



Intel[®] Technology Journal

Multi-Core Software

**Inside the Intel[®] 10.1 Compilers:
New Threadizer and New Vectorizer
for Intel[®] Core[™]2 Processors**

Inside the Intel[®] 10.1 Compilers: New Threadizer and New Vectorizer for Intel[®] Core[™]2 Processors

Xinmin Tian, Software and Solutions Group, Intel Corporation
Ernesto Su, Software and Solutions Group, Intel Corporation
David Kreitzer, Software and Solutions Group, Intel Corporation
Hideki Saito, Software and Solutions Group, Intel Corporation
Rakesh Krishnaiyer, Software and Solutions Group, Intel Corporation
Abhay Kanhere, Software and Solutions Group, Intel Corporation
John Ng, Software and Solutions Group, Intel Corporation
Chu-Cheow Lim, Mobility Group, Intel Corporation
Somnath Ghosh, Mobility Group, Intel Corporation

Index words: multi-core, optimizing compiler, threadization, vectorization, advanced optimization

ABSTRACT

The fast introduction of the Intel[®] Core[™]2 Duo and Quad processors to the mass market has drawn attention to threadization (a.k.a. parallelization) and vectorization of the existing code in many application domains. In fact, multi-core processor vendors are eager to enable their users to exploit various levels of parallelism in order to harness the additional compute resources of multi-core processors. The Intel[®] C++/Fortran compiler provides an essential tool for unleashing the power of Intel Core 2 Duo and Quad processors. This is accomplished by means of high-level loop optimizations and scalar optimizations to exploit multi-core processors and single-instruction-multiple-data (SIMD) instructions, combined with advanced code generation, that is built on an intimate knowledge of micro-architectural performance aspects.

In this paper we outline the design and implementation of a new threadizer and vectorizer inside the Intel[®] 10.1 compilers, and we also provide an overview of the enhanced high-level loop optimizations and the low-level code generation used to obtain higher performance on platforms based on Intel Core 2 Duo and Quad processors. Significant performance gains are shown using the SPEC CPU2006* suite running on a system configured with two Intel[®] quad-core processors.

INTRODUCTION

The aggressive delivery of Intel[®] multi-core processors to the mass computer market shows that, as the performance

improvements from continuously increasing clock frequencies start to taper off, other architectural advances that reduce latency or increase memory bandwidth are gaining importance [9]. In particular, since packaging densities are still growing, integrating multiple processors on a single die and using SIMD extensions are becoming more widespread [1]. The Intel Core 2 Duo and Quad processors are equipped with a rich set of micro-architectural and architectural features to boost performance:

- dual-core or quad-core on a single chip
- wider execution units for Streaming SIMD Extensions (SSE, SSE2, SSE3)
- a set of new instructions referred to as Supplemental Streaming SIMD Extensions 3 (SSSE3)
- advanced smart shared L2 cache among cores on the same chip

Due to the complexity of modern processors, compiler support has become an important part of obtaining higher performance. Most importantly, to assist programmers in leveraging all parallel capabilities of Intel's new processors, the Intel C++/Fortran compiler provides an essential tool for unleashing the power of Intel multi-core processors and SIMD instructions by means of high-level optimizations and advanced code generation.

The Intel compilers perform automatic optimizations of programs using threadization [10], vectorization [1, 2, 5], classical loop transformations (e.g., distribution, unrolling, interchange, fusion) [7, 11, 12], scalar optimizations such

as constant propagation, Partial Dead Store Elimination (PDSE), Partial Redundancy Elimination (PRE), copy propagation, Inter-Procedural Optimizations (IPO) [7], and advanced machine code generation techniques that together yield a significant performance gain compared to the default level of optimization. The contributions of the new threadizer and vectorizer are as follows:

- The new threadizer yields up to 4.63x speedup (with 8 cores) by exploiting thread-level parallelism from a serial program in the SPEC[®] CPU2006 benchmark suites. Overall, the auto-threadization delivers a 15.45% gain (geomean with 8 cores) for SPEC CFP2006 suite and a 12.17% gain (geomean with 8 cores) for SPEC CINT2006 suite.
- The new vectorizer yields up to 1.28x performance speedup by exploiting SIMD-type vector parallelism from a serial program in the SPEC CPU2006 suites. Overall, the auto-vectorization delivers a 5.11% gain (geomean) for SPEC CFP2006 suite and a 2.01% gain (geomean) for SPEC CINT2006 suite.

The rest of this paper is organized as follows. First, we provide some basics on the Intel[®] Core™ micro-architecture. Then, we discuss the design and implementation of the new threadizer and vectorizer, respectively, inside the Intel 10.1 compilers. Subsequently, we discuss the loop optimizations and enhancements made to support efficient threadization and vectorization. We also present an overview of advanced code generation for the Intel Core 2 Duo and Quad processors. Finally, we provide performance results using the SPEC CPU2006 industry-standard benchmark suite built with the Intel 10.1 C++ and FORTRAN compilers.

INTEL[®] CORE™ MICRO-ARCHITECTURE

Intel Core micro-architecture is the foundation for all new Intel[®] architecture-based desktop, mobile, and server multi-core processors. This state-of-the-art multi-core processor with optimized micro-architecture delivers a number of innovative features that have set new standards for energy-efficient performance. In this section we outline a few innovations relevant to this paper. A more detailed description can be found in the Intel[®] literature [4].

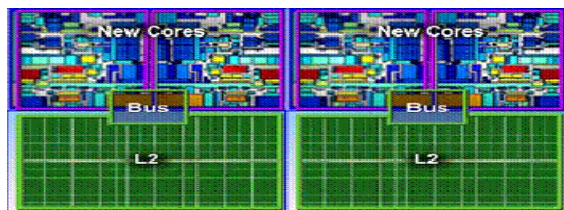


Figure 1: Quad-core processor schematic

Figure 1 shows a schematic of the Intel Core 2 Quad processor. Two independent cores with their own private L1 caches reside on a single die. Two shared Level 2 (L2) caches, referred to as the Intel[®] Advanced Smart Cache, work by sharing the L2 cache between cores so that data are stored in one place accessible by the cores. Sharing the L2 cache enables a core to dynamically use up to 100% of the available L2 cache, thus optimizing cache resources.

The quad-core processor is equipped with Intel[®] Smart Memory Access techniques that boost system performance by optimizing available data bandwidth from the memory subsystem and hiding the latency of memory accesses through two techniques: memory disambiguation and an instruction pointer-based prefetcher that fetches memory contents to the shared L2 cache and then into each private L1 cache before they are requested. The data prefetcher can detect strided memory access patterns to make accurate predictions about future load addresses.

