Volume 11 Issue 04 Published, November 15, 2007 ISSN 1535-864X DOI: 10.1535/itj.1104

Intel® Technology Journal

Multi-Core Software

This Intel Technology Journal (Volume 11, Issue 4, Q4 2007) focuses on multi-core software and takes a detailed and comprehensive look at important tools and methodologies to successfully thread many types of applications.

Inside you'll find the following articles:

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Volume 11 Issue 04 Published, November 15, 2007 ISSN 1535-864X DOI: 10.1535/itj.1104

Intel® Technology Journal

Multi-Core Software

Articles

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

By Lin Chao
Editor and Publisher, *Intel Technology Journal*

Multi-core processors are balanced for performance and power consumption. They can achieve high performance with optimal power consumption by sharing the work of executing tasks on multiple execution cores. To take advantage of these multiple cores, software (SW) needs to be designed to execute in parallel across multiple processor cores by the use of threads. SW threads have been long used in scientific and high-performance computing where large-scale computing resources are tied together to crunch on complex, numerically based mathematical problems. Today, multi-core processors are not just used in scientific computing; they are the standard in consumer desktop and mobile computers.

Application developers face a daunting task when threading their applications. New SW development tools won't eliminate the inherent challenges, but they can help simplify the problem by identifying thread correctness issues and performance opportunities. The nine papers in this Volume 11, Issue 4 of the *Intel Technology Journal* focus on multi-core SW and take a detailed and comprehensive look at important tools and methodologies to thread successfully many types of applications. Among these tools, the Threading Building Blocks Library (TBB) is available in open source. As the number of available cores continues to increase, it is important for SW developers to have the right tools to design and implement scalable solutions for today's and tomorrow's multi-core systems.

Below are snapshots of the nine papers.

The first paper takes a detailed look at the **Intel® 10.1 C++/Fortran Compiler** that includes new tools for code parallelization and vectorization. This compiler features various advanced optimizations to leverage the enhanced capabilities of 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.

The second paper looks at the **Intel® Performance Tuning Utility** whose performance analysis feature can assist at virtually every stage of both parallel and sequential SW performance tuning, and which may be extremely helpful in the preliminary stages of determining parallelization strategies. The authors discuss data-level decomposition strategy in real program examples and illustrate how the efficiency of a parallel implementation can be estimated, and which steps should be performed to optimize a parallel program using this Intel utility.

Intel provides a set of threading tools targeting various phases of the development cycle. In the third paper, we introduce the **principles of parallel application** design; and we then show how to parallelize an application with the help of threading tools during each phase of the development cycle. A multiple pattern matching algorithm is used as an example.

In the fourth paper, we present the **Intel® Math Kernel Library (MKL)** as a mathematical SW package for scientific and technical computation designed for ease of use in environments that can vary greatly. This paper is devoted to the optimization and parallelization of the library. The goal has been to provide an easy-to-use SW package to aid in the development of mathematical SW. Achieving this goal has a number of facets including functionality, compiler independence, and the most recent efforts in performance, focusing on helping the user get the full benefits from Intel® multi-core systems.

In the fifth paper, we provide an overview of **the Intel® Threading Building Blocks** (TBB), a SW C++ template library that simplifies the development of SW applications running in parallel. In this paper, we describe the design of the TBB task scheduler and several scheduling optimizations users can keep in mind while coding their applications. The task scheduler is complemented by the Intel TBB scalable memory allocator. We provide an overview of its design and look at the tradeoffs.

The sixth paper looks at a case study of semi-automatic **parallelization of large-scale integer applications**. The application is Intel's own C++ Compiler and we detail how we threaded this compiler to achieve an average of 2x speedup when compiling a range of CPU2000 benchmarks, showcasing our methodology and tools. We believe our approach is generally applicable to threading a large class of applications.

The seventh paper looks at a forward-looking, scalable **programming model called the *Ct*** and its associated API that leverages the strengths of data parallel programming to help address the challenges of multi-core software development. In this paper, we describe how Ct is designed for minimal effort by the developer, while providing forward scaling on multi-core Intel® Architecture platforms.

In the eighth paper, we look at applications for video analysis and management including search and retrieval. These applications are becoming mainstream and are mass-marketed. "**Content-Based Video Information Retrieval** (CBVIR)" is one of the commonly used techniques in this class of applications. In this paper, we optimize and parallelize a set of typical visual feature extraction applications. The underlying optimization and parallel techniques are representative of those used in video-analysis applications and can be further used in other applications to maximally improve their performance on multi-core systems.

The ninth paper looks at how the **different multi-core topologies** and the associated processor power management technologies bring new optimization opportunities to the process scheduler. We look into different scheduling mechanisms and the associated tradeoffs. Using the Linux* Operating System as an example, we also look into how some of these scheduling mechanisms are currently implemented. As the multi-core platform is evolving, some portions of the hardware (HW) and SW are being reshaped to take maximum advantage of the platform resources. We close this paper with a look at where future efforts in this technology are heading.

Foreword

By Bill Savage

General Manager, Developer Products Division, Software and Solutions Group

We at Intel and the industry need to accelerate the understanding of parallel software development and greatly reduce the effort needed to develop it. The relatively new prevalence of multi-core processors means both Intel and the software industry are confronted with the challenge of making parallel software development a common task for all programmers. While parallel software is common in high-performance computing and enterprise applications, running on multi-processor systems from workstations to large clusters, it is not widespread in mobile and desktop applications. However, multi-core processors are the new standard for these systems, and multi-core hardware can only achieve its full potential with parallel software: this means that both the excitement and challenge of parallel software has found its way to all computers.

Parallel computing in enterprise and high-performance computing has evolved over time. Hardware, software, and development tools have matured together, leading to improvements in and greater understanding of each area. Such maturation, while benefiting all these areas, has also placed greater demands on them. Over time, parallelization techniques improved and some standard practices were developed. Compilers, libraries, runtimes, and operating systems were developed to support these practices. But, even with these aids, creating parallel software remains a difficult problem, requiring a high level of expertise.

Examples of the problems that remain in parallel programming include finding parallel tasks in applications, expressing parallelism through appropriate constructs, finding errors in parallel code such as deadlocks and data race conditions, and tuning the performance and scaling of the application on the targeted hardware. Many of the papers in this issue of the *Intel Technology Journal* explore ways to advance our capabilities to meet these challenges, and how to improve capabilities in parallel programming and multi-core enabling.

Both Intel and the industry need to take parallel software development to the next level. We need to make it accessible to the majority of software developers, so that it becomes the prevalent method of development. We need to accelerate the evolution of the development environment to greatly reduce the amount of effort and expertise required. With each discovery and improvement in parallel software, we move closer to fully harnessing the power of multi-core processors.

I hope you enjoy this issue.

THIS PAGE INTENTIONALLY LEFT BLANK

Technical Reviewers for Q4 2007 ITJ

Zia Ansari, Software and Solutions Group
George Chaltas, Software and Solutions Group
Yen-Kuang Chen, Corporate Technology Group
Marius Cornea, Software and Solutions Group
Mark Davis, Software and Solutions Group
Bruce Greer, Software and Solutions Group
Victoria Gromova, Software and Solutions Group
Bo Huang, Software and Solutions Group
Wooyoung Kim, Software and Solutions Group
Alexey Kukanov, Software and Solutions Group
Sanjeev Kumar, Corporate Technology Group
David Levinthal, Software and Solutions Group
Tong Li, Corporate Technology Group
Hsin-Ying Lin, Software and Solutions Group
Geoff Lowney, Software and Solutions Group
Zhiqiang Ma, Software and Solutions Group
Sergey Maidanov, Software and Solutions Group
Tim Mattson, Corporate Technology Group
Paul Petersen, Software and Solutions Group
Leena Puthiyedath, Software and Solutions Group
James Reinders, Software and Solutions Group
Bratin Saha, Corporate Technology Group
Sanjiv Shah, Software and Solutions Group
Kevin J. Smith, Software and Solutions Group
Jackie (Weihua) Wu, Intel Information Technology
Youfeng Wu, Corporate Technology Group

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].

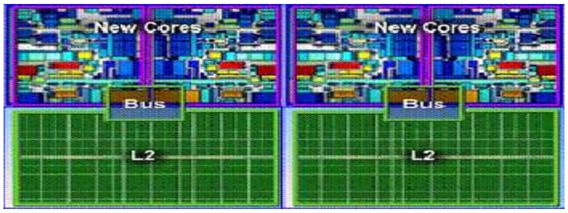


Figure 1: Quad-core processor schematic

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(α, 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(α, s, e, d)***).** The depth is computed recursively as follows:

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

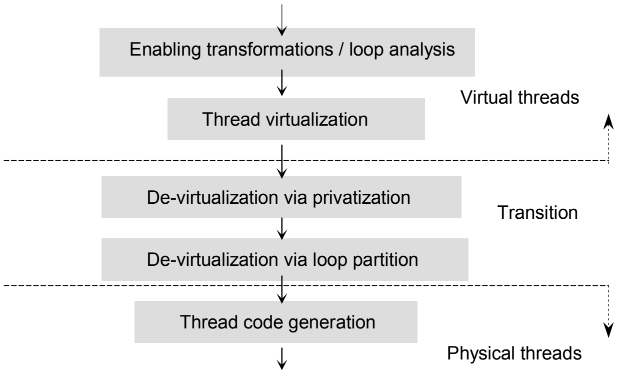

Enabling transformations / loop analysis
Thread virtualization
Virtual threads
De-virtualization via privatization
Transition
De-virtualization via loop partition
Thread code generation
Physical threads

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:

LPARTITION (*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 $E_{tripcount}(L)$, 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:

- $C(L) < Threshold_{par}$
- $E_{tripcount}(L) \times C(L) < Threshold_{par}$

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) < Threshold_{par}; however, the loop's upper bound `t0` (representing size) and the tripcount `t4` in the IL below are unknown at compile time. Hence, a runtime test code `t4` < Threshold_{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       // threadized 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 `t4` 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
   jb        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 $<2^n, o>$, 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 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
`palignr`		*Packed align right*
`psign`	*[b,w,d]*	*Packed negation based on sign*
`pabs`	*[b,w,d]*	*Packed absolute value*
`phadd`	*[w,d]*	*Packed horizontal add*
`pshuf`	*[b]*	*Packed shuffle*

