is to generate an independent subroutine for each parallel loop. This is known as the *outlining* technology [6]. In contrast to this conventional technology, we propose a new technology called the *multi-entry threading* technique, which introduces three new concepts in the control flow graph: T-entry (threaded-entry), T-ret (threaded-return), and T-region (threaded-code-block). The ideas behind the new technology are as follows:

- The T-entry node contains the data environment for each thread that is necessary to build communication between the *invoker* (master thread) and the *invokee* (worker thread).
- The T-ret node informs the multithreaded runtime system about termination of the thread.
- A T-region is defined by a [T-entry, T-ret] pair and is kept inlined in the user-defined subroutine.
- Within a *single* user-defined subroutine, *multiple* [T-entry, T-ret] pairs are permitted to represent multiple T-regions.
- The [T-entry, T-ret] pairs can be nested (e.g., [T-entry, [T-entry, T-ret], T-ret]) to represent nested parallelism.

The main feature of the *multi-entry threading* technique is to keep all newly generated T-regions for parallel loops inlined in the same user-defined subroutine, without splitting them into independent subroutines. This technique provides subsequent compiler phases with more potential to optimize the code.

The following is an example of multithreaded code generation using the *multi-entry threading* technique.

```
float z[10000], w[10000];
void  foo(void) {
   int k, m, x[5000], y[5000];
   … …
   for (k=0;  k<5000; k++) {
      x[k] = x[k] + y[k] ;
   }
    for (m=0; m<10000; m++) {
      z[m] = z[m] * w[m];
   }
… …
}
```

There are two parallelizable loops in the subroutine "foo." The variables "k" and "m" are marked as **private** induction variables; the arrays "x", "y", "z", and "w" are marked as **shared**. The resulting multithreaded code is illustrated below. The Intel C++/Fortran compiler has adopted the KAI* Guide runtime library for thread creation and management.

```
float z[10000], w[10000];
void  foo(void)
{   int k, m, x[5000], y[5000];
     … …
    __kmpc_fork_call(loc, 2, T-entry(_foo_ploop_0), x, y)
    goto L1:
    T-entry _foo_ploop_0(loc, tid, x[], y[]) {
       lower_k = 0;
       upper_k = 5000;
      __kmpc_for_static_init(loc, tid, s, &lower_k, &upper_k, …);
      for (par_k=lower_k,  par_k<=upper_k; par_k++)  {
              x[par_k] = x[par_k] + y[par_k] ;
      }
     __kmpc_for_static_fini(loc, tid);
     T-ret;
   }
L1:
   __kmpc_fork_call(loc, 0, T-entry(_foo_ploop_1));
   goto L2:
   T-entry _foo_ploop_1(loc, tid) {
      lower_m  = 0;
      upper_m = 10000;
      __kmpc_for_static_init(loc, tid, s, &lower_m, &upper_m, …);
      for (par_m=lower_m; par_m<=upper_m; par_m++)  {
          z[par_m] = z[par_m] * w[par_m];
      }
      __kmpc_for_static_fini(loc, tid);
      T-ret;
   }
 L2:
   … …
}
```

The multithreaded code generator inserts the thread invocation call "__kmpc_fork_call" with the T-entry point and data environment (e.g., line number "loc") for each loop. This call into the KAI runtime library will fork a number of threads that execute the iterations of the loop in parallel.

The serial loops are converted to multithreaded code by localizing the loop lower and upper bound, and by privatizing the induction variable. Finally, multithreading runtime initialization and synchronization code is generated for each T-region defined by a [T-entry, T-ret] pair. The library call "__kmpc_for_static_init" computes the localized loop lower bound, upper bound, and stride for each thread according to a scheduling policy. The library call "__kmpc_for_static_fini" informs the runtime system that the current thread has completed one loop chunk.

Compared with the existing *outlining* technology, there are three advantages to the *multi-entry threading* technique for generating efficient multithreaded code:

- The *multi-entry threading* technique does not create separate compilation units for parallel loops, and the required program transformations are very natural and simple. It reduces the complexity of handling separate routines in the optimizer.
- _____

* Kuck and Associates, Inc., an Intel Corporation.