Another key feature of Intel Core micro-architecture is the Intel[®] Advanced Digital Media Boost that can issue 128-bit SSE instructions with a *throughput* of one per clock cycle. Previous-generation Intel processors had a sustained throughput of one instruction per two clock cycles, typically one cycle for the lower 64 bits followed by another cycle for the upper 64 bits. By widening execution units to the full 128 bits, the Intel processor effectively doubles the performance of a series of 128-bit SSE instructions relative to previous-generation Intel processors. In addition, the *latency* of various individual 128-bit SSE instructions has been reduced, and SSSE3 has been added to extend the instruction set. As a result, more overall performance improvements can be expected from vectorization (i.e., transforming sequential code into SIMD instructions).

REVAMPING THE THREADIZER

In this section, we present our new threadizer framework that is highly integrated with our classical high-level loop optimizations, and we describe its main components. The strengths of the new threadizer include the following:

- A new Abstract Thread Representation (ATR), based on the concept of virtual threads, is designed to bridge the semantic gap between high-level representation and physical (hardware or OS) threads.
- Better interaction with other high-level loop-related optimizations gives better performance.
- The new threadizer is moved downstream to take advantage of scalar optimizations such as global constant propagation and Single-Static-Assignment (SSA) PRE, and some loop optimizations.
- A table-driven cost model simplifies maintenance and future extensibility.

- Effective runtime threadization control and multiple schedule types such as static, dynamic, guided, and runtime are supported.

The threadizer in the Intel compiler serves as a single module that covers different languages (C++ and Fortran), architectures (IA-32, Intel[®] 64, and IA-64), and operating systems (Microsoft Windows*, Linux*, and MacOS*).

Virtual Threads

Our new threadization framework is based on the concept of **virtual threads**. A virtual thread is an abstraction above physical threads provided by hardware threads or OS threads. Virtual threads can consist of arbitrary code blocks and have no nesting-level constraints as long as they obey the specified program execution order.

A virtual thread denoted as a quadruple $V(\alpha, s, e, d)$ corresponds to a thread with *instruction entry* s , *instruction exit* e , *data environment* d , and *thread id* α that are assigned at runtime. An important property of a virtual thread is its lexical nesting level, which is denoted as **depth**($V(\alpha, s, e, d)$). The depth is computed recursively as follows:

When $V(\alpha, s, e, d)$ represents a thread at the outer-most lexical nesting level of parallel constructs, we set its nesting level to **depth**($V(\alpha, s, e, d)$) = 0. When $V(\alpha, s, e, d)$ represents a thread at an inner lexical nesting level of parallel constructs, we set its nesting level to **depth**($V(\alpha, s, e, d)$) = **depth**(parent($V(\alpha, s, e, d)$)) + 1.

This lexical nesting-level property is not to be confused with the dynamic (runtime) nesting level of the physical threads supported by the threading runtime library. Another property of a virtual thread is its code block type (a loop, a region, a section, or a task) that can distinguish different threading semantics of a virtual thread and can guide the compiler to generate threaded code according to the definitions of the compiler-to-runtime interface. We say that a virtual thread is mapped to a physical thread (or a runtime thread) when the virtual thread is assigned a unique thread identifier α at runtime. Note that a virtual thread can be mapped to a team of physical threads for a parallel loop and region by assigning a unique thread identifier for each mapping.

Threadization Framework

Figure 2 outlines the new framework. The first two phases extract thread-level parallelism within different program scopes to construct virtual threads. The next two phases de-virtualize virtual threads progressively to match precise threading runtime constraints. The final phase lowers a virtual thread to threaded Intermediate Language (IL) by emitting calls supported by the runtime library.

Phase I: Enabling transformations and loop analysis.

This phase enables loop transformations that can increase thread-level parallelism, improve data locality, and identify threadizable loops within a compilation unit (or routine). This phase is enabled with Inter-Procedural Optimization (IPO) as well. Therefore, it is not limited to a single compilation unit, but rather allows whole-program parallelism extraction.

Phase II: Virtual thread graph construction. This phase extracts thread-level parallelism captured by parallel loops and it constructs sibling/nesting relationships between virtual threads. In addition, it also collects private, firstprivate, lastprivate, and reduction variables to build the data environment d for each virtual thread.

Phase III: Devirtualization via privatization. This phase conducts transformations for all private, firstprivate, lastprivate, and reduction variables that are captured by the data environment d of virtual threads. For instance, given *firstprivate*(x), a local clone *thr_x* of global variable x is created on the stack and initialized with the value of x . All memory references to x in the thread are then substituted with *thr_x*.

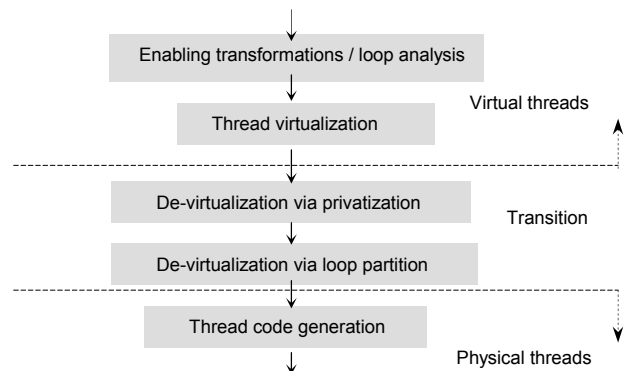


Figure 2: The new threadization framework

Phase IV: Devirtualization via loop partition. This phase partitions a loop using the thread identifier α based on a default schedule setting, or a scheduling type and chunk size specified with `-par-schedule-<type>` and `-par-schedule-size` options. The loop partition is represented internally with the following format:

*L*PARTITION (*tid*, *sched*, *cs*, *lv*, *glow*, *gup*, *gstride*, *vlow*, *vup*)

where *tid* denotes the thread identifier, *sched* denotes the loop scheduling type, *cs* denotes chunk size, *lv* denotes whether the code for computing last value is needed or not (FALSE means *last-value* is not needed), *glow* and *gup* denote the original loop lower and upper bounds, and *gstride* denotes the original loop stride. The parameters *vlow* and *vup* denote the loop's lower and upper bounds after loop partitioning for the virtual thread, and they are

computed by an OpenMP runtime library routine to which we pass in the other parameters in *LPARTITION*.