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], xmm1
    add      eax, 16
    cmp      eax, ebx
    jb       L2
```

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

[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.

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 to 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.

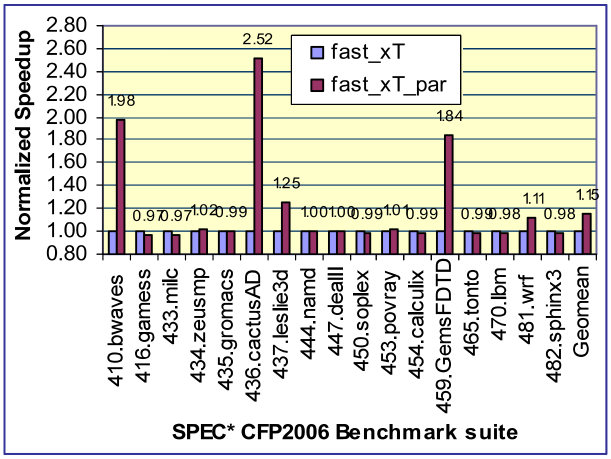


Figure 3: SPEC CPU2006 speedup estimates with auto-threadizer based on internal measurements

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.

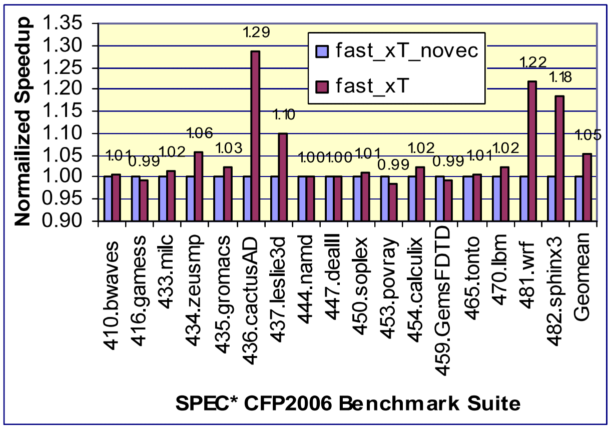


Figure 4: SPEC CPU2006 speedup estimates with auto-vectorizer based on internal measurements

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 Kulkami, 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.

Parallelization Made Easier with Intel® Performance-Tuning Utility

Alexei Alexandrov, Software and Solutions Group, Intel Corporation
Stanislav Bratanov, Software and Solutions Group, Intel Corporation
Julia Fedorova, Software and Solutions Group, Intel Corporation
Dr. David Levinthal, Software and Solutions Group, Intel Corporation
Igor Lopatin, Software and Solutions Group, Intel Corporation
Dmitry Ryabtsev, Software and Solutions Group, Intel Corporation

Index words: performance analysis, multi-core, parallelization, multi-threading, stack sampling, data access analysis

ABSTRACT

While multi-core processors are all around us, their effective use is made much easier with performance analysis tools that enable the developer to identify parallel execution opportunities and parallel execution bottlenecks. In this paper we introduce the new profiling capabilities available in the Intel® Performance Tuning Utility. These include statistical call tree analysis based on stack sampling, profile-guided loop detection, and event-based sampling data access profiling. The coordinated use of these features allows the developer to achieve better multi-core application performance.

INTRODUCTION

Parallel processing has been in common use for decades, but it's only recently that it became available on virtually every computer with the advent of multi-core processors. Historically, mass performance analysis tools [1, 2, 3, 4] have not generally had features designed to help identify parallel execution opportunities nor many of the common parallel execution bottlenecks. The Intel Performance Tuning Utility (Intel PTU), externally available at [5], has many of these features available in a single tool on Intel® Architecture.

Building on the experience of the Intel VTune™ Performance Analyzer, Intel PTU was designed to significantly improve on the data collection and display features available and add capabilities needed for enabling and analysis of parallel execution. Initially supported instrumentation-based control flow analysis (Exact Call Graph) suffers from excessive overhead and the resulting data distortion. This was replaced with a statistical approach to data collection based on call stack sampling in Intel PTU. The new statistical call stack sampling is supplemented with a precise call count data collection that can be used when required. Binary analysis was added to improve the disassembly displays through the use of basic blocks as the underlying execution units and to generate a control flow graph for the disassembly to simplify its interpretation. The binary analysis also enables the identification of loops, which, coupled with the performance data, allow for the identification of parallel execution opportunities. The full use of the Precise Event Based Sampling (PEBS) mechanism, only available on Intel® processors, enables simultaneous profiling by both Instruction Pointer (IP) and by data address, and a graphical filtering interface facilitates the analysis and identification of performance bottlenecks due to data access and layout issues.

All Intel PTU features are thread and CPU aware and can display data specific to either. Intel PTU works on a wide range of Windows* and Linux* operating system flavors and provides the same look-and-feel on all of them. It can be used from the command-line or from a GUI, which integrates into the Eclipse* IDE.

In this paper, we first describe the new features of Intel PTU in detail, as well as the analysis models facilitated by those features. We then illustrate the process of parallel software analysis and parallel execution discovery using Intel PTU on real program examples. We continue with an outline of areas for further development such as the quality of analysis and data representation, and finally we look at modern hardware performance monitoring capabilities.

Reading this paper requires some experience in parallel program design, as well as a certain knowledge of parallel performance monitoring and analysis. The sections below should not be viewed as providing a final recipe of efficient parallel software development or as describing methods of automated parallelization. Our goal, rather, is to illustrate the information that may be of use when dealing with parallel software and how that information may be collected, presented, and best interpreted with Intel PTU in order to ease the task of exploiting parallelization opportunities and parallel performance tuning.

NEW PERFORMANCE ANALYSIS MODELS

Performance tuning is like debugging: you'd like to avoid it but you cannot. And similar to the debugging process, you cannot do anything effectively unless you have a reliable tool that can save you a lot of time and effort. The importance of good debuggers and performance analyzers becomes critical as we move into the all-parallel world of microprocessing.

Intel PTU is meant to become such a time-saving tool. We do not pretend though that the tool can fully automate the tuning process. We simply believe that it is more important to put the burden of routine work on the tool, and let engineers think about their performance problems rather than about the tool itself.

To accomplish this, we focus on the following:

- Provide an easy way to perform repetitive data collection and analysis tasks.
- Provide effective and reliable methods of data collection and analysis that are relevant to both sequential and parallel programs.

The rest of this section describes in detail how the above goals are addressed by Intel PTU. We explicitly indicate product features that are especially valuable in the case of parallel analysis.

Projects, Configurations, and Experiments

To be effective in repetitive performance tuning tasks Intel PTU introduces the concepts of *project* and *profile configurations*.

Project contains information about the application, working directory, input arguments, maximum data collection time, etc. — in other words, it specifies *what* should be analyzed.

Profile configurations are a means of organizing collection methods into convenient and reusable shortcuts that can be reused for any project. Configurations can be predefined or user-defined. The predefined configurations for Intel PTU are as follows:

- Basic Statistical Call Graph
- Basic Sampling
- Basic Call Count
- Basic Data Access Profiling

A single profile configuration can be defined for multiple Intel processors, thereby generalizing its use. For example, the predefined Basic Sampling configuration is defined to collect two performance events corresponding to the "number of cycles" and "instructions retired" events mapped to different hardware events on different processors.

Creating a project is the first thing a user does. Once it is created, the predefined configurations list can be invoked by a simple right-click on the project in the navigator, floating the mouse over the "Profile As" option and selecting one of the profiling options (Figure 1).

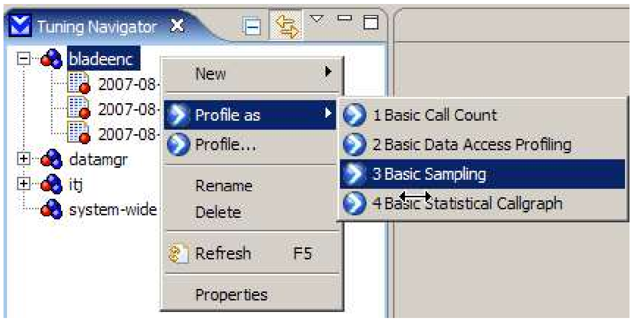
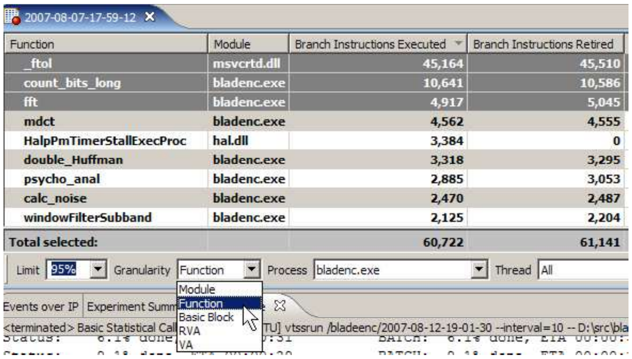


Figure 1: Launching a predefined data collection

Alternatively, right clicking on the project and selecting the "Profile..." option will invoke the configuration editor allowing users to select one of their own existing configurations or to create and invoke a new one.

After a profile configuration is applied to a project, the data are collected into an *experiment*. The basic visualization of the experiment data in Intel PTU is a tabular spreadsheet. The rows correspond to the currently chosen aggregation unit: module, function, basic block, or single address. The columns display the metrics for that region. The granularity of the aggregation unit can be selected through pull down menus (Figure 2).



2007-08-07-17-59-12

Function	Module	Branch Instructions Executed	Branch Instructions Retired
_ftol	msvcrtd.dll	45,164	45,510
count_bits_long	bladenc.exe	10,641	10,586
fft	bladenc.exe	4,917	5,045
mdct	bladenc.exe	4,562	4,555
HalpPmTimerStallExecProc	hal.dll	3,384	0
double_Huffman	bladenc.exe	3,318	3,295
psycho_anal	bladenc.exe	2,885	3,053
calc_noise	bladenc.exe	2,470	2,487
windowFilterSubband	bladenc.exe	2,125	2,204
Total selected:		60,722	61,141

Limit 95% Granularity Function Process bladenc.exe Thread All

<terminated> Basic Statistical Cal TU] vtssrun /bladeenc/2007-08-12-19-01-30 --interval=10 -- D:\src\bla

Figure 2: Intel® PTU tabular data view

For parallel programs, Intel PTU includes the current thread identifier (TID) and the CPU identifier in the program state information so that for each collection point it is clear which thread was executing on which processor at that moment. It is possible to filter the data for a specific thread, process, or CPU by using pull-down filter menus. Specifically this is useful for analysis of thread- or CPU-balancing.

Now, let's discuss the predefined analysis methods Intel PTU suggests.

Statistical Call Graph Analysis

Statistical Call Graph (a.k.a. Stack Sampling) collects its data by interrupting the program execution periodically (100 times per second) and capturing the current instruction pointer (IP) and the call stack. Using the IP value, it calculates how many samples occurred in a given function. This number is called Self Samples, because it corresponds to the number of samples that occurred in the function itself, not in the functions called by this function (callees). Places where a significant portion of samples occur are called *hotspots*.

The sample data can be aggregated by different units: function, module, basic block, or address. In the Intel PTU GUI the aggregation unit concept is exposed as "granularity." The function granularity is still the most popular, so it was made the default one in Hotspot view. By default, functions (rows) are sorted by the number of Self Samples, so the most active functions are displayed at the top.

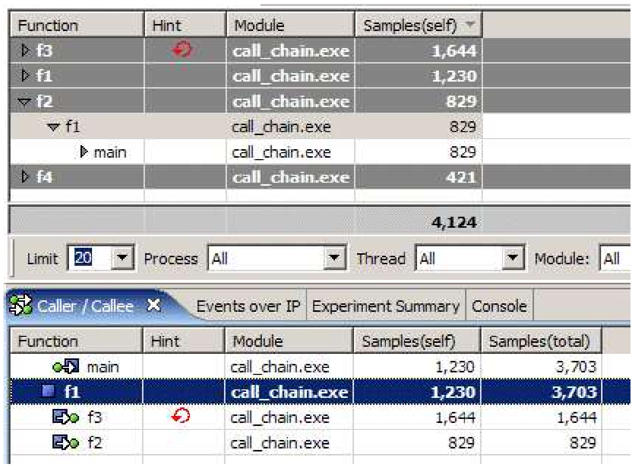
A second metric for each function, Total Samples, can be defined as the number of samples in the function plus all the samples that have the function in the call stack. Thus Total Samples measures the time in the function and everything the function calls.

To illustrate this we used a simple program:

```
int main() {
    f1();
}

int f1() {
    loop 30;
    f2();
    f3();
}

void f2() {
    loop 20;
}
```

```
int f3() {
    loop 40;
    foo();
}

void foo() {
    f4();
}

void f4() {
    loop 10;
}
```

The time spent in each function is proportional to the iteration count of the loops. The loops in f1, f2, f3 and f4 are defined to split the total execution time of the application and thus the expected numbers of samples, in a known manner. 30% in f1, 20% in f2, 40% in f3, and 10% in f4, while main and foo are negligible.



Function	Hint	Module	Samples(self)
f3	⟲	call_chain.exe	1,644
f1		call_chain.exe	1,230
f2		call_chain.exe	829
f1		call_chain.exe	829
main		call_chain.exe	829
f4		call_chain.exe	421
			4,124

Limit 20 Process All Thread All Module: All

Function	Hint	Module	Samples(self)	Samples(total)
main		call_chain.exe	1,230	3,703
f1		call_chain.exe	1,230	3,703
f3	⟲	call_chain.exe	1,644	1,644
f2		call_chain.exe	829	829

Figure 3: Statistical Call Graph results

The top hotspot display shows that the self samples are in the ratio of 4:3:2:1 (Figure 3). The call stack expanded from f2 shows f1 and main as its callers. The four hotspots are all highlighted and the total sample count for them is shown below as 4124 samples. An important thing to note is that 829 samples for main here does not mean that we had 829 samples in the main() itself. It is all about samples in f2: we had 829 samples in f2, and for all those samples we had f1 and main as callers.

Note the pull down menus for process, thread, and module filtering of the data displayed in the hotspot view. This greatly simplifies use of the view. This technique is common to hotspot displays for all the collection modes.

The four functions were highlighted, one by one, by clicking the left mouse button and holding down the CTRL key. As the last function selected was f1, it is displayed in the Caller/Callee view. The call chain is expanded in both directions around f1 with callers of f1 shown above it and its callees shown below it. The total number of samples for f1 is equal to its self samples plus the total number of samples for the functions it calls, `f2` and `f3`. So we get 3703 total samples for `f1` as 1230 plus 1644 plus 829. The total for `f3` is equal to the total number of self samples for `f3` and `f4`. As `main` is the caller of `f1` it inherits the self and total times associated with `f1`.

Functions `foo` and `main` are not visible in the hotspot view because no samples occurred within their code ranges. Statistical Call Graph doesn't capture every call the program made. There is another collection technique called Exact Call Graph that instruments all functions in the program and can collect information about each call. However, this method has a much higher overhead. It is also very intrusive and distorts the execution of parallel programs making it impossible to map the results of this analysis to the behavior of the original program. Hence, it has a limited scope of applicability and is not always relevant for parallelization tasks. Exact Call Graph has the advantage of providing function call counts; to provide this useful information Intel PTU has a special Basic Call Count configuration.

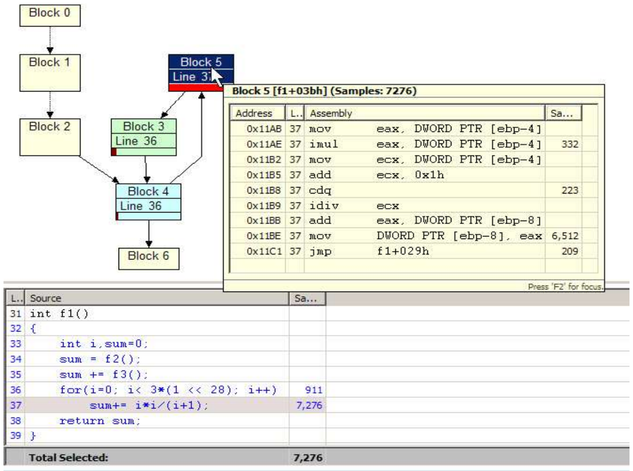
Profile-Guided Loop Analysis

An important step in program parallelization is deciding which parts of the program should be parallelized. Since loops are often good candidates for parallelization, Intel PTU treats loops in a very special way.

Loops are identified by analysis of the binary. This information is then used to generate entries in the hint column. The hints tell you if there is a hot loop in the function and if a hot function was called from a loop. Hot functions called from a loop can be considered for parallelization.

Event-based Sampling

Intel processors have a powerful performance monitoring unit (PMU) that can count and interrupt execution for sampling on a wide variety of performance critical signals (e.g., CPU cycles, instructions retired, last-level cache misses, etc.). Intel PTU has made it easier to use the hundreds of performance events by displaying the sampling data in a logical and convenient manner. The event based sampling hotspot view shows an ordered spreadsheet of all functions, in all modules and processes by default. The spreadsheet can be sorted by any of the collected events. The granularity can be set to module, function (default), basic block, or instruction. A histogram of samples vs. IP can be viewed for any event with a right click option.



Block 5 [f1+03bh] (Samples: 7276)

Address	L...	Assembly	Sa...
0x11AB	37	mov eax, DWORD PTR [ebp-4]	
0x11AE	37	imul eax, DWORD PTR [ebp-4]	332
0x11B2	37	mov ecx, DWORD PTR [ebp-4]	
0x11B5	37	add ecx, 0x1h	
0x11B8	37	cdq	223
0x11B9	37	idiv ecx	
0x11BB	37	add eax, DWORD PTR [ebp-8]	
0x11BE	37	mov DWORD PTR [ebp-8], eax	6,512
0x11C1	37	jmp f1+029h	209

L...	Source	Sa...
31	`int f1()`	
32	`{`	
33	`    int i,sum=0;`	
34	`    sum = f2();`	
35	`    sum += f3();`	
36	`    for(i=0; i< 3*(1 << 28); i++)`	911
37	`        sum+= i*i/(i+1);`	7,276
38	`    return sum;`	
39	`}`	
	Total Selected:	**7,276**

Figure 4: Source view in Intel® PTU

Double clicking on a row will open a source view display that includes a source view spreadsheet and a disassembly spreadsheet organized into units of basic blocks and a control flow graph for the basic blocks (Figure 4). The disassembly spreadsheet can be sorted by the sample totals for the basic blocks to ease the identification of hotspots. The disassembly view can be collapsed to only show the basic block data summary rows for analysis of large complex functions.

There is also the ability to compare two event-based sampling experiments. This is particularly useful for identifying the performance differences from two binaries that have been compiled differently.

As there are hundreds of events, their use must be organized into a methodology. An introduction to the use of the Intel® Core™2 processor PMU is discussed in [6]. A detailed discussion of the cycle accounting methodology on that processor is offered in [7]. The same Web site [8] also contains a number of articles about the use of the Itanium® processor PMU. There are a variety of performance issues associated with parallel execution. Their identification with the Intel Core 2 processor PMU is discussed in [9].

Data Access Analysis

Data access tends to dominate application performance, even in single-threaded execution. Parallel execution only exacerbates this, as the number of execution units available has increased faster than the memory access capability. The actions of the processor, in response to data access requests, can be monitored with performance events counting last-level cache misses, bus traffic, and the like. What has not been generally available is the ability to analyze the application memory access behavior in terms of the data address patterns.

To collect and present performance metrics for accessed data addresses Intel PTU uses advanced features of Intel processors. Intel Itanium processor CPU supports capturing the data access address and access latency directly. On Intel Core 2, Xeon® and Pentium® 4 processors the tool uses *precise* performance events that allow the capture of the values of all the registers at a known value of IP. When coupled with the disassembly of the function, load and store operations can have their target addresses reconstructed. This feature is unique for mass-market CPUs, and for end users, the aspects of the collection mechanism are abstracted by a predefined data access configuration that is used the same way on any of the supported processors. It is also possible to define custom data access configurations using any combination of memory-related events.

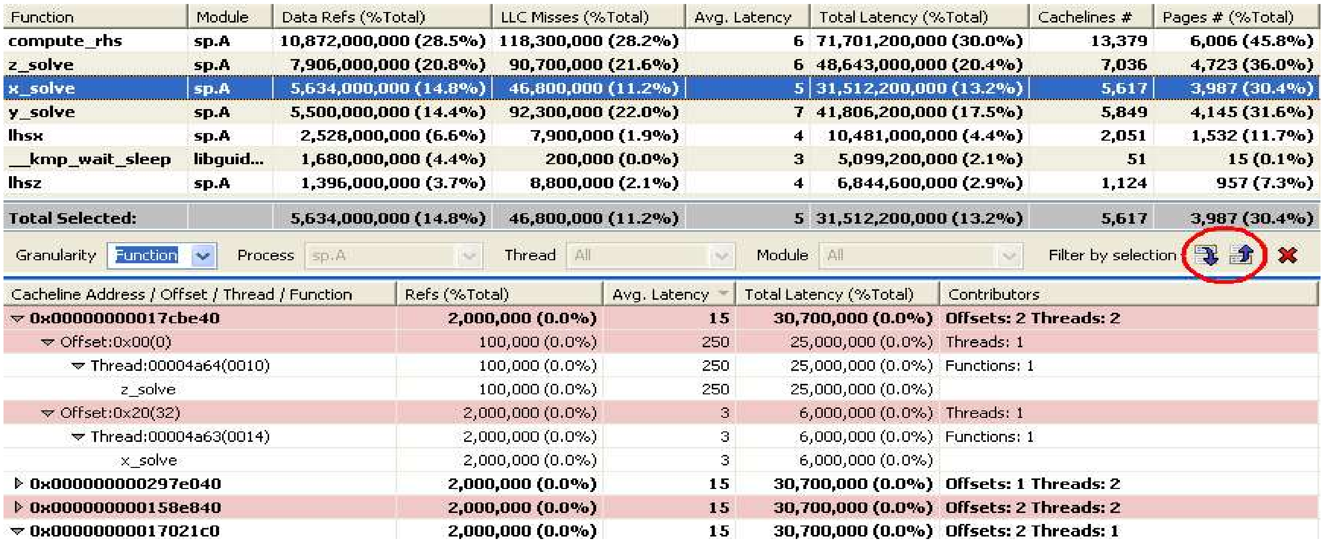
Some of the more obvious objectives of address profiling would be identifying the following:

1. Cachelines that are only partially consumed, increasing memory bandwidth and wasting cache space for no benefit.
2. Cachelines shared by multiple threads unnecessarily (false sharing).
3. Variables (and cachelines) that are being thrashed during synchronization.
4. Arrays of structures that are not organized by usage, resulting in 1 above.
5. Cachelines and variable access resulting in disproportionate access latency.

Today data access analysis provides good help in pinpointing items 2 and 5, while easy identification of the rest of the items is still dependent on future development of the technology.

For data access analysis, Intel PTU provides two hotspot views in both IP and data address (Figure 5). The IP hotspot view is similar to the other hotspot views but has columns associated with data access metrics (average and total latency, reference count, page access count, etc.). The address hotspot view uses a granularity of 64 byte aligned address ranges for IA-32 and Intel® 64 Architecture-based processors and 128 byte aligned ranges on Itanium processors. These correspond to cachelines even though we use virtual rather than physical addresses. Similarly "pages" are usually defined as 4KB aligned ranges and 8KB ranges per the architecture.

The address hotspots can be expanded to show which offsets into the lines were accessed, and which threads and functions accessed the offset. This easily identifies lines that are falsely shared by multiple threads. As a result of the automatic analysis, the tool highlights such lines in pink (note those pink lines in the address hotspot view in Figure 5). However, for now the false-positives are possible, although we hope to minimize their number in the future.



Function	Module	Data Refs (%Total)	LLC Misses (%Total)	Avg. Latency	Total Latency (%Total)	Cachelines #	Pages # (%Total)
compute_rhs	sp.A	10,872,000,000 (28.5%)	118,300,000 (28.2%)	6	71,701,200,000 (30.0%)	13,379	6,006 (45.8%)
z_solve	sp.A	7,906,000,000 (20.8%)	90,700,000 (21.6%)	6	48,643,000,000 (20.4%)	7,036	4,723 (36.0%)
x_solve	sp.A	5,634,000,000 (14.8%)	46,800,000 (11.2%)	5	31,512,200,000 (13.2%)	5,617	3,987 (30.4%)
y_solve	sp.A	5,500,000,000 (14.4%)	92,300,000 (22.0%)	7	41,806,200,000 (17.5%)	5,849	4,145 (31.6%)
lhsx	sp.A	2,528,000,000 (6.6%)	7,900,000 (1.9%)	4	10,481,000,000 (4.4%)	2,051	1,532 (11.7%)
__kmp_wait_sleep	libguid...	1,680,000,000 (4.4%)	200,000 (0.0%)	3	5,099,200,000 (2.1%)	51	15 (0.1%)
lhsz	sp.A	1,396,000,000 (3.7%)	8,800,000 (2.1%)	4	6,844,600,000 (2.9%)	1,124	957 (7.3%)
Total Selected:		5,634,000,000 (14.8%)	46,800,000 (11.2%)	5	31,512,200,000 (13.2%)	5,617	3,987 (30.4%)

Granularity: Function Process: sp.A Thread: All Module: All

Cacheline Address / Offset / Thread / Function	Refs (%Total)	Avg. Latency	Total Latency (%Total)	Contributors
▽ 0x000000000017cbe40	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2
▽ Offset:0x00(0)	100,000 (0.0%)	250	25,000,000 (0.0%)	Threads: 1
▽ Thread:00004a64(0010)	100,000 (0.0%)	250	25,000,000 (0.0%)	Functions: 1
z_solve	100,000 (0.0%)	250	25,000,000 (0.0%)	
▽ Offset:0x20(32)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Threads: 1
▽ Thread:00004a63(0014)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Functions: 1
x_solve	2,000,000 (0.0%)	3	6,000,000 (0.0%)	
▷ 0x000000000297e040	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 1 Threads: 2
▷ 0x000000000158e840	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2
▽ 0x000000000017021c0	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 1

Figure 5: Data access analysis views

The two hotspot views (IP and data address) are coupled and a selection in one can be used to filter the display of the other (with control buttons indicated in Figure 5). Thus the user can select a single function, identify which lines it accesses heavily, select a set of those lines, and then see which other functions also access those same lines. This filtering extends down to the source views.

We went through the most important features of Intel PTU. We learned the important concepts provided by the tool (project, profile configuration, and experiment). We found out which collection and analysis capabilities are supported and identified which of them are specifically applicable for parallelization tasks. Now it's time to discuss how what we learned can be applied to solving real-world parallelization problems.

TUNING FOR DATA-LEVEL PARALLELISM

In this section we provide a real tuning example to highlight the capabilities of Intel PTU in real-world software analysis. We start with discovering parallel execution opportunities, and then we analyze the efficiency of parallelization by locating thread interaction and data layout issues. In the course of our analysis we consider the data-level parallelism wherein different data ranges are processed in parallel on a shared-memory multi-processor.

SP Application and Environment

For our example we took the SP code from the NAS 2.3 Benchmark Suite (NPB2.3) [10]. We started by profiling the serial version of SP, then took the OpenMP C implementation, made by the OMNI compiler team [11, 12], analyzed, and tuned it.

SP is a simulated computational fluid dynamics application. The finite difference solution is based on an approximate factorization that decouples the x, y, and z dimensions [13]. The data set we use is class A, for which a problem size is equal to 64. The simulation is done in 400 high-level iterations over time. The main loop contains the following calls:

```
for (it=1; it<=niter; it++)  {
    compute_rhs();
    txinrv();
    x_solve();
    y_solve();
    z_solve():
    add();
}
```

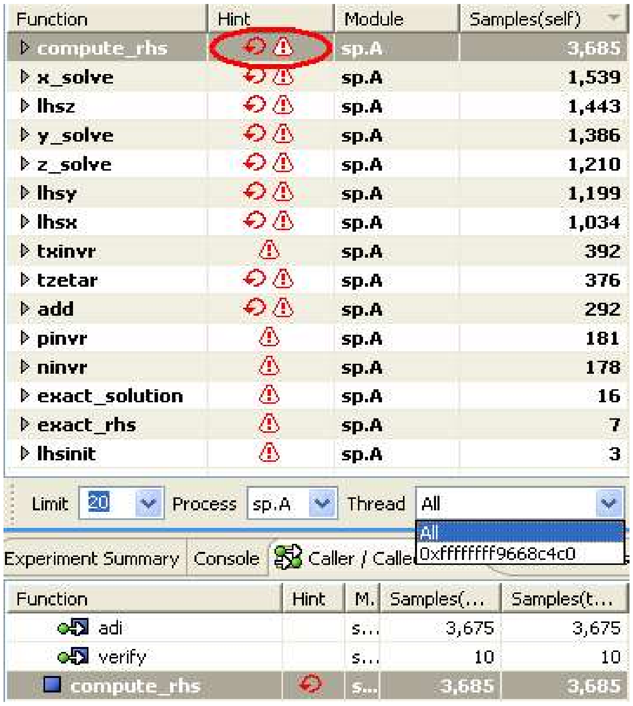
Where `x_solve` calls `lhsx` and `ninvr`; `y_solve` – `lhsy` and `pinvr`; `z_solve` – `lhsz` and `tzetar`.

At each iteration, SP re-calculates a number of three-(64x64x64) and four-dimensional (5x64x64x64) arrays consisting of double precision floating-point numbers and consuming ~76 Mb of the memory space in total.

Our environment was Red Hat Linux* 3.0 Update 8 running on a 2.66 GHz Quad-Core Intel® Xeon® processor 5300[1] series system. This system had eight cores configured in four paired CPUs, with two such pairs per package. Each CPU had a 4Mb L2 cache. The application was compiled using the Intel® Compiler 10.0 with options "-O3 –openmp –g."

Identifying Synchronization Overhead Using Statistical Call Graph

We started with the Basic Statistical Call Graph and Loop Analysis to understand the behavior of the serial version of SP. As can be seen in Figure 6, the tool identified a number of hotspot functions (most of the samples reside in `compute_rhs`). These hotspots have significant numbers of samples that fall in loops inside the hotspots (circled arrow icons). In addition every hotspot is called from within a loop (exclamation mark icons).



Function	Hint	Module	Samples(self)
▷ compute_rhs	⟲ ⚠	sp.A	3,685
▷ x_solve	⟲ ⚠	sp.A	1,539
▷ lhsz	⟲ ⚠	sp.A	1,443
▷ y_solve	⟲ ⚠	sp.A	1,386
▷ z_solve	⟲ ⚠	sp.A	1,210
▷ lhsy	⟲ ⚠	sp.A	1,199
▷ lhsx	⟲ ⚠	sp.A	1,034
▷ txinvr	⚠	sp.A	392
▷ tzetar	⟲ ⚠	sp.A	376
▷ add	⟲ ⚠	sp.A	292
▷ pinvr	⚠	sp.A	181
▷ ninvr	⚠	sp.A	178
▷ exact_solution	⚠	sp.A	16
▷ exact_rhs	⚠	sp.A	7
▷ lhsinit	⚠	sp.A	3

Limit 20 Process sp.A Thread All

All
0xffffffff9668c4c0

Function	Hint	M.	Samples(...	Samples(t...
adi		s...	3,675	3,675
verify		s...	10	10
compute_rhs	⟲	s...	3,685	3,685

Figure 6: SP serial version hotspots in Statistical Call Graph display

Inspection of the source for the hotspot functions suggests that we cannot parallelize the program in question

[1] Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

assigning a different thread to every time iteration (that is, trying to multi-thread the loop surrounding the hotspot calls), because every new time iteration depends on the arrays produced by the previous iterations. Instead, we can employ data decomposition and assign multiple threads to different iterations of the loops inside the hotspots. For such an approach, OpenMP [11] is the obvious choice.

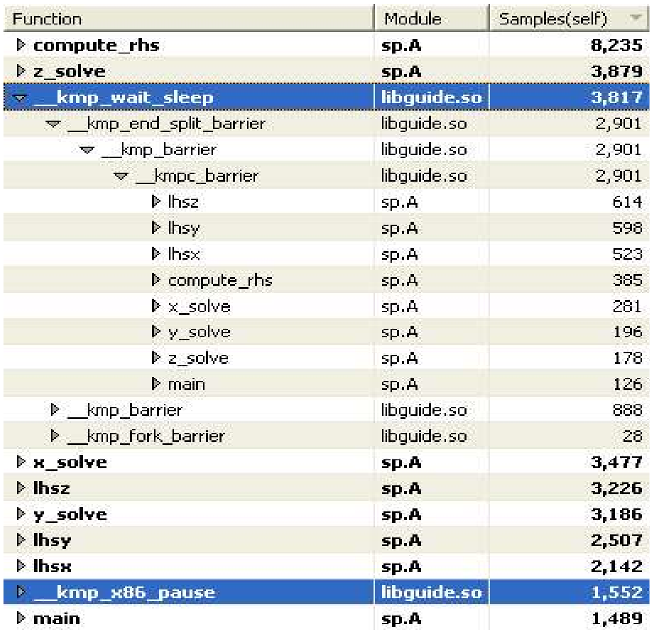
The timing of the parallel version of SP from the OMNI package [12] when running from two to eight threads is shown in Table 1. The two-threaded version's execution time decreased from 130 seconds to 91 seconds.

Table 1: SP OpenMP execution time and relative __kmpc_barrier contribution for different numbers of threads

#of threads	1	2	4	8
SP OpenMP execution time (sec)	130	91	92	94
Total __kmpc_barrier (%)	N/A	16	22	27

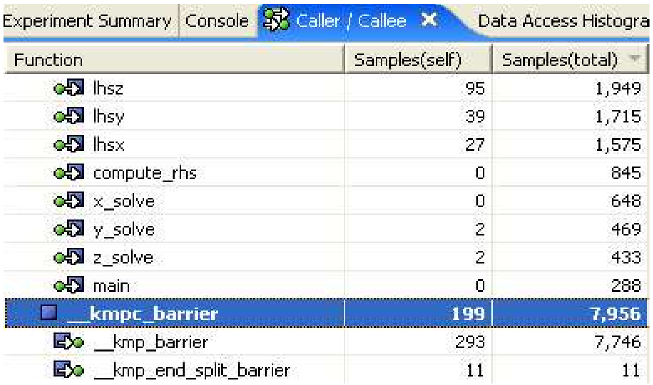
However, running the code with four threads or more shows no additional performance gain. This clearly indicates that there are some problems with SP OpenMP implementation.

The Statistical Call Graph profile for the 4-thread execution (Figure 7) shows that one of the main hotspots, namely the `__kmp_wait_sleep` function, belongs to the `libguide` library. Another substantial hotspot that belongs to `libguide` is `__kmp_x86_pause`. These hotspots all have the `__kmpc_barrier` function on their stacks. `__kmpc_barrier` in turn is called from many SP functions. This can be seen either from the expanded stack of the `__kmp_wait_sleep` hotspot (Figure 7) or, in an aggregated form, in the caller-callee view (Figure 8). `__kmpc_barrier` is dominantly called from the `lhsx`, `lhsy` and `lhsz` functions as the total number of samples for these three functions clearly account for the majority of the time (Figure 8).



Function	Module	Samples(self)
▷ **compute_rhs**	**sp.A**	**8,235**
▷ **z_solve**	**sp.A**	**3,879**
▽ **__kmp_wait_sleep**	**libguide.so**	**3,817**
▽ __kmp_end_split_barrier	libguide.so	2,901
▽ __kmp_barrier	libguide.so	2,901
▽ __kmpc_barrier	libguide.so	2,901
▷ lhsz	sp.A	614
▷ lhsy	sp.A	598
▷ lhsx	sp.A	523
▷ compute_rhs	sp.A	385
▷ x_solve	sp.A	281
▷ y_solve	sp.A	196
▷ z_solve	sp.A	178
▷ main	sp.A	126
▷ __kmp_barrier	libguide.so	888
▷ __kmp_fork_barrier	libguide.so	28
▷ **x_solve**	**sp.A**	**3,477**
▷ **lhsz**	**sp.A**	**3,226**
▷ **y_solve**	**sp.A**	**3,186**
▷ **lhsy**	**sp.A**	**2,507**
▷ **lhsx**	**sp.A**	**2,142**
▷ **__kmp_x86_pause**	**libguide.so**	**1,552**
▷ **main**	**sp.A**	**1,489**

Figure 7: Hotspots for SP OpenMP. Number of threads = 4. The partial stack for the `__kmp_wait_sleep` hotspot is shown.



Function	Samples(self)	Samples(total)
lhsz	95	1,949
lhsy	39	1,715
lhsx	27	1,575
compute_rhs	0	845
x_solve	0	648
y_solve	2	469
z_solve	2	433
main	0	288
__kmpc_barrier	**199**	**7,956**
__kmp_barrier	293	7,746
__kmp_end_split_barrier	11	11

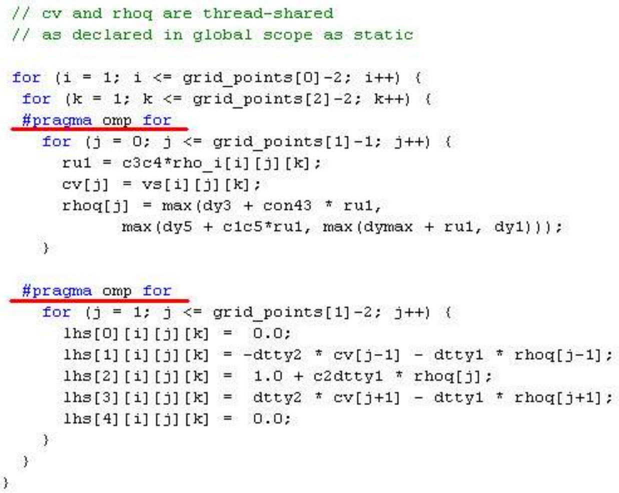
Figure 8: Caller/callee view for SP OpenMP with `__kmpc_barrier` as a target function. This view is useful in evaluating an aggregated target contribution and the relative contributions of its callers.

The data for the `__kmpc_barrier` itself and its callees contribution are summarized in Table 1. The tables show that the number of total samples for `__kmpc_barrier` grows up to one quarter of the application's total number of samples when running with more than four threads. By total samples we mean the self sample count plus the self counts of all the functions down the call chain (callees).

The conclusion from the profiling session is that the initial SP OpenMP implementation doesn't scale because of a significant synchronization overhead exposed as a substantial number of total samples associated with the `__kmpc_barrier function`.

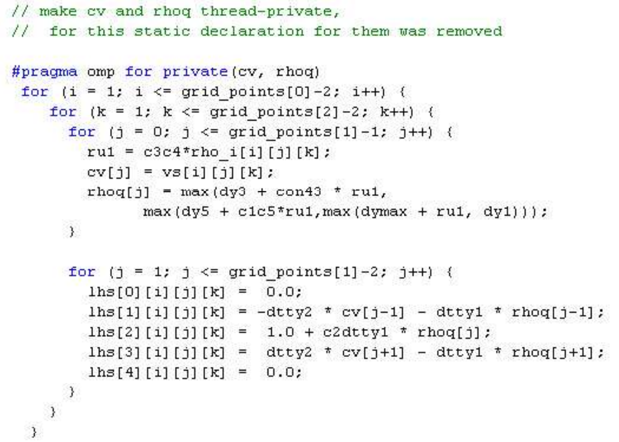
Note that Intel PTU significantly simplifies the identification of the total contribution of a function by automatically synchronizing views for the focus function. Thus, the Caller/Callee view displays aggregated total samples for a function selected in the Hotspot view.

To find the cause of the synchronization overhead, we look at the lhs[*] functions code (Figure 9) and find where the OpenMP "omp for" pragmas are applied. Instead of being applied to the outer loops, they are applied to the inner loop, decomposing the leading dimensions of the multi-dimensional arrays and causing a considerable overhead at an implicit barrier.