- All generated T-regions for parallel loops are kept inlined in the same compilation unit. This minimizes the impact on other optimizations such as constant propagation, scalar replacement, loop transformation, common expression elimination, and interprocedural optimization.
- Besides global and file-scope static variables, the memory location of a local shared static variable can be accessed naturally by multiple threads without passing an argument on T-entry, since the generated multithreaded code is kept inlined in the user-defined subroutine.

AUTOMATIC VECTORIZATION

The Pentium III and Pentium 4 processors feature a rich set of SIMD instructions on packed integers and floating-point numbers that can be used to boost the performance of loops that perform a single operation on different elements in a data set.

The Pentium III processor introduced the 128-bit streaming SIMD extensions [10], supporting floating-point operations on 4 single-precision floating-point numbers and some more instructions for the 64-bit integer MMX™ technology. The Pentium 4 processor further extended this support for floating-point operations on two double-precision floating-point numbers and widened the integer MMX technology into 128-bit [9]. Because a single instruction processes multiple data elements in parallel, all these extensions are very useful to utilize SIMD parallelism in numerical and multimedia applications.

The Intel C++/Fortran compiler follows the standard approach to the vectorization of inner loops [2,11,12]. First, 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 SIMD instructions.

Consider as an example the loop shown below.

```
double a[100], b[100], c[100];  /* assume arrays start at
                                   16-byte boundaries */
…
for (i = 0; i < 100; i++) {
    a[i] = b[i] - c[i];
}
```

Since there are no data dependencies in this loop, the Intel C++/Fortran compiler translates this loop into the following SIMD instructions for the Pentium 4 processor. Note that because double elements are eight bytes wide and the vector loop processes two elements in each iteration, the upper bound and stride for the offsets into the arrays are 100x8=800 and 2x8=16, respectively.

```
SUB:
   movapd   xmm0, b[ecx]      ;  load      2 DP FP numbers
   subpd    xmm0, c[ecx]      ;  subtract 2 DP FP numbers
   movapd   a[ecx], xmm0      ;  store     2 DP FP numbers
   add      ecx, 16
   cmp      ecx, 800
   jl       SUB               ;  looping logic
```

For loops with a trip count that cannot be evenly divided by the vector length, a cleanup loop is used to execute any remaining iteration serially. In the PGO mode, a profile-guided estimation of statically unknown trip counts is used to determine whether vectorization is actually worthwhile.

Alignment Optimizations

In the previous example, the aligned data movement instruction "movapd" can be used because the compiler has aligned the first elements of the three arrays at a 16-byte boundary. For unaligned (or unknown) access patterns, the compiler must use unaligned data movement instructions, like "movupd." Because there can be a substantial performance penalty for unaligned data references, the Intel C++/Fortran compiler has at its disposal a variety of *static* and *dynamic* alignment optimizations.

In the loop shown below, for instance, the compiler will statically peel off one iteration to align all access patterns.

```
double a[100], b[100];   /* 16-byte aligned */
…                                          a[1] = b[1] - 1;
for (i = 1; i < 100; i++) {                for (i = 2; i < 100; i++) {
  a[i] = b[i] – 1;                  ◊          a[i] = b[i] – 1;
}                                          }
```

For cases where the alignment of data structures cannot be determined at compile time, the compiler uses a dynamic loop peeling alignment strategy in which, at runtime, first a few iterations are executed serially until one or several access patterns become 16-byte aligned.