Phase V: Threaded-code generation. This phase maps a virtual thread to the compiler's intermediate code such as IL statements or intrinsics, and to OpenMP runtime library calls according to the target platform. These statements and calls include (i) *_fork_threads* call that creates physical threads; (ii) a loop partitioning call to compute *vlow*, *vup* based on loop information captured in the *LPARTITION* of each virtual thread node; (iii) a *T-entry* and *T-return* pair of statements for the virtual thread based on the *MET* technology presented in [10].

A distinct characteristic of the new framework is that the threadization is carefully broken down into a sequence of transformations, each of which gradually transforms a virtual thread IL, without a thread identifier, to a virtual thread IL parameterized by a unique symbolic thread identifier. This process is clearly illustrated by the evolution of data properties and code re-structuring through each phase.

Loop Transformations for Threadization

Under the new framework, the compiler performs all necessary loop transformations to achieve a good data locality while preserving and enabling threadization opportunities. Consider the following loops from the subroutine *Bench_StagegeredLeadfrog2* in 436.cactusADM of the SPEC CPU2006 benchmarks.

```
km = 1
do j=1,ny
  do i=1,nx
    lalp(i,j,km) = alp(i,j,1)
    fac = -2.0d0*dt*lalp(i,j,1)
    ADM_gxx(i,j,1) =
      ADM_gxx_p(i,j,1)+fac*ADM_kxx_stag_p(i,j,1)
    lgxx(i,j,km) = ADM_gxx(i,j,1)
    ...
  end do
end do
k0 = 2
do j=1,ny
  do i=1,nx
    lalp(i,j,2) = alp(i,j,2)
    fac = -2.0d0*dt*lalp(i,j,2)
    ADM_gxx(i,j,2) =
      ADM_gxx_p(i,j,2)+fac*ADM_kxx_stag_p(i,j,2)
    lgxx(i,j,k0) = ADM_gxx(i,j,2)
    ...
  end do
end do
```

In Phase I, the compiler analysis proves that there are no loop-carried data dependencies for the loops, and no data dependencies that prevent loop fusion. Thus, the actions taken by the compiler are to fuse the two loops first, and then to perform the steps described in the previous section. When threadization is done, the compiler emits a "FUSED LOOP WAS PARALLELIZED" diagnostic. In this example, loop fusion increases the granularity of the

parallel loop, which is an effective loop transformation to reduce thread forking and mapping overhead. After threadization, the vectorization phase will operate on the virtual thread code. In this example, the compiler continues by distributing the *i*-loop to restrict the number of data streams per resulting loop, which favors write-buffer combining, and then it vectorizes the resulting smaller loops. The compiler emits two "PARTIAL LOOP WAS VECTORIZED" diagnostics in this case. This indicates that an effective interaction of loop transformations, threadization, and vectorization can leverage the full potential of the Intel Core 2 Duo and Quad processor to achieve higher performance.

Cost Model for Threadization

Once a threadizable loop is identified in Phase I, Phase II forms a region within which the virtual thread will be constructed at compile time. Additionally, as the cost of thread activation and synchronization in a real system is in the range of hundreds of cycles on Intel Core 2 Duo and Quad processors, a key criterion in selecting a proper parallel loop candidate is to minimize the overhead of thread management.

A complementary goal is to ensure that the virtual thread, once invoked, runs for an adequate number of cycles in order to amortize the thread activation cost. Therefore, it is desirable to choose a loop that iterates a reasonably large number of times. The cost estimation is done via a table-driven technique based on the Intel Core 2 Duo and Quad processor instruction latency information combined with the profiling information of basic block execution counts. This algorithm is effective, especially when combined with function inlining.

Runtime Threadization Control

Statically, loops that incur a large number of instruction cycles and no loop-carried data dependencies are identified for threadization. However, selecting an appropriate loop for threadization requires that loop tripcount and number of cycles taken for each iteration are known. Often, the loop's lower and upper bounds are unknown at compile time, so the compiler can not compute the tripcount statically. In general, the static cost analysis may not provide an accurate cost estimation to guide and guard threadization in this case. To solve this issue, the new threadizer generates symbolic runtime test expressions and multi-versioned loops. Assume the symbolic tripcount expression of loop *L* is $E_{\text{tripcount}}(L)$, the estimated execution cycles of loop body of loop *L* is $C(L)$. The following runtime tests are generated to control the threadization at runtime:

- $C(L) < \text{Threshold}_{\text{par}}$
- $E_{\text{tripcount}}(L) \times C(L) < \text{Threshold}_{\text{par}}$

Multi-versioning is necessary for runtime threadization control. Consider, for example, the following sequential loop in C:

```
unsigned i, nd[1000];
/* size and target are incoming arguments */
for (i=0; i<size; i++) {
    nd[i] *= (1 << target);
}
```

This loop is selected as a candidate loop for threading based on loop analysis. Then, static cost analysis finds that $C(L) < \text{Threshold}_{\text{par}}$; however, the loop's upper bound t_0 (representing size) and the tripcount t_4 in the IL below are unknown at compile time. Hence, a runtime test code $t_4 < \text{Threshold}_{\text{par}} / C(L)$ is generated together with two-versioned loops. The pseudo-threaded code generated is sketched below.

```
BEGIN REGION
    t97 = _ok_to_fork();
    if ( t97 != 0 ) {
        if (t4 < 1363) JMP L123; // runtime test
        u = t0; // t0 represents "size"
        p1 = t1; // t1 represents "target"
        _fork_threads(... _parloop, &u, &p1, &nd, ...);
    }
    else {
L123:
        DO i = 1, t0
            nd[i-1] = nd[i-1]*(1<<t1);
        END DO
    }
END REGION
```

```
BEGIN REGION // threaded code
T-entry _parloop( (... , tid, upper, nd, p1);
    low = 1; // original loop lower bound
    up = upper; // original loop upper bound
    gu = up;
    _loop_partition(tid, sched,... , &low, &up);
    t105 = low; // local loop lower bound
    t106 = up; // local loop upper bound
    t1 = p1; // p1 represents "target"
    if (t105 <= gu) {
        DO i = 0, t106 - t105
            (*nd)[i + t105] =
                (*nd)[i + t105] * (1 << t1);
        END DO
    }
    _join_threads(tid);
T-return;
END REGION
```

In this example, if t_4 is less than 1363, the execution will switch to serial loop to avoid threading overhead. The runtime threadization control is a simple yet efficient way for parallelizing loops with unknown bounds at compile time. We obtained good speedup by emitting multi-versioned serial and threaded code at compile time, and using runtime tests to select the most beneficial version to execute in some applications.