```
 // cv and rhoq are thread-shared
 // as declared in global scope as static

 for (i = 1; i <= grid_points[0]-2; i++) {
  for (k = 1; k <= grid_points[2]-2; k++) {
  #pragma omp for
    for (j = 0; j <= grid_points[1]-1; j++) {
      ru1 = c3c4*rho_i[i][j][k];
      cv[j] = vs[i][j][k];
      rhoq[j] = max(dy3 + con43 * ru1,
            max(dy5 + c1c5*ru1, max(dymax + ru1, dy1)));
    }

  #pragma omp for
    for (j = 1; j <= grid_points[1]-2; j++) {
      lhs[0][i][j][k] =  0.0;
      lhs[1][i][j][k] = -dtty2 * cv[j-1] - dtty1 * rhoq[j-1];
      lhs[2][i][j][k] =  1.0 + c2dtty1 * rhoq[j];
      lhs[3][i][j][k] =  dtty2 * cv[j+1] - dtty1 * rhoq[j+1];
      lhs[4][i][j][k] =  0.0;
    }
  }
}
```

Figure 9: A code fragment from the `lhsy` function causing barrier overhead



```
// make cv and rhoq thread-private,
//  for this static declaration for them was removed

#pragma omp for private(cv, rhoq)
 for (i = 1; i <= grid_points[0]-2; i++) {
    for (k = 1; k <= grid_points[2]-2; k++) {
      for (j = 0; j <= grid_points[1]-1; j++) {
        ru1 = c3c4*rho_i[i][j][k];
        cv[j] = vs[i][j][k];
        rhoq[j] = max(dy3 + con43 * ru1,
              max(dy5 + c1c5*ru1,max(dymax + ru1, dy1)));
      }