Consider, for instance, a simple initialization loop.

```
char *p = …;
…
for (i = 0; i < 100; i++)  p[i] = 0;
```

Without any further points-to information for "p", the compiler would have to conservatively assume that the access pattern is unaligned. Dynamically peeling off some iterations based on the starting address of the array, can, nevertheless, enforce aligned references.

```
peel = p & 0x0f;
if (peel != 0) {
   peel = 16 - peel;
   for (i = 0; i < peel; i++) p[i] = 0;
}
/* aligned access pattern */
for (i = peel; i < 100; i++) p[i] = 0;
```

Reductions

Although reductions give rise to data dependence cycles, such idioms can be translated into SIMD instructions that compute partial results in parallel. Consider, for example, the accumulation that occurs in the DDOT kernel.

```
double d = 0.0;
for (i = 0; i < N; i++) {
    d += a[i] * b[i];
}
```

This reduction can be implemented as follows. Note that in this fragment, the size of double elements is accounted for in the effective address computations. As stated before, serial cleanup code is generated after the vector loop to deal with odd values of N.

```
      xorpd     xmm1, xmm1       ;  reset accumulator
DDOT:
      movapd    xmm0, a[ecx*8]   ;  load,
      mulpd     xmm0, b[ecx*8]   ;    multiply,
      addpd     xmm1, xmm0       ;     and accumulate
      add       ecx, 2           ;      2 DP FP numbers
      cmp       ecx, N
      jl        DDOT             ;  looping logic

      movapd    xmm0, xmm1       ;  postlude:
      unpckhpd  xmm0, xmm1       ;   add 2 partial
      addpd     xmm1, xmm0       ;      results into
      movsd     [esp], xmm1      ;           scalar d
```

Other reductions (based on any of the operators "+", "-", "*", "&", "|", "MIN" or "MAX") are handled similarly.

Short Vector Mathematical Library

The Intel C++/Fortran compiler comes with a Short Vector Mathematical Library (SVML), developed at Intel® Nizhny Novgorad Labs in Russia (INNL), that provides efficient software implementations for computing (inverse) trigonometric, (inverse) hyperbolic, exponential, and logarithmic functions on (sub)arrays. This library provides a clean interface to operate on packed floating-point numbers.

The library allows the vectorization of loops that contain any of these mathematical functions. Consider, for example, the following loop.

```
for (i = 0; i < 100; i++) {
    a[i] = sin(b[i]) + c[i];
}
```

Using the SVML allows the compiler to proceed with vectorization of this loop as follows (an implementation that passes arguments and results in the xmm-registers is planned as well).

```
SIN:
      lea       ecx, b[esi]
      lea       eax, [esp+16]
      mov       [esp], ecx       ;  define input address
      mov       [esp+4], eax     ;  define output address
      call      _vmldSin2        ;  call SVML
      movapd    xmm0, [esp+16]   ;  read result
      addpd     xmm0, c[esi]
      movapd    a[esi], xmm0
      add       esi, 16
      cmp       esi, 800
      jl        SIN              ;  looping logic
```

Advanced Instruction Selection

Advanced instruction selection is used to vectorize certain frequently occurring operations that can be efficiently mapped onto the SIMD instructions of the Intel architecture. Consider, for example, the following loop (the suffix letter "u" denotes an unsigned constant).

```
unsigned char x[256];
…
for (i = 0; i < 256; i++)
    x[i] = (x[i] >= 20u) ? x[i] - 20u : 0u;
}
```

The Intel C++/Fortran compiler recognizes the saturation arithmetic done in this code fragment (if the result of the subtraction would be negative, the result is saturated to zero) and converts the serial loop into the following SIMD instructions that operate on 16 unsigned characters in each iteration.

```
      movdqa    xmm0, CONVEC  ;  load <20u,….,20u>
SAT:
      movdqa    xmm1, x[eax]
      psubusb   xmm1, xmm0       ;  perform  16 saturated
      movdqa    x[eax], xmm1     ;       subtractions
      add       eax, 16
      cmp       eax, 256
      jl        SAT              ;  looping logic
```

The compiler also carefully selects the instructions that are used to implement scalar expansions, certain type conversions, and non-unit stride references. In addition, the use of bit-masks supports the vectorization of singly nested conditional statements.

For a detailed presentation of all the vectorization methods used by the Intel C++/Fortran compiler, we must refer to [5].