REVAMPING THE VECTORIZER

The new vectorizer is designed to be tightly integrated with our existing enhanced high-level loop transformation

framework. The strengths of the new vectorizer include the following:

- A new Abstract Vector Representation (AVR) is designed to bridge the semantic gap between high-level representation and low-level instruction.
- Better interaction with the new FP-model and other loop optimizations produces better performance.
- The new vectorizer is moved downstream to use SSA and leverage global constant propagation and Common Sub-expression Elimination (CSE).
- Table-driven type selection and code generation with a well-tuned cost model simplify maintenance and future extensibility.

Essentially, the vectorizer converts sequential code into a vector form that exploits all Streaming SIMD Extensions. Consider, for example, the following sequential loop in C:

```
unsigned char a[N], b[N];
...
for (i = 0; i < N; i++) {
    int t = a[i] + b[i];
    b[i] = (t > 255) ? 255 : t;
}
```

When compiled for a target architecture that supports SSE2, the compiler generates a vectorized loop with the following assembly code:

```
xor     eax, eax
L: movdqa xmm0, XMMWORD PTR [_a+eax]
; load 16 char from a
paddusb xmm0, XMMWORD PTR [_b+eax]
; add and saturate 16 char from b
movdqa XMMWORD PTR [_b+eax], xmm0
; store 16 char into b
add     eax, 16
cmp     eax, ebx
jnb     L ; looping logic
```

Here, the compiler first recognizes a vector loop with idiomatic saturation arithmetic and proper alignment of all access patterns and subsequently converts the code into appropriate SIMD instructions with vector length 16. Due to the removal of a conditional branch relative to a sequential implementation of the loop, in this particular case, vectorization typically exhibits a speedup that exceeds the vector length.

Vectorization for Streaming SIMD Extensions strongly resembles vectorization for traditional vector architectures [1, 11], like a pipelined vector processor. There are a few important differences as well [2], briefly described below:

- A relatively short and fixed vector length requires a sequential “cleanup” loop to deal with the remaining iterations, but it also makes the vector instructions more suitable for fine-grained parallelism, as was first advocated in [2]. The shorter vector length can also be exploited during data dependence analysis.

- A strong sensitivity to natural alignment (typically 16-byte) requires elaborate compiler support to select, detect, or enforce a proper alignment on memory references.
- An idiomatic instruction set requires advanced idiom recognition in the compiler, such as detecting the saturation addition in the example above.

Since vector lengths increase for narrower data types, compiler analysis is required to choose the narrowest possible data type that preserves the original meaning, such as recognizing that all 32-bit operations on variable t can be done in 8-bit precision. An in-depth description of vectorization technology in the Intel C++/Fortran compiler is given in Reference [2]. For the remainder of this section we focus on a few specifics for the Intel Core 2 Duo and Quad processors.

Alignment Optimization

In the Intel Core micro-architecture, SIMD performance is still rather sensitive to natural alignment. Therefore, an important aspect of effective vectorization in the compiler is to select, detect, and enforce a favorable alignment on memory references. For instance, the vector loop in the previous section may only use the efficient `movdqa` to load 16 bytes of data after the compiler has proven that both the *initial* alignment (alignment on entry of the loop) and the *sustained* alignment (alignment preserved during execution of a loop)¹ of the memory reference `a[i]` is 16-byte aligned. The less efficient `movdqu` should have been used if the memory reference had an unknown alignment or was misaligned, because using aligned data movement instructions on unaligned memory locations yields a program fault. The Intel compiler uses a continuously growing assortment of alignment optimizations, including data layout optimization, inter- and intra-procedural alignment propagation, and loop transformations such as static and dynamic loop peeling and multi-versioning [1].

Alignment propagation resembles classical constant propagation, but uses a more elaborate lattice of alignment values $\langle 2^n, o \rangle$, where o denotes a non-negative offset relative to a base 2^n and corresponding jump functions. Using a lattice of bases combined with offsets, a method described in [2], propagates more accurate information than just bases and ultimately offers more opportunities for optimizations, such as peeling off unfavorable alignments or using specific instruction sequences for a data movement that splits a cache line. The information is associated with *all* variables, not just pointers, and has

¹ A vector loop using SSE always sustains an initial 16-byte alignment for unit stride memory references. For a scalar loop, the sustained alignment depends on the data width of these memory references.

been proven empirically to improve the accuracy of the computed results. A variety of alignment-related optimizations can be found in [1, 3, 6, 8].

Vectorizer Support for SSSE3

The SIMD Extensions 3 [4] extend previous generations of SIMD extensions with sixteen new instructions that can operate on 128-bit operands or old-style 64-bit operands of the MMX™ technology. New instructions most commonly used by automatic vectorization are listed in Table 1.

Table 1: SSSE3 instructions used for auto-vectorization

Instruction	Suffix	Description
<code>palignr</code>		<i>Packed align right</i>
<code>psign</code>	$[b, w, d]$	<i>Packed negation based on sign</i>
<code>pabs</code>	$[b, w, d]$	<i>Packed absolute value</i>
<code>phadd</code>	$[w, d]$	<i>Packed horizontal add</i>
<code>pshuf</code>	$[b]$	<i>Packed shuffle</i>

The `palignr` instruction is used to optimize multiple unaligned loads with a statically known offset into aligned loads that are subsequently rearranged into the appropriate vector format. The idiomatic `psign` instruction is recognized in programming constructs that negate data elements based on the sign of other data elements. The packed absolute value instruction `pabs` provides a more compact and efficient way of vectorizing this operation than previously-used emulation sequences. Consider, as an example, the following loop that computes the absolute value of all elements in an array of type `char`.

```
for (i = 0; i < N; i++) { /* ABS EXAMPLE */
    int t = a[i];
    if (t < 0) t = -t;
    a[i] = t;
}
```

The generated assembly code for plain SSE2 as well as SSSE3 is illustrated below. In this case, SSE2 shows a ~20x speedup, while SSSE3 shows a ~30x speedup.

```
L1: movdqa  xmm1, XMMWORD PTR [_a+eax]
    pxor   xmm0, xmm0
    pcmpgtb xmm0, xmm1
    pxor   xmm1, xmm0
    psubb  xmm1, xmm0
    movdqa XMMWORD PTR [_a+eax], xmm1
    add    eax, 16
    cmp    eax, ebx
    jb    L1

L2: pabsb   xmm0, XMMWORD PTR [_a+eax]
    movdqa XMMWORD PTR [_a+eax], xmm0
    add    eax, 16
    cmp    eax, ebx
    jb    L2
```