      for (j = 1; j <= grid_points[1]-2; j++) {
        lhs[0][i][j][k] =  0.0;
        lhs[1][i][j][k] = -dtty2 * cv[j-1] - dtty1 * rhoq[j-1];
        lhs[2][i][j][k] =  1.0 + c2dtty1 * rhoq[j];
        lhs[3][i][j][k] =  dtty2 * cv[j+1] - dtty1 * rhoq[j+1];
        lhs[4][i][j][k] =  0.0;
      }
    }
  }
```

Figure 10: An optimized code fragment at the `lhsy` function

Similar problems were observed in other functions where the "omp for" pragmas were applied to the middle loop of three nested loops. We modified the initial SP OpenMP implementation by making changes to `lhs*` and `*_solve` functions so that the "omp for" pragmas were properly applied to the outermost loop. Further, we merged some separate loops under one "omp for" pragma. We also had to privatize several variables as part of the changes to ensure the correctness of the program.

The improved version of the same non-optimal code fragment (from Figure 9) is shown in Figure 10. We refer to this version of the SP code as "SP OpenMP Opt."

Data Layout Analysis Using Sampling and Data Access Profiling

However, while the issue of the large barrier overhead was fixed by these modifications, the overall performance and scaling did not improve much beyond two threads (see Table 2).

Table 2: Execution time for SP OpenMP initial version and optimized version (time is in seconds)

#of threads	1	2	4	8
SP OpenMP initial	130	91	92	94
SP OpenMP Opt	130	91	78	72

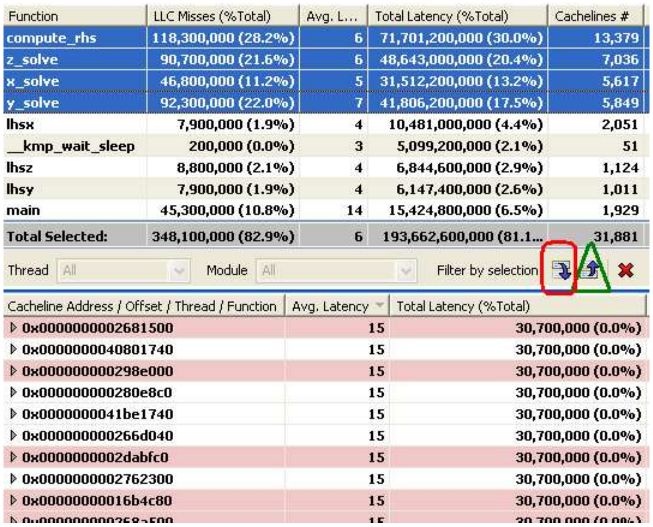
Since the SP application uses ~76 Mb of data space and our system has only 16 Mb of shared L2 cache, the memory usage approach might be the reason for the poor improvement in scaling. To prove this we launched sampling, and we collected the MEM_LOAD_RETIERED.L2_LINE_MISS event for SP OpenMP Opt running with thread numbers 1 through 8. The results (summarized in Table 3) clearly indicate that the number of L2 cache line misses grows with the increasing number of threads. Although for the code to be scalable the number of cache misses should remain the same or even decrease.

Profiling runs for four and eight threads reveal that `compute_rhs` and `z_solve` functions are the main hotspots, contributing ~28% and ~12% L2 cache line misses, respectively, in both runs. The other main hotspots are the `x_solve` and `y_solve` functions.

Table 3: Count of the MEM_LOAD_RETIERED.L2_LINE_MISS event for the SP OpenMP Opt code

# of threads	1	2	4	8
Event count	3.5E+08	3.3E+08	4.1E+08	6.3E+08

The Data Profiling analysis for the four- and eight-thread runs confirms the same functions as the memory access bottlenecks. The Data Access Display shows that `compute_rhs`, `z_solve`, `y_solve` and `x_solve` functions are also hotspots in terms of Last Level Cache (LLC) misses, Total Latency, and Cachelines accessed (Figure 11).



Function	LLC Misses (%Total)	Avg. L...	Total Latency (%Total)	Cachelines #
compute_rhs	118,300,000 (28.2%)	6	71,701,200,000 (30.0%)	13,379
z_solve	90,700,000 (21.6%)	6	48,643,000,000 (20.4%)	7,036
x_solve	46,800,000 (11.2%)	5	31,512,200,000 (13.2%)	5,617
y_solve	92,300,000 (22.0%)	7	41,806,200,000 (17.5%)	5,849
lhsx	7,900,000 (1.9%)	4	10,481,000,000 (4.4%)	2,051
__kmp_wait_sleep	200,000 (0.0%)	3	5,099,200,000 (2.1%)	51
lhsz	8,800,000 (2.1%)	4	6,844,600,000 (2.9%)	1,124
lhsy	7,900,000 (1.9%)	4	6,147,400,000 (2.6%)	1,011
main	45,300,000 (10.8%)	14	15,424,800,000 (6.5%)	1,929
Total Selected:	348,100,000 (82.9%)	6	193,662,600,000 (81.1...	31,881

Cacheline Address / Offset / Thread / Function	Avg. Latency	Total Latency (%Total)
▷ 0x0000000002681500	15	30,700,000 (0.0%)
▷ 0x0000000040801740	15	30,700,000 (0.0%)
▷ 0x000000000298e000	15	30,700,000 (0.0%)
▷ 0x000000000280e8c0	15	30,700,000 (0.0%)
▷ 0x0000000041be1740	15	30,700,000 (0.0%)
▷ 0x000000000266d040	15	30,700,000 (0.0%)
▷ 0x0000000002dabfc0	15	30,700,000 (0.0%)
▷ 0x0000000002762300	15	30,700,000 (0.0%)
▷ 0x00000000016b4c80	15	30,700,000 (0.0%)
▷ 0x000000000268a500	15	30,700,000 (0.0%)

Figure 11: Main hotspots for the SP OpenMP Opt 8 threads run in the Data Access Display. The figure also illustrates how to filter the cachelines accessed by selected functions.

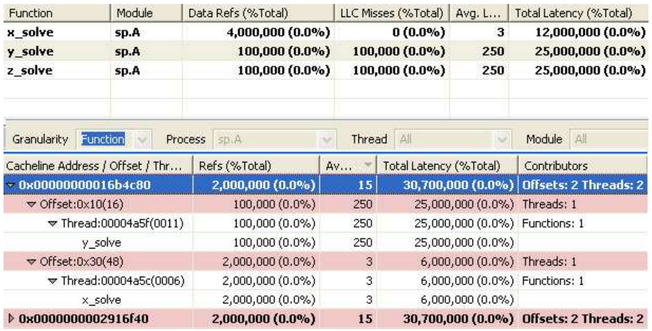
The reason for the growing number of cache misses (refer to Table 3) for four- and eight-thread runs might be interfering data accesses. The data access profile allows us to investigate if there are access contention issues for the cachelines used in the hotspots `compute_rhs`, `z_solve`, `y_solve`, and `x_solve`, particularly those caused by the threads running on different cores but accessing the same cachelines. This will show if there are any contentious lines associated with high average latencies and accessed by several threads.

To explore contention issues we ran the SP OpenMP Opt version with eight threads, bounding each thread to a distinct core using the KMP_AFFINITY environment variable supported by the Intel Compiler OpenMP run-time library. The execution time and hotspots do not change with respect to the non-bound run.

In data access view we select the hotspot functions and use the "Filter by Selection in Code Hotspots" button (circled in Figure 11), to display only the cachelines accessed by these functions.

A number of filtered cachelines are marked in pink as likely suffering from false-sharing. But false-sharing (as well as true-sharing) is a particular case of thread access contention. Consequently, we sort cachelines by average latency and select the high latency lines. We then filter back either on a specific cacheline or a few of them to identify the functions that are associated with the contention. This is done by using the "Filter by Selection in Data Hotspots" button (triangled in Figure 11).

We found a number of cases (one example is in Figure 12) where the same cachelines were accessed from different threads by the functions `compute_rhs` & `x_solve`, `x_solve` & `y_solve`, `y_solve` & `z_solve`. Specifically, Figure 12 demonstrates that the same highlighted cacheline was accessed by the different threads in the `x_solve` and `y_solve` functions. The second access (by `y_solve`, as it called after `x_solve` in the code) is associated with the high latency (250 cycles) equal to an L2 miss penalty.



Function	Module	Data Refs (%Total)	LLC Misses (%Total)	Avg. L...	Total Latency (%Total)
x_solve	sp.A	4,000,000 (0.0%)	0 (0.0%)	3	12,000,000 (0.0%)
y_solve	sp.A	100,000 (0.0%)	100,000 (0.0%)	250	25,000,000 (0.0%)
z_solve	sp.A	100,000 (0.0%)	100,000 (0.0%)	250	25,000,000 (0.0%)

Cacheline Address / Offset / Thr...	Refs (%Total)	Av...	Total Latency (%Total)	Contributors
▽ 0x00000000016b4c80	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2
▽ Offset:0x10(16)	100,000 (0.0%)	250	25,000,000 (0.0%)	Threads: 1
▽ Thread:00004a5f(0011)	100,000 (0.0%)	250	25,000,000 (0.0%)	Functions: 1
y_solve	100,000 (0.0%)	250	25,000,000 (0.0%)	
▽ Offset:0x30(48)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Threads: 1
▽ Thread:00004a5c(0006)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Functions: 1
x_solve	2,000,000 (0.0%)	3	6,000,000 (0.0%)	
▷ 0x0000000002916f40	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2

Figure 12: The IP hotspot view filtered by the selected cacheline

We drill down from the filtered hotspot view to the source view (now only displaying the filtered accesses) of the functions, e.g., `x_solve` and `y_solve` ones, to identify the source lines that generated the access contention.

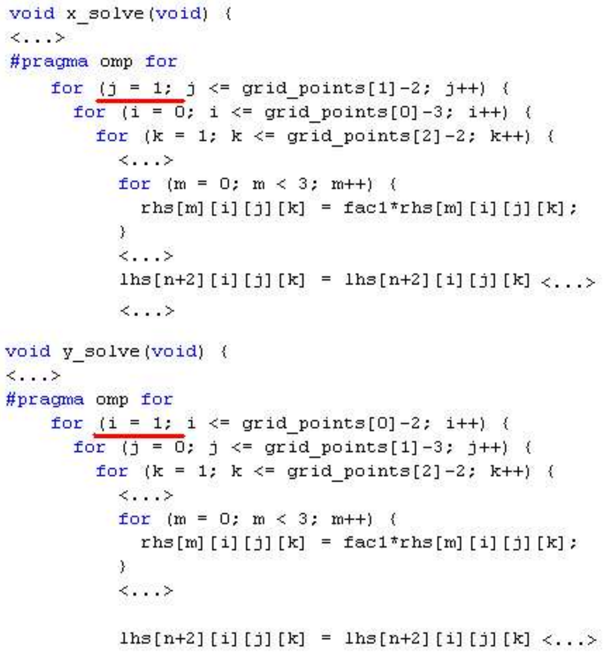
The source code identified by the access counts on these lines in turn identifies a number of cache contention patterns. Figure 13 displays the typical one we discovered.

In this case the `x_solve` function writes to the elements of the arrays `rhs` and `lhs`, and the `y_solve` function reads from them. The "`omp for`" pragma is placed in such a way that the data decomposition of these arrays is different in these two function fragments. In `x_solve` the decomposition, over the third index of `rhs`, causes thread_1 to write into `rhs[*][*][T1_range][*]`, thread_2 writes into `rhs[*][*][T2_range][*]` and so on. While in `y_solve`, the decomposition is over the second index, so thread_1 here reads from `rhs[*][T1_range][*][*]`. This results in multiple cores having to shuffle the cachelines between themselves as they execute `x_solve` and then `y_solve`. This in turn results in a large number of load-driven cache misses and the resulting execution stalls.

We didn't go further with optimizing the SP code since our purpose was just to demonstrate how an application using data parallelism is analyzed and tuned with Intel PTU.

Possible ways for further optimization could be code transformations to make the "`omp for`" pragmas apply to the outermost loops, iterating over the same dimension indices. This would decrease the shuffling of the cachelines between cores and thereby improve the performance.

It would be also useful to consider decreasing some array sizes (to apply data blocking optimization), as described in [13]. This would bring an even bigger performance gain due to a more efficient cache usage.



```
void x_solve(void) {
<...>
#pragma omp for
    for (j = 1; j <= grid_points[1]-2; j++) {
      for (i = 0; i <= grid_points[0]-3; i++) {
        for (k = 1; k <= grid_points[2]-2; k++) {
          <...>
          for (m = 0; m < 3; m++) {
            rhs[m][i][j][k] = fac1*rhs[m][i][j][k];
          }
          <...>
          lhs[n+2][i][j][k] = lhs[n+2][i][j][k] <...>
          <...>

void y_solve(void) {
<...>
#pragma omp for
    for (i = 1; i <= grid_points[0]-2; i++) {
      for (j = 0; j <= grid_points[1]-3; j++) {
        for (k = 1; k <= grid_points[2]-2; k++) {
          <...>
          for (m = 0; m < 3; m++) {
            rhs[m][i][j][k] = fac1*rhs[m][i][j][k];
          }
          <...>