EXPERIMENTAL RESULTS

In this section, we discuss the results of some experiments with automatic vectorization and parallelization. Consider, for instance, the following code that computes the product of a double-precision floating-point matrix and vector.

```
for (i = 0; i < n; i++) {
  double d = 0.0;
  for (j = 0; j < n; j++) {
    d += a[i][j] * y[j];
  }
  x[i] = d;
 }
```

In the graph shown in Figure 1, we present the speedup (uniprocessor vs. multiprocessor execution time) obtained by automatic parallelization of the outermost loop in this kernel on a dual 500MHz. Pentium III processor for varying matrix orders. In the same figure, we also show the speedup of serial vs. parallel execution obtained on a quad 550MHz. Pentium III processor. Speedups up to 3.2 and 1.6 are obtained for the quad and dual system, respectively.

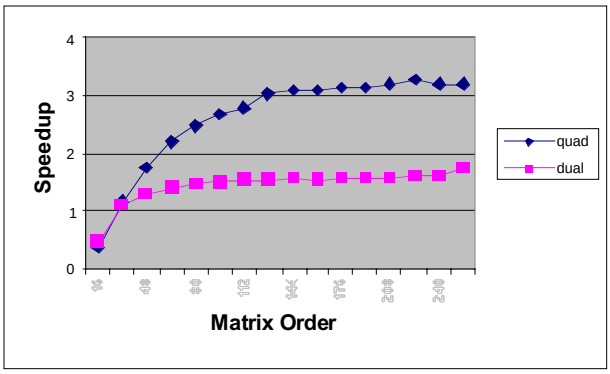

Figure 1: Speedup for matrix x vector on a dual and quad Pentium® III processor

As another example of automatic parallelization, consider LU-factorization without pivoting.

```
for (k = 0; k < n-1; k++) {
  for (i = k+1; i < n; i++) {
     a[i][k] = a[i][k] / a[k][k];
     for (j = k+1; j < n; j++)
        a[i][j] = a[i][j] - a[i][k] * a[k][j];
  }
}
```

In this fragment, loop-carried data dependencies prohibit parallelization of the outermost k-loop. The iterations of the i-loop, on the other hand, can be executed in parallel. In Figure 2, we show the corresponding speedup on a dual and quad shared-memory multiprocessor for varying matrix orders. Despite the fact that the outermost loop has to remain serial, speedups up to 1.3 and 2.6, respectively, are still obtained.

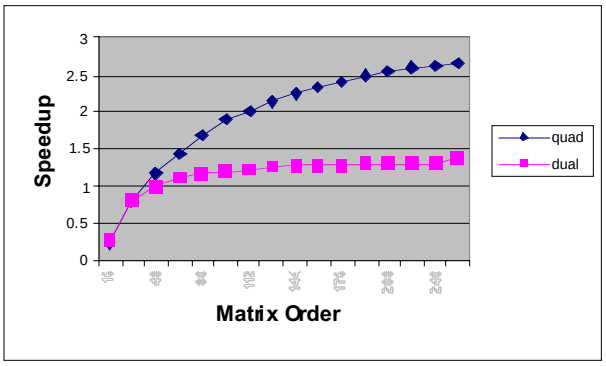

Figure 2: Speedup for LU-factorization on a dual and quad Pentium® III processor

In Figure 3, we show the speedup (serial vs. vector execution time) obtained on a 1.5GHz. Pentium 4 processor by automatic vectorization of a single-precision dot-product kernel (SDOT) and a double-precision dot-product kernel (DDOT) for array lengths ranging from 1 to 64K. For comparison, we also present the speedup obtained by a hand-coded assembly version of the latter kernel (ASM, courtesy Henry Ou). Execution times were obtained by running the kernel many times and dividing the total execution time accordingly, so that for data sizes that fit in the 256KB L1 cache, effectively "warm cache behavior" is measured.