Similarly, the `phadd` instruction provides a more compact way of summing up partial results after

vectorization sum reductions [1, 2]. However, the current micro-architectural implementation does not provide any latency reduction over the more elaborate instruction sequences used formerly. Finally, the `pshufb` instruction provides an efficient way to perform a wide variety of data rearranging, as illustrated with the following loop that operates on two arrays of type `char`.

```
for (i = 0; i < N; i+=4) {
    a[i+0] = b[i+3];
    a[i+1] = b[i+2];
    a[i+2] = b[i+1];
    a[i+3] = b[i+0];
}
```

This conversion between a little-endian and big-endian representation of 32-bit data elements (4 bytes) can be vectorized effectively as follows.

```
L: movdqa    xmm1, XMMWORD PTR [_b+eax]
    ; load 16-bytes from b
    pshufb   xmm1, xmm0
    ; shuffle 16-bytes as defined in xmm0
    movdqa   XMMWORD PTR [_a+eax], xmm1
    ; store 16-bytes into a
    add     eax, 16
    cmp     eax, ebx
    jb     L    ; looping logic
```

Here, register `xmm0` is pre-loaded with the appropriate 4x4 reshuffling pattern. In fact, any reshuffling of 4 consecutive bytes, even allowing for repeats, can be similarly implemented. The instruction is also used in a peep-hole-like optimization of various data rearranging sequences generated by the vectorizer.

ENHANCED LOOP OPTIMIZATIONS

Besides revamping the threadizer and vectorizer, in the Intel 10.1 compilers, a single unified framework is designed primarily to provide better interaction among loop optimizations, threadizer, and vectorizer. The loop optimizations target cache and memory optimizations that are well known in the literature such as linear loop transformations, distribution, fusion, blocking, unroll-jam, loop-multi-versioning, and scalar replacement [7, 11, 12]. In order to derive the maximum possible performance for programs with effective threadization and vectorization, individual loop optimizations are enhanced and ordered in such a way as to achieve the best memory-locality while retaining the property that the innermost-loop can be efficiently vectorized. Similarly, optimizations are applied to a loop-nest to enable the threadization of the outer loop wherever possible, thus increasing the granularity of parallelism and reducing the overheads.

Loop Distribution Enhancements

Loop Distribution Pass-1 is invoked to generate more coarse-grained threadizable loops with statement re-ordering and grouping while preserving the correctness

and perfect nested loops that enable further loop optimizations such as interchange.

Loop Distribution Pass-2 is invoked before vectorization. For each distributed loop, this groups together memory-references that have required stride, data-type, and alignment. These properties ensure efficient vectorization of each such loop (where vectorization is legal) making good use of the available micro-architectural resources. Loop distribution heuristics also trade off maximally distributing for vectorization against improving cache reuse for vectorized loops. Intel Core micro-architecture features more write-combining buffers and larger data caches with higher associativity than previous generations. This enables better performance through vectorization without excessive loop distribution, thereby reducing vectorized loop overheads.

Loop Multi-versioning

The multi-versioning helps to deal with two potential roadblocks that prevent a loop from being vectorized or parallelized. The first roadblock is when the loop contains references with cross-iteration data dependencies. The second one is when the references' cross-iteration strides are unknown, e.g., dope vector based arrays in Fortran90. In either case, the multi-versioning module generates code that checks whether "required conditions" hold during runtime. It also generates different copies of the loop such that each copy is guarded under a different condition, and optimized according to the guarded condition.

For example, if a loop has references `a(i)` and `b(i)`, and data dependence cannot prove that `a` and `b` do not overlap, there are two possible ways that multi-versioning can help. If the compiler decides that vectorization is important, versioning will generate a test to ensure that `a(0)` and `b(0)` are at least 16 bytes apart. If this condition is tested true at runtime, a version of the loop that has been vectorized will be run. Otherwise, a non-vectorized version of the loop will be run; the latter version may still be optimized in other ways (e.g., unroll). Both loop versions and the runtime test have been pre-generated into the executable by the compiler. The multi-versioning module generates the different loop versions, and it annotates their properties with internal directives that are then used by the vectorizer.

On the other hand, if threadization is more important, versioning will generate a test to ensure that the arrays do not overlap (using the initial addresses of `a` and `b`, and the number of loop iterations). The loop version guarded by this independence test can then be safely parallelized.

Similarly, if a loop has references to dope vector-based arrays (e.g., assumed shape arrays), versioning can generate checks to examine the stride value of the arrays

from the dope vectors during runtime. If the strides are all one, the loop may then be efficiently vectorized (assuming other vectorization conditions pass.)

The versioning uses a heuristic to decide on the number of the tests and number of versions of the loops, to reduce the impact on executable size.

Loop Blocking and Unrolling

The loop blocking and loop unrolling phases have been improved for the Intel Core micro-architectures. Based on our experience with application code, the enabling decisions and the optimization parameters have been modified to make the best use of the new cache architecture. The phase ordering of the blocking phase has also been modified with respect to the vectorization phase to extract the maximum benefit possible from these optimizations.

Vectorizer modifies simple inner loops to create vector loops. This leads to complex loop structure that is not amenable to blocking—there are several cases where vectorization degrades performance when compared to just loop blocking. Another loop-blocking phase has been added before vectorization, so that blocking can make better use of the cache, and later vectorization on the innermost blocked loop can further improve parallelism across loop iterations.

The loop blocking phase has also been enhanced in our new unified framework to get the best out of the Intel Core micro-architecture. Blocks or Tiles are used to hold data in the cache and are the stride factors for the outer block-controlling loops. Block or Tile-size selection algorithms are also improved. Our primary focus now is to improve cache locality at the L2 cache level. We try to enable more register re-use by performing unroll-jam (a.k.a register-blocking) of outer loops inside the inner blocked loops.

The mechanism that controls the enabling or disabling of the loop unrolling has been improved. Unrolling can lead to register pressure resulting in poor code performance due to register spills and fills. Besides the obvious cases, it is hard to predict at compile-time whether loop unrolling would help or degrade performance. Our implementation makes this decision based on various program and architectural parameters. Determination of loop unrolling factors also needs to be aware of register pressure in the inner loop. Our experience shows that small unroll factors are effective in most cases.