          lhs[n+2][i][j][k] = lhs[n+2][i][j][k] <...>
```

Figure 13: Code fragments causing cacheline contention

In this section we have shown that Statistical Call Graph analysis may be very helpful in the initial stages of parallel code tuning. Proceeding with the analysis requires some knowledge of the processor architecture to identify the hardware events to collect and to interpret the collected data. Advanced scalability estimations can hardly be performed without the help of data access profiling whose automatic analysis and flexible filtering interface enable pinpointing of such problems as cache contention (particularly false-sharing) and high latency loads at the source code level.

CHALLENGES AND FUTURE DIRECTIONS

Extensibility was among the primary design concepts of the Intel PTU architecture, which may enable us to integrate more advanced profiling techniques in the future, many of which we can already define and describe.

The first step that we would like to take in the near future is to extend the Statistical Call Graph (which is now time-based in Intel PTU) to also use rich event-based sampling capabilities. The major advantages of this are expected to be as follows:

- Increased sampling granularity (as the sampling interval will no longer be limited by the operating system timer resolution and task scheduler properties).
- Higher correlation of the sampled execution paths with the architectural characteristics of a computer system.

In data profiling, a unique problem is dealing with arrays of large structures. Being able to display the access pattern in terms of the structure size granularity allows the user to split the structures by usage, reducing bandwidth and increasing cache utilization efficiency.

Another important improvement and a major challenge with regard to data access analysis is the need to operate on the categories that are understandable by a programmer. This means we need to switch from raw addresses (which may mean anything) to the actual variable names, allocation blocks, and so on.

A very promising technology that we are also going to implement is the ability to handle the lowest-level operating system task context switches. This should enable the retrieval of information about thread synchronization patterns, excessive synchronization, overall processor utilization by multiple threads, thread migration between processors, thread switch overhead, and other characteristics that are vital for a detailed analysis of heavily threaded, multi-component applications.

The above described data collection challenges and improvements necessitate changes in the visualization as well. Thus, we would like to introduce a timeline view, the natural representation of thread activity over time. The timeline will reflect the state of threads, their location with respect to processors and cores, and the actual performance characteristics for each thread activity point in time. The overtime representation is supposed to facilitate intuitive understanding of the logic of a parallel program and thread state transition patterns; and it may help to determine distinct phases within the program's operation flow. Most importantly, the timeline is designed to be fully integrated into the rest of the existing views to simplify navigation, incorporate new cross-filtering modes, and make it possible to quickly obtain aggregated characteristics for each thread execution point, state, or phase.

CONCLUSION

The performance analysis features available in the Intel Performance Tuning Utility assist at virtually every stage of both parallel and sequential software performance tuning and may be extremely helpful at the preliminary stages of determining parallelization strategies.

We discussed data-level decomposition strategy in real program examples and illustrated how the efficiency of a parallel implementation can be estimated, and which steps should be performed to optimize a parallel program using the Intel Performance Tuning Utility.

Being easy to use and powerful at the same time, Intel PTU is growing its customer base inside Intel for solving today's problems and serving as a vehicle for exploring new features for future commercial tools.

We plan on improving Intel PTU with newer performance data-collection techniques and analysis models to keep pace with user needs and modern processor architecture developments. Intel PTU is available for an external download from Whatif.intel.com Web site [5].

REFERENCES

[1] Intel VTune™ Performance Analyzer, at http://www3.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

[2] Sun Studio Performance Analyzer, at http://developers.sun.com/sunstudio/

[3] Optimizing with Shark, at http://developer.apple.com/tools/shark_optimize.html

[4] Gprof, at http://www.gnu.org/software/binutils/binutils.html

[5] Intel® What If site, at http://Whatif.intel.com

[6] D. Levinthal, "Introduction to Performance Analysis on Intel Core 2 Duo Processors," at http://www.devx.com/go-parallel/Link/33305

[7] D. Levinthal, "Execution-based Cycle Accounting on Intel Core 2 Processors," at http://www.devx.com/go-parallel/Link/33315

[8] "Go Parallel Web Site," at http://www.devx.com/go-parallel/Door/32532

[9] D. Levinthal, "Analyzing and Resolving Multi-Core Non Scaling on Intel Core 2 Processors," at http://www.devx.com/go-parallel/Link/34762

[10] "NAS Parallel Benchmarks," at http://www.nas.nasa.gov/Resources/Software/npb.html

[11] "Open MP standard," at http://www.openmp.org

[12] "OpenMP C versions of NPB2.3," at http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html

[13] H. Jin, M. Frumkin, J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," at http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf

AUTHORS' BIOGRAPHIES

Alexei Alexandrov is a Senior Software Engineer in the Software and Solutions Group at Intel. His interests include building and designing modern performance analysis tools, large software development, CPU micro-architecture, and performance analysis. Alexei has a Ph.D. degree from the Saratov State Technical University. His e-mail is alexei.alexandrov at intel.com.

Stanislav Bratanov is a Research Engineer in the Software and Solutions Group at Intel. His research interests include multi-processor software platforms, operating system environments, software performance monitoring and analysis systems, and platform-dependent media data coding. He graduated from Nizhniy Novgorod State University, Russia. His e-mail is stanislav.bratanov at intel.com.

Julia Fedorova is a Senior Software Engineer in the Software and Solutions Group at Intel. Her research interests are performance analysis tools, tuning and optimization, and data access analysis. Prior to Intel she worked in the Russian Nuclear Center. Julia has an M. Sc. degree in Computational Physics from the Moscow Engineering-Physical Institute. Her e-mail is julia.fedorova at intel.com.

David Levinthal is a Senior Software Engineer in the Software and Solutions Group at Intel. His research interests include hardware performance events, computer architecture, and software optimization. He holds a Physics degrees from the University of California at Berkeley and Columbia University. He was a Professor of Physics at Florida State University. He has been awarded the DOE OJI award, the NSF PYI award, and a Sloan Foundation Fellowship. His e-mail is david.a.levinthal at intel.com.

Igor Lopatin is a Software Engineer in the Software and Solutions Group at Intel. His research interests include software for multi-core architectures and tools based on dynamic binary instrumentation techniques. He graduated from Nizhny Novgorod State University, Russia. His e-mail is igor.loopatin at intel.com.

Dmitry Ryabtsev is a Senior Software Engineer in the Software and Solutions Group at Intel. He has worked on the VTune Performance Analyzer and currently is

focusing on DAP for Intel PTU. He received his B.S. and M.S. degrees from the Nizhny Novgorod State University, Russia. His e-mail is dimitry.ryabtsev at intel.com.

Parallel Software Development with Intel® Threading Analysis Tools

Lihui Wang, Intel Information Technology, Intel Corporation
Xianchao Xu, Intel Information Technology, Intel Corporation

Index words: Intel® Threading Analysis Tools, multi-core, parallel programming, development cycle, threading methodology

ABSTRACT

While multi-core processors are designed for greater performance with optimal power consumption, the parallel algorithm design and software development that is needed to maximize the performance potential of multi-core systems are much more complicated than those associated with serial computing. Even though parallel computing has long been studied by researchers, there is no general framework to implement parallel programming for different software applications. Programmers face three immediate challenges when applying parallelism to software development: scalability, correctness, and maintainability. Applicable parallel methodology and new software development tools are greatly needed by programmers working in this environment. To keep software applications in sync with the multi-core processors that are becoming mainstream in the marketplace, Intel provides a whole set of threading software products.

A multiple pattern matching algorithm is the core algorithm of the detection engine in the rule-based Intrusion Detection System (IDS). Most of the research on improving the performance of this algorithm is based on serial computing. Actually, the performance of the algorithm can be improved greatly through parallelization on multi-core systems. By threading and tuning a typical multiple pattern matching algorithm, we show how to apply parallel principles during each phase of a generic development cycle while utilizing Intel® Threading Analysis Tools to pinpoint the bottlenecks and thread-safety errors and to improve overall performance. Moreover, we specifically compare the implementation method and performance gain of the Windows* Threading API against that of Intel® Threading Building Blocks (Intel® TBB), when implementing the parallel multiple pattern matching algorithm with the experimental performance data presented.

INTRODUCTION

As multi-core processors become mainstream in the market place, software needs to be parallel to take advantage of multiple cores. However, there is no general framework available to implement parallel programming for different applications to achieve the highest performance gain. Generally implemented with multi-threading, parallel programming is notoriously difficult for developers to design, implement, and debug. In order to make life easier for developers, Intel provides a set of threading tools targeting various phases of the development cycle. In a generic development cycle, program development can be divided into four phases [1]:

- Analysis phase: Profiling the serial version of the program to determine the areas that are suitable for parallel decomposition.
- Design/implementation phase: Examining identified threading candidates, determining the changes that have to be made to the serial version, and converting them to the actual code.
- Debug phase: Ensuring the correctness of the program. Detecting and solving common threading errors such as data race and deadlocks.
- Testing/tuning phase: Validating the correctness of the program and testing its performance. Detecting performance issues and fixing them by improved design or by eliminating bottlenecks.

Intel's threading tools provide aids for developers from performance analysis to implementation and debugging:

- Intel® VTune™ Performance Analyzer [4]. This tool helps developers tune an application to better perform on Intel® architectures. It locates the performance bottlenecks and program hotspots by collecting, sampling, and displaying system-wide data down to specific functions, modules, or instructions. It is usually used during the analysis and tuning phase of the development cycle.

- Intel® Thread Profiler [5]. This tool helps to identify performance bottlenecks in Win32* and OpenMP* threaded software. It detects threading performance issues such as thread overhead and synchronization cost. The profiler is usually used in the tuning phase.
- Intel® Thread Checker [6]. This tool helps to find bugs in Win32 and OpenMP threaded software. It locates threading issues such as race conditions, thread stalls, and potential thread deadlocks. The Intel Thread Checker is usually used during the design and debugging phases.
- Intel Threading Building Blocks (Intel TBB) [7]. This is a threading abstraction library that provides high-level generic implementation of parallel patterns and concurrent data structures [2]. Intel TBB is usually used in the design, implementation, and tuning phases.

In the sections that follow, we first introduce the principles of parallel application design; then we show how to parallelize an application with the help of threading tools during each phase of the development cycle. A multiple pattern matching algorithm is used as an example. We use the Win32 threading API and Intel TBB to implement the parallelism, and we compare the performance of the two.

PRINCIPLES OF PARALLEL APPLICATION DESIGN

Decomposition Techniques

Dividing a computation into smaller computations and assigning these to different processors for execution are two key steps in parallel design. Two of the most common decomposition techniques [3] are functional decomposition and data decomposition [2].

- Functional decomposition is used to introduce concurrency in the problems that can be solved by different independent tasks. All these tasks can run concurrently.
- Data decomposition works best on an application that has a large data structure. By partitioning the data on which the computations are performed, a task is decomposed into smaller tasks to perform computations on each data partition. The tasks performed on the data partitions are usually similar. There are different ways to perform data partitioning: partitioning input/output data or partitioning intermediate data.

Parallel Models

These are some of the commonly used parallel models [3].

- *Data parallel model*. This is one of the simplest parallel models. In this model, the same or similar computations are performed on different data repeatedly. Image processing algorithms that apply a filter to each pixel are a common example of data parallelism. OpenMP is an API that is based on compiler directives that can express a data parallel model.
- *Task parallel model*. In this model, independent works are encapsulated in functions to be mapped to individual threads, which execute asynchronously. Thread libraries (e.g., the Win32 thread API or POSIX* threads) are designed to express task-level concurrency.
- *Hybrid models*. Sometimes, more than one model may be applied to solve one problem, resulting in a hybrid algorithm model. A database is a good example of hybrid models. Tasks like inserting records, sorting, or indexing can be expressed in a task-parallel model, while a database query uses the data-parallel model to perform the same operation on different data.

Amdahl's Law

Amdahl's law provides the theoretical basis to assess the potential benefits of converting a serial application to a parallel one. It predicts the speedup limit of parallelizing an application:

$T_{parallel} = \{(1 – P) + P / N\} * T_{serial} + O_N$

T_{serial}: time to run an application in serial version

P: parallel portion of the process

N: number of processors

O_N: parallel overhead in using N threads

We can predict the speedup by looking at the scalability:

Scalability = $T_{serial} / T_{parallel}$

We can get the theoretical limit of scalability, assuming there is no parallel overhead:

Scalability = $1/ \{(1– P) + P/N\}$

When $N\rightarrow\infty$, Scalability$\rightarrow 1/ (1– P)$

CHALLENGES OF PARALLEL PROGRAMMING

Parallel Overhead

Parallel overhead refers to the amount of time required to coordinate parallel tasks as opposed to doing useful work. Typical parallel overhead includes the time to start/terminate a task, the time to pass messages between tasks, synchronization time, and other extra computation time. When parallelizing a serial application, overhead is inevitable. Developers have to estimate the potential cost and try to avoid unnecessary overhead caused by inefficient design or operations.

Synchronization

Synchronization is necessary in multi-threading programs to prevent race conditions. Synchronization limits parallel efficiency even more than parallel overhead in that it serializes parts of the program. Improper synchronization methods may cause incorrect results from the program. Developers are responsible for pinpointing the shared resources that may cause race conditions in a multi-threaded program, and they are responsible also for adopting proper synchronization structures and methods to make sure resources are accessed in the correct order without inflicting too much of a performance penalty.

Load Balance

Load balance is important in a threaded application because poor load balance causes under utilization of processors. After one task finishes its job on a processor, the processor is idle until new tasks are assigned to it. In order to achieve the optimal performance result, developers need to find out where the imbalance of the work load lies between different threads running on the processors and fix this imbalance by spreading out the work more evenly for each thread.

Granularity

For a task that can be divided and performed concurrently by several subtasks, it is usually more efficient to introduce threads to perform some subtasks. However, there is always a tipping point where performance cannot be improved by dividing a task into smaller-sized tasks (or introducing more threads). The reasons for this are 1) multi-threading causes extra overhead; 2) the degree of concurrency is limited by the number of processors; and 3) for most of the time, one subtask's execution is dependent on another's completion. That is why developers have to decide to what extent they make their application parallel. The bottom line is that the amount of work per each independent task should be sufficient to leverage the threading cost.

MULTIPLE PATTERN MATCHING ALGORITHM

Aho-Corasick_Boyer-Moore Algorithm

The Aho-Corasick_Boyer-Moore (AC_BM) [8] algorithm is an efficient string pattern matching algorithm that is derived from the Boyer-Moore algorithm to improve efficiency when there are multiple string patterns to search against. Multiple string pattern matching is one of the core problems in the rule-based Intrusion Detection System (IDS) such as Snort [9]. The AC_BM algorithm can be used to improve the efficiency of the detection engine of the IDS. The basic idea of the algorithm is that the string patterns are first built into a pattern tree, and then the algorithm slides the pattern tree using bad character and good prefix shifts to find the matches. Figure 1 shows the pseudo code of the AC_BM algorithm. In the code, we use acbm_init() to construct the pattern tree, and we use acbm_search() to compare the given text with the pattern tree.

Serial Implementation

```
pattern_tree *acbm_init(pattern_data
*patterns, int npattern);
int acbm_search(pattern_tree *ptree, unsigned
char *text, int text_len, unsigned int
matched_indexs[], int nmax_index);

int _tmain(int argc, char* argv[])
{
    pattern_data *patterns = NULL;
    int npattern = argc  -2;
        …
/* read app args, file length, and other
  initialization operati ons. */

    ptree = acbm_init(patterns, npattern);

    matched = acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexs,
nmax_index);
    printf("total match is %d\n", gMatched);
    return 0;
}
```

Figure 1: Serial implementation of AC_BM algorith

Before moving on to parallelize the application and tuning the performance, it is better to baseline the performance of the serial version. We test the performance against different workloads based on different file sizes.

System environment:

- OS: Windows XP* SP2;
- CPU: Intel® Core™2 Duo processor 6300[1] @ 1.86 GHz; Memory: 1.00 GB

Figure 2 shows the test result of matching patterns against different file sizes:

Text size	1M	2M	4M	8M	16M	32M
Time (ms)	12.167	24.826	49.327	96.304	191.728	383.353

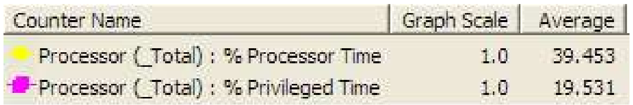
Figure 2: Performance of different file sizes

Analysis

Identifying performance bottlenecks is the first step in improving the performance of an application. For simple applications, it's possible to find the bottleneck by looking at the algorithm and source code, and by analyzing the nature of an application. However, for large and complicated applications, it's difficult to predict performance issues when the bottlenecks are not only dependent on the algorithm, but also on the execution environment such as the processor or memory system. That is why we need a performance analyzing tool.

The Intel VTune Performance Analyzer finds performance bottlenecks by automating the process of data collection with three types of data collectors: sampling, call graph, and counter monitor. These data collectors help find the hotspots and the bottlenecks in the system, application, and micro-architecture levels. We use the VTune Performance Analyzer to profile the serial version of the AC_BM application and uncover the parallel opportunities.

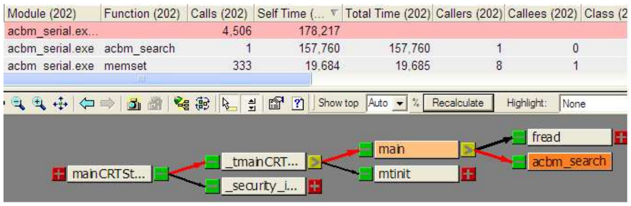
In counter monitor view (as shown in Figure 3), we find that the average processor time is only 39.453%, which means that the processors are not fully utilized; therefore, we can improve the performance by introducing more threads.



Counter Name	Graph Scale	Average
Processor (_Total) : % Processor Time	1.0	39.453
Processor (_Total) : % Privileged Time	1.0	19.531

Figure 3: Counter monitor view

The call graph view (as shown in Figure 4) shows the critical path of the application. It helps to determine which function in the call tree to thread that will result in the best performance gain. Threading a function higher up in the tree results in a coarser-grain level of parallelism, while threading a child function lower in the tree results in a finer-grain level of parallelism. In the graph, we can see that the only function we can thread on the critical path is acbm_search, and acbm_search doesn't have any child function.



Module (202)	Function (202)	Calls (202)	Self Time (...	Total Time (202)	Callers (202)	Callees (202)	Class (2
acbm_serial.ex...		4,506	178,217				
acbm_serial.exe	acbm_search	1	157,760	157,760	1	0	
acbm_serial.exe	memset	333	19,684	19,685	8	1	

Show top: Auto % Highlight: None

Figure 4: Call graph view

Threading Methods

Currently there are several threading strategies: OS threading libraries such as the Win32 API; compiler directives OpenMP; Message passing API (MPI); and the newly introduced Intel TBB. Each method has its own features and there is no simple way to determine which threading method is the best. It mainly depends on the specific application to be threaded.

In the following sections, we parallelize the AC_BM algorithm using the Win32 threading library and Intel TBB, and we compare the performance of the two.

Estimate the Performance Speedup

Before threading the application it is better to estimate the performance speedup of the parallelization: on the one hand, you may know about the upper bound of the performance limit and set the optimization goal for threading and tuning; on the other hand, estimating the performance speedup helps you to determine when to stop and accept the performance gain without trying for more tuning iterations.

In our system, there are two processors, and in the VTune Performance Analyzer sampling view, we find that the acbm_search function takes up 92.90% of CPU time. According to Amdahl's law

$T_{parallel} = \{(1\text{-}0.929) + 0.929/2\}{*}T_{serial} + O_N$;

Suppose $O_N \rightarrow 0$, Scalability → 1.867

So the upper bound of the speedup is 1.867.

[1] Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families.

See www.intel.com/products/processor_number for details.

PARALLELIZATION WITH THE WIN32 THREADING LIBRARY

Decomposition Method

In call graph view, we can see that the acbm_search function is called only once, but it takes up a significant amount of CPU time. The operation of acbm_search is to match the pattern tree against the given file to count the total matches between the patterns and the file. Since different text portions of the file are independent, each portion can be matched against the pattern tree separately. Data decomposition is used to logically divide the file into several parts and to use one thread to match against the patterns and return the matches of each part. Thus for the acbm_search function, these threads only have two points of serialization, i.e., loading file and combining the final result of total matches after all threads are finished.

Implementation

Figure 5 and Figure 6 show the source codes that illustrate the basic operations of threading the algorithm with the Windows threading API.

When splitting the file into several parts, some matched strings may be split into different text portions. So in each search thread, after getting the matched pattern number from acbm_search() for their assigned portion of text, we use acbm_connection() to calculate the matched number of strings on the text boundary.

```
#define THREAD_NUM 4
CRITICAL_SECTION rsCriticalSection;
int gMatched;
int _tmain(int argc, char* argv[])
{
    pattern_data *patterns = NULL;
    int npattern = argc - 2;
    HANDLE hThread[THREAD_NUM];
    DWORD dwThread[THREAD_NUM];
    file_info *finfo[THREAD_NUM];
        …
    /* read app args, file length, read
file,  and other initialization operations.
*/

    ptree = acbm_init(patterns, npattern);

    int step = textLength / THREAD_NUM;

    for(i = 0; i < THREAD_NUM; i++)
    {
        … // initialize finfo[i]
        hThread[i] = CreateThread(
        NULL, 0, ACBMThreadProc, finfo[i],
0,&dwThread[i]);
     }

     WaitForMultipleObjects(THREAD_NUM,
hThread, true, INFINITE);
     printf("total match is %d\n", gMatched);
        return 0;
}
```

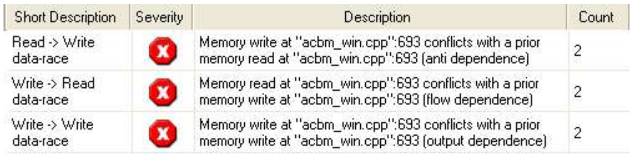
Figure 5: Main() function

```
DWORD WINAPI ACBMThreadProc(LPVOID pParam)
{
    unsigned int matched_indexs[0xffff];
    int nmax_index = 0xffff;

    file_info *finfo = (file_info*)pParam;
    if(finfo == NULL)return 1;

    textLength = finfo->file_len;
    text = finfo->text;

    gMatched += acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexs,
nmax_index);

    …
    /* Ajust the value for the adjacent parts
    of the  divided text portions */
    gMatched += acbm_ connection(…);

    … // clean up code
    return 0;

}
```

Figure 6: AC_BM Search thread

Debugging

When introducing the threads, it is necessary to ensure their correctness. There are some notorious thread bugs, such as data race, stall, and deadlock, and they are usually difficult to detect. In our application, we use the Intel Thread Checker to check whether there are any bugs in the threads.



Short Description	Severity	Description	Count
Read -> Write data-race	✖	Memory write at "acbm_win.cpp":693 conflicts with a prior memory read at "acbm_win.cpp":693 (anti dependence)	2
Write -> Read data-race	✖	Memory read at "acbm_win.cpp":693 conflicts with a prior memory write at "acbm_win.cpp":693 (flow dependence)	2
Write -> Write data-race	✖	Memory write at "acbm_win.cpp":693 conflicts with a prior memory write at "acbm_win.cpp":693 (output dependence)	2

Figure 7: Data race detected by Thread Checker

From the results shown in Figure 7, we can see that there is data race on the result calculating part. In order to avoid incorrect access to gMatched, we use a CRITICAL_SECTION to protect gMatched between different threads.

```
CRITICAL_SECTION rsCriticalSection;
DWORD WINAPI ACBMThreadProc(LPVOID pParam)
{
    …

    EnterCriticalSection(&rsCriticalSection);
    gMatched += acbm_search(ptree, (unsigned
char *)text, textLength, matched_indexs,
nmax_index);
    gMatched += acbm_ connection(…);
    …
    LeaveCriticalSection(&rsCriticalSection);

    …
    return 0;
}
```

Figure 8: Fix the result calculating part

Test and Performance Tuning

After multi-threading the application, we test the performance with a 32M file: the result is *339.553ms*. The scaling performance to the serial version, *383.353ms*, is not very good, which means we need to tune the performance.

Performance tuning involves several steps:

1. Gather performance data.
2. Analyze data and identify issue.
3. Generate alternatives to resolve the issue.
4. Implement enhancements.
5. Test results.

All these steps are performed in one iteration of the performance tuning process. Usually, several iterations are required to get the best performance. Before any tuning process, we need to ensure our program is correct and ready for tuning. Another thing that should be kept in mind is to only make one change for each tuning iteration.

In order to find tuning opportunities for a multi-threaded program, we use the Intel Thread Profiler to gather thread-related data.

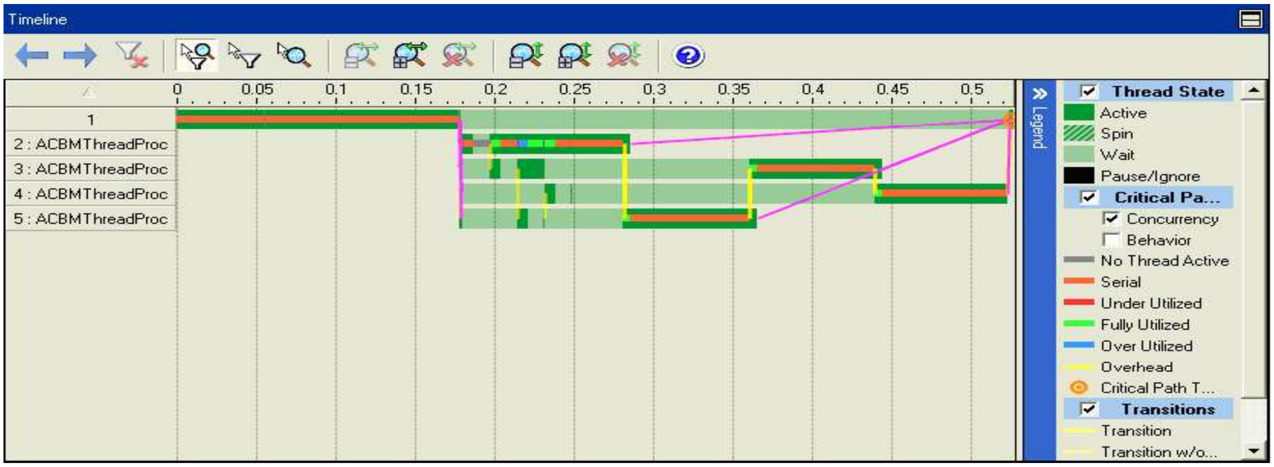


Figure 9: Thread Profiler Timeline view

In the Timeline view (shown in Figure 9), we can see that a large majority of the threads are executed in serial (the orange line indicates serial execution) and there are overhead and transitions between different threads (represented by yellow lines). We can double click one of the yellow lines and it will drill down to the source code where the problem occurs.

```
EnterCriticalSection(&rsCriticalSection);
   gMatched += acbm_search(…);
        …
LeaveCriticalSection(&rsCriticalSection);
```

Figure 10: Computing the search result

The transition and overhead are caused by acquiring access to the critical section. And, since only one thread can get access to the critical section at one time, the code essentially allows only one thread to do the search, which causes the serial problem. We modify the code like this (in Figure 11):

```
int matched = acbm_search(…);
matched += acbm_ connection(…);
EnterCriticalSection(&rsCriticalSection);
   gMatched += matched;
        …
LeaveCriticalSection