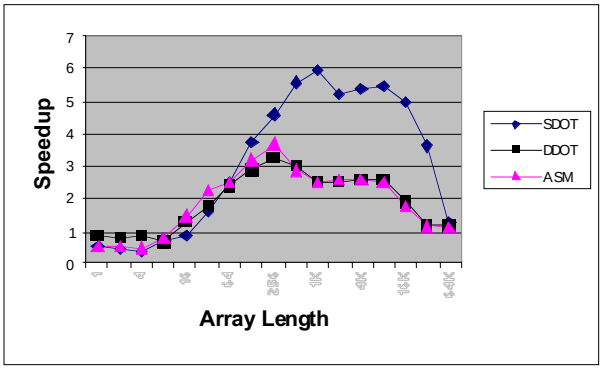

Figure 3: Speedup for dot-product on a Pentium® 4 processor

The performance of the SDOT and DDOT kernels observed after automatic vectorization (counting one floating-point addition and multiplication per iteration) exceeded 3.3 GFLOPS and 1.8 GFLOPS, respectively.

Automatic vectorization of a LINPACK benchmark (available at http://www.netlib.org.benchmark/) boosted the performance of solving a system of linear equations defined by a 100x100 double-precision matrix on a 1.5GHz. Pentium 4 processor from 582 MFLOPS to 700 MFLOPS.

In the last graph shown in Figure 4, we show the speedup obtained on a 1.5GHz. Pentium 4 processor by automatic vectorization of kernels of the form "x[i] = F(y[i])". The experiments are done for three different double-precision floating-point functions, supported by SVML, and array lengths varying from 1 to 256, with input sets consisting of uniformly distributed values in the range 0 through 2*_.

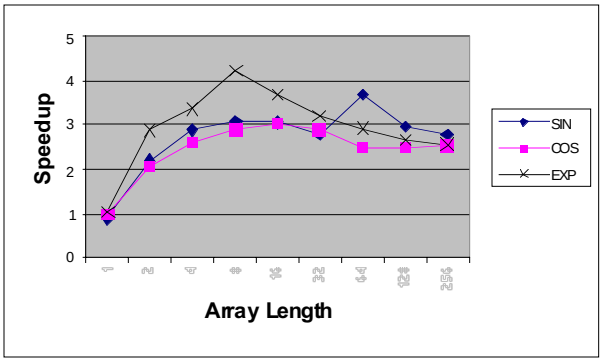

Figure 4: Speedup for math functions on a Pentium® 4 processor

DISCUSSION

The experiments reveal that the automatic detection of implicit parallelism in serial software can provide a very

portable way of effectively exploiting SIMD instructions or multiple CPUs on systems that are based on the Pentium III and Pentium 4 processors. Automatic parallelization of the outermost loop in the matrix times vector product starts to speed up for matrices with an order that exceeds 32 on both the dual and quad multiprocessor with an efficiency (Speedup / #processors x 100%) going up to over 80% for larger matrices. Likewise, automatic parallelization of the second outermost loop in an implementation of LU factorization, without pivoting, yields efficiencies of over 60%.

Automatic vectorization of the DDOT kernel yields speedup comparable to the speedup obtained by a hand-optimized assembly implementation. Combining vectorization with efficient software implementations of frequently used mathematical functions already exhibits speedup for arrays with a length of only 2. Another clear advantage of having a vector implementation of mathematical functions is that vectorization of a loop does not have to bail out in the presence of such function calls.

CONCLUSION

Explicitly exploiting parallelism in a program can be a cumbersome and error-prone task. It may require the use of inline assembly to generate the appropriate SIMD instructions or the use of a complicated threading library to take advantage of the computing power available on a multiprocessor. Although such explicit techniques can be extremely effective, they are not portable and greatly complicate program development and maintenance. An alternative approach is to let a compiler do (at least part of) the exploitation of fine- and medium-grained parallelism automatically. With this approach, the compiler analyzes a program that is written in a sequential language for implicit opportunities to exploit parallelism, and it generates code that takes advantage of this implicit parallelism.

In this paper, we provided a high-level overview of the automatic parallelization and vectorization methods used by the Intel C++/Fortran compiler developed at the Microcomputer Software Labs. We have shown that these methods can obtain good speedup on systems based on the Pentium III and Pentium 4 processors, without the need for any source code modifications. Hence, automatically exploiting 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 parallelism. Finally, the approach allows the automatic parallelization and vectorization of existing serial software, thereby avoiding the potentially huge investments that would be required to hand optimize this code.