Loop Fusion and Interchange

Loop fusion combines adjacent conforming nested loops into a single nested loop. This optimization can improve the cache context and increase the amount of computation,

thus increasing the granularity of threadization reduced overheads. Loop interchange is done in such a way as to improve threadization at the outer level, and at the same time, keep the memory accesses in the innermost-loop unit-strided to enable efficient vectorization.

ADVANCED CODE GENERATION

The Intel compiler uses its intimate knowledge of the Intel micro-architecture to guide instruction selection tradeoffs. The compiler takes advantage of efficient instructions and instruction forms while avoiding inefficient instruction sequences. In addition, a restricted instruction scheduling form is used to enhance performance.

Instruction Selection

The bit test instruction `bt` was introduced in the i386™ processor. In some implementations, including the Intel NetBurst® micro-architecture, the instruction has a high latency. The Intel Core micro-architecture executes `bt` in a single cycle, when the bit base operand is a register. Therefore, the Intel C++/Fortran compiler uses the `bt` instruction to implement a common bit test idiom when optimizing for the Intel Core micro-architecture. The optimized code runs about 20% faster than the generic version on an Intel Core 2 Duo processor. Both of these versions are shown below:

C source code

```
int x, n;
...
if (x & (1 << n)) ...
```

Generic code generation

```
; edx contains x, ecx contains n.
mov     eax, 1
shl    eax, cl
test   edx, eax
je     taken
```

Intel Core micro-architecture code generation

```
; edx contains x, eax contains n.
bt     edx, eax
jae   taken
```

Variable-length instructions pose a challenge to the processor's instruction decoder, which must identify where one instruction ends and the next begins. Some instruction prefixes change the length of their instructions and cause a significant decoder stall in the Intel Core micro-architecture. Integer instructions that take immediate arguments and use the operand size override prefix `0x66` suffer from this penalty, because the size of the immediate operand is changed by the prefix. The compiler avoids these instructions, as shown below:

C source code

```
short *p;
...
*p &= 0x5555;
```

Generic code generation

```
; The and instruction encodes
; as hex 66 81 20 55 55.
; The immediate is 2 bytes.
mov  eax, DWORD PTR _p
and  WORD PTR [eax], 0x5555
```

Intel Core micro-architecture code generation

```
; The and instruction encodes
; as hex 25 55 55 00 00.
; The immediate is 4 bytes.
mov   edx, DWORD PTR _p
movzx eax, WORD PTR [edx]
and   eax, 0x5555
mov   WORD PTR [edx], ax
```

The vector unpack low instructions are convenient for gather and broadcast operations, which occur frequently in vector code. With the exception of the 64-bit to 128-bit instructions `punpcklqdq` and `unpcklpd`, unpack instructions are costly in the Intel Core micro-architecture compared to alternative code sequences. The Intel Fortran/C++ compiler favors alternative code sequences when optimizing for the Intel Core micro-architecture. Two examples are given below:

Example I: Broadcast the least-significant single-precision floating-point vector element.

Generic code generation

```
unpcklps xmm0, xmm0
unpcklps xmm0, xmm0
```

Intel Core micro-architecture implementation

```
movsldup  xmm0, xmm0
movlhps   xmm0, xmm0
```

Example II: Gather four single-precision floating-point elements from locations 128 bytes apart.

Generic code generation

```
movss  xmm3, [eax]
movss  xmm2, [128+eax]
movss  xmm0, [256+eax]
movss  xmm1, [384+eax]
unpcklps xmm3, xmm0
unpcklps xmm2, xmm1
unpcklps xmm3, xmm2
```

Intel Core microarchitecture implementation

```
movss  xmm2, [eax]
movss  xmm3, [128+eax]
movss  xmm0, [256+eax]
movss  xmm1, [384+eax]
unpcklpd xmm2, xmm0
unpcklpd xmm3, xmm1
psllq  xmm3, 32
orps   xmm3, xmm2
```

The conditional move instruction `cmovCC` presents an interesting dilemma for the compiler. It can achieve dramatic performance improvements when replacing a poorly predicted branch. On the other hand, replacing a branch with `cmovCC` may lengthen the critical path and cause a slowdown in cases where the branch is well predicted. Branch predictability is difficult to determine at compile time, so the decision of whether to use a branch

or conditional move is made by rough heuristics that can often yield poor results. The Intel Core micro-architecture simplifies this tradeoff by providing a low-latency `cmovCC` implementation compared to previous generations. When optimizing for the Intel Core micro-architecture, the Intel compiler more aggressively eliminates branches in favor of `cmovCC`. This strategy yields a substantial speedup for some applications.

Instruction Scheduling

In a dynamically scheduled environment like the Intel Core micro-architecture, the effectiveness of instruction scheduling at compile time is greatly reduced. Using its knowledge of machine internals, however, the Intel C++/Fortran compiler is able to schedule instructions to avoid micro-architectural pitfalls and to take advantage of micro-architectural features.

As described earlier, the Intel Core micro-architecture features a data prefetcher that speculatively load data into the caches. The L2 to L1 cache prefetcher uses a 256-entry table to map loads to load address predictors. This table is indexed by the lower eight bits of the instruction pointer (IP) address of the load. Since there is only one table entry per index, two loads offset by a multiple of 256 bytes cannot both reside in the table. If a conflict occurs in a loop and involves a predictable load, the effectiveness of the data prefetcher can be drastically reduced. In a critical loop, this can cause a significant reduction in overall application performance.

The compiler attempts to avoid IP prefetch conflicts in inner loops. It first identifies and classifies load instructions, distinguishing between loads that are likely to benefit from prefetching and those that are not. For example, loads from constant addresses will not benefit from prefetching. An IP prefetch conflict between two such loads is unlikely to affect performance. After identifying and classifying loads, the compiler inserts `nop` padding such that each prefetchable load has a modulo-256 address that is different from every other load in the inner loop.

The Intel Core micro-architecture can combine an integer compare (`cmp`) or test (`test`) instruction and a subsequent conditional jump instruction (`jcc`) into a single micro-operation through a process called macro-fusion. For macro-fusion to occur between `cmp` and `jcc`, the jump condition must test only the carry and/or zero flags, which is typically the case for unsigned integer compare and jump operations. The Intel Fortran/C++ compiler takes advantages of the macro-fusion feature by generating code that is likely to expose macro-fusion opportunities by detecting compare and jump instructions that are candidates for fusion. During scheduling, it forces these compare and jump instructions to be adjacent. Note

that this strategy conflicts with a traditional latency-based strategy, which tends to separate producers (the compare in this case) from consumers (the conditional jump).

PERFORMANCE RESULTS

In this section we provide performance validation of the new threadizer and vectorizer using the industry-standardized computationally intensive benchmark suite SPEC^{*} CPU2006 in which the CINT2006 suite comprises 12 integer C and C++ benchmarks, and the CFP2006 suite comprises 17 floating-point Fortran, C and C++ benchmarks, all derived from real-life applications that have up to 932818 lines of code. The SPEC CPU2006 benchmarks are widely used and considered to be representative of a wide spectrum of application domains. The multi-core system used to measure performance is configured with two 2.67 GHz Intel Core 2 Quad processors with a 4M L2 cache, an 8 GB RAM, and booted with an SuSE Linux OS.

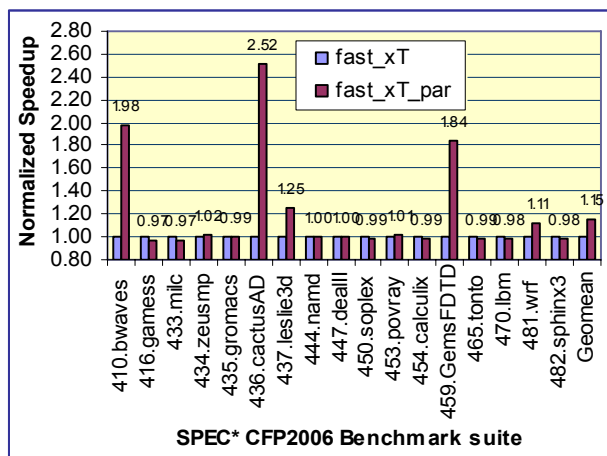


Figure 3: SPEC CPU2006 speedup estimates with auto-threadizer based on internal measurements

To evaluate the effectiveness of the new threadizer, we first measured the baseline performance with the option `-fast` (i.e., `-ipo -O3 -xT -no-prec-div -static`). Then, we added the `-parallel` switch to measure the speedup over the fully optimized baseline performance. The contributions from threadization are shown in Figure 3, which shows the speedup of benchmarks in the SPEC CFP2006 suite delivered by the auto-threadizer. The 15.45% geomean gain of all speedups is shown in the last column. Even though default base optimizations already obtain acceptable performance, auto-threadization of the Intel C++/Fortran compiler further boosts the performance of a number of benchmarks substantially, going up to a 2.52x speedup for a 436.cactusADM. No benchmark suffered a noticeable slowdown due to the auto-threadizer.

Auto-converting a sequential program into threaded code becomes an increasingly important technique to leverage multi-core platforms in a transparent manner. Besides the gain delivered for SPEC CFP2006 performance, the auto-threadizer delivered a 12.17% gain (geomean) for SPEC CINT2006 on top of fully optimized serial code by using `-parallel` and `-par-runtime-control` options that contributed to a 4.63x performance speedup for the 462.libquantum.

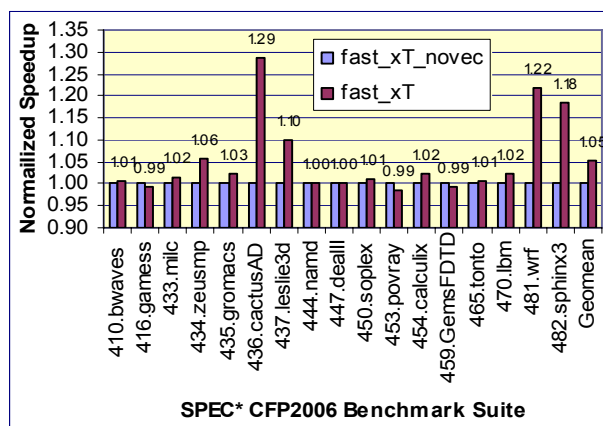


Figure 4: SPEC CPU2006 speedup estimates with auto-vectorizer based on internal measurements

Vectorization also forms a significant part of performance improvements. To evaluate the effectiveness of the new vectorizer, we first measured the baseline performance using `-fast` but with the vectorizer off (`fast_xT_novec`). Then, we measured the performance with the vectorizer enabled (`fast_xT`) to get the speedup over `fast_xT_novec`. The contributions made by vectorization are shown in Figure 4, which shows the speedup of benchmarks in the SPEC CFP2006 suite delivered by the auto-vectorizer. The 5.11% geomean gain is shown in the last column. Even though baseline optimizations already provide high performance, the auto-vectorizer of the Intel C++/Fortran compiler further boosts the performance of a number of benchmarks substantially, going up to a 1.29x speedup for 436.cactusADM. Albeit generally biased towards floating-point applications, the advanced code generation makes a noticeable contribution to integer applications: a 33.6% gain. In other cases, experience shows that it makes performance less sensitive to minor changes in the generated code.

CONCLUSION

The Intel 10.1 C++/Fortran compiler features various advanced compiler optimizations to leverage the enhanced capabilities of Intel Core 2 Duo and Quad processors. Threadization exploits thread-level parallelism in serial programs; vectorization exploits SIMD-based vector-level parallelism; and advanced code generation exploits

important micro-architectural features for gaining a higher performance. This paper presented the implementation of the new threadizer and vectorizer and an overview of advanced code generation that specifically leverages the Intel Core micro-architecture.

Performance validation was conducted with a large set of real-life industry-standard benchmarks. It was shown that advanced optimizations of the Intel C++/Fortran compiler can obtain further improvements over optimized code, with contributions from threadization, vectorization, loop optimizations, and target-specific code generation. Furthermore, these optimizations were added in a manner that still allows for our overall goal of continuing to generate code that runs well across all processors.

More information on Intel high-performance compilers for Intel Architectures can be found at the Intel website <http://intel.com/software/products/>.

ACKNOWLEDGMENTS

The authors thank Aart J.C. Bik, Peng Tu, Kannan Narayanan, Sumesh Udayakumaran and all members of the loop optimizer team in Intel Novosibirsk Compiler Lab for their direct and indirect contributions and for their productive collaboration throughout the new threadizer and vectorizer development projects. Special thanks to Aart J.C. Bik for his technical leadership as the vectorizer architect in his tenure at Intel. We also appreciate the opportunities and guidance from Kevin J. Smith, Suresh K. Rao, Wei Li, Bill Savage, and other members of the management of the Intel Compiler Lab. In addition, we thank all members of the Intel C++/Fortran compiler teams and the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper.