More information on Intel's high-performance compilers can be found at

http://developer.intel.com/software/products/

ACKNOWLEDGMENTS

The authors thank the other members of the Proton team for their hard work to implement and test the Intel C++/Fortran compiler. We thank the compiler group at KAI for providing Guide, the multithreading runtime library, and INNL for providing SVML. Both these libraries are currently part of the Intel C++/Fortran compiler.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers—Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Boston, Massachusetts, 1986.

[2] John Randal Allen and Ken Kennedy, "Automatic Translation of Fortran Programs into 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, Cambridge, UK, 1998.

[4] Utpal Banerjee, *Dependence Analysis*, Kluwer Academic Publishers, Boston, Massachusetts, 1997.

[5] Aart Bik, Milind Girkar, Paul Grey, and Xinmin Tian, "An Auto-Vectorizing Compiler for the Intel® Architecture," *Submitted to the ACM Transactions on Programming Languages and Systems*, 2000.

[6] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors," *in Proceedings of CASCON: 76-89, Toronto, ON, November 12-14*, 1996.

[7] George B. Dantzig and B. Curtis, "Fourier-Motzkin Elimination and its Dual," *Journal of Combinatorial Theory: 14:288-297*, 1973.

[8] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. "Detecting Coarse-Grain Parallelism using an Interprocedural Parallelizing Compiler," *in Proceedings of Supercomputing, San Diego, California, December*, 1995.

[9] Intel Corporation, *Intel® Architecture Software Developer's Manual With Preliminary Willamette Architecture Information*, manual available at http://developer.intel.com/.

[10] Shreekant Thakkar and Tom Huff. "Internet Streaming SIMD Extensions," *IEEE Computer: 32:26-34*, 1999.

[11] Michael J. Wolfe, *High Performance Compilers for Parallel Computer*, Addison-Wesley Publishing Company, Redwood City, California, 1996.

[12] Hans Zima, *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, NY, 1990.

AUTHORS' BIOGRAPHIES

Aart Bik received his M.Sc. degree in Computer Science from Utrecht University, The Netherlands, in 1992, and his Ph.D. degree from Leiden University, The Netherlands, in 1996. In 1997, he did a post-doc at Indiana University, Bloomington, Indiana, where he conducted research in high-performance compilers for Java*. In 1998, he joined Intel Corporation where he is currently working in the vectorization and parallelization group of the Microcomputer Software Labs. His e-mail is aart.bik@intel.com

Milind Girkar received a B.Tech. from the Indian Institute of Technology, Mumbai, an M.Sc. degree from Vanderbilt University, and a Ph.D. degree from the University of Illinois at Urbana-Champaign, all in Computer Science. Currently, he manages the vectorization and parallelization group in Intel's Microcomputer Software Labs. Before joining Intel, he worked on a compiler for the UltraSPARC platform at Sun Microsystems. His e-mail is milind.girkar@intel.com

Paul Grey did his B.Sc. degree in Applied Physics at the University of the West Indies and his M.Sc. degree in Computer Engineering at the University of Southern California. Currently he is working in Intel's Microcomputer Software Labs, researching compiler optimizations for parallel computing. Before joining Intel, he worked on parallel compilers, parallel programming tools, and graphics system software at Kuck and Associates, Inc., Sun Microsystems, and Silicon Graphics. His research interests include optimizing compilers, advanced microarchitectures and parallel computer systems. His e-mail is paul.grey@intel.com

Xinmin Tian is currently working in the vectorization and parallelization group at Intel's Microcomputer Software Labs where he works on compiler parallelization and optimization. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from Tsinghua University. He was a postdoctoral researcher in the School of Computer Science at McGill University, Montreal. Before joining Intel, he worked on parallelizing compilers, code generation, and performance optimization at IBM. His e-mail is xinmin.tian@intel.com

---

*Other brands and names are the property of their respective owners.