REFERENCES

- [1] Aart J.C. Bik, David Kreitzer, Xinmin Tian, "Compiler optimizations for the Intel[®] Core[™]2 Duo Processor," submitted to *International J. of Parallel Programming*, April 2007.
- [2] Aart J.C. Bik, *The Software Vectorization Handbook*, Intel Press, Hillsboro, Oregon, 2004.
- [3] A. Eichenberger, P. Wu, K. O'Brien, "Vectorization for SIMD Architectures with Alignment Constraints," in *Proceedings of the ACM SIGPLAN 2004 Conference on Prog. Lang. Design and Implementation*, 82-93, Washington DC, June 2004.
- [4] Intel Corporation, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corp. at <http://developer.intel.com/>, 2007.

- [5] Andreas Krall and Sylvain Lelait, "Compilation Techniques for Multi-media Processors," *International Journal of Parallel Programming*, 28(4):347-361, 2000.
- [6] Samuel Larsen and Saman Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, B.C., June 2000.
- [7] Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, San Mateo, California, 1997.
- [8] Ivan Pryanishnikov, Andreas Krall and Nigel Horspool, "Pointer Alignment Analysis for Processors with SIMD Instructions," in *Proceedings of the 5th Workshop on Media and Streaming Processors*, San Diego, CA, December 2003.
- [9] John D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture TCCA*, Newsletter, December 1995.
- [10] Xinmin Tian, Milind Gikar, Aart J.C. Bik, and Hideki Saito, "Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs," *The Computer Journal*, Vol. 48, Issue 5, pps. 558-601, 2005.
- [11] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, California, 1996.
- [12] Somnath Ghosh, Abhay Kanhere, Rakesh Krishnaiyer, Dattatraya Kulkarni, Wei Li, Chu-Cheow Lim, John Ng, "Integrating High-Level Optimizations in a Production Compiler: Design and Implementation Experience," in *Compiler Construction, 12th International Conference, CC 2003*: 303-319, Warsaw, Poland, April 2003.

AUTHORS' BIOGRAPHIES

Xinmin Tian is a Principal Engineer and Compiler Architect with Intel's Software and Solutions Group. He leads parallelization, vectorization, OpenMP compiler and transactional memory compiler development projects for IA-32, Intel[®] 64 and IA-64 multi-core processors in the Intel Compiler Lab. He holds a Ph.D. degree in Computer Science from Tsinghua University. He joined Intel in 1999. His e-mail is xinmin.tian at intel.com.

Ernesto Su is a Senior Staff Engineer with Intel's Software and Solutions Group. He received a B.S. degree from Columbia University and M.S. and Ph.D. degrees from the University of Illinois at Urbana-Champaign, all

in Electrical Engineering. He joined Intel in 1997 and is currently working on High-Performance Optimizations including loop optimizations, parallelizing compilers, and OpenMP. His e-mail is ernesto.su at intel.com.

David Kreitzer is a Senior Staff Engineer with Intel's Software and Solutions Group. He received his B.S. degree in Electrical Engineering from the University of Virginia in 1994 and his M.S. degree in Electrical and Computer Engineering from Carnegie Mellon University in 1996. He joined Intel as a rotation engineer in 1996 and in 1997 began working on compilers for IA-32 processors. He leads IA-32 and Intel 64 code generator development projects in the Intel Compiler Lab. His e-mail is david.l.kreitzer at intel.com.

Hideki Saito is a Staff Engineer with Intel's Software and Solutions Group. He received a B.E. degree in Information Science in 1993 from Kyoto University, Japan and a M.S. degree in Computer Science in 1998 from the University of Illinois at Urbana-Champaign. Prior to joining Intel, he was a Ph.D. candidate at UIUC. He is currently working on vectorization, parallelization, performance analysis and OpenMP. His e-mail is hideki.saito at intel.com.

Rakesh Krishnaiyer is a Senior Staff Engineer with Intel's Software Solutions Group. He received his B.Tech. degree in Computer Science and Engineering from IIT Madras in 1993 and his M.S. and Ph.D. degrees from Syracuse University in 1995 and 1998, respectively. Currently, he leads the High-Level Optimizer project in the Intel Compiler Lab. His e-mail is rakesh.krishnaiyer at intel.com.

Abhay Kanhere is a Staff Engineer with Intel's Software Solutions Group. He received a B.E. in Computer Engineering from Gujarat University, India and a Master of Science in Computer Science from the Indian Institute of Science, India. He joined Intel in 2000 and has been working on the high-level optimizer. He is currently a Project Lead in Emerging Products Lab, targeting compiler optimizations for Intel Architecture. His e-mail is abhay.kanhere at intel.com.

John Ng is a Principal Engineer with Intel's Software and Solutions Group. Currently, he manages the High Performance Optimizer and Interprocedural Optimizer team. He received a B.S. degree in Mathematics from Illinois State University and an M.S. degree in Computer Science from Rutgers University. He joined Intel in 1996. Prior to that, he worked on memory optimizations, vectorization, parallelization, and threading libraries at IBM for 15 years. His email is john.ng at intel.com

Chu-Cheow Lim is a Senior Staff Engineer with Intel's Mobility Group. He received a B.Sc. degree in Mathematical and Computational Sciences, an M.Sc.

degree in Computer Science from Stanford University, and a Ph.D. degree from the University of California at Berkeley. He has worked on loop optimizations and the Itanium code generator and also did research on speculative parallel threading in Intel. He is currently working on the graphics compiler for Intel's next-generation GPU. His e-mail is chu-cheow.lim at intel.com.

Somnath Ghosh is a Senior Staff Engineer with Intel's Mobility Group. He received his B.Tech. degree in Computer Science and Engineering from IIT Kharagpur, and his M.S. and Ph.D. degrees in Electrical Engineering from Princeton University. He is currently working on the graphics compiler for Intel's next-generation GPU. His e-mail is somnath.ghosh at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

SPEC[®], SPECint[®] and SPECfp[®] are registered trademarks of the Standard Performance Evaluation Corporation. For more information on SPEC benchmarks, please see <http://www.spec.org>

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS[®] Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from <http://www.intel.com>.

Additional legal notices at:

<http://www.intel.com/sites/corporate/tradmarx.htm>.

For further information visit:

developer.intel.com/technology/itj/index.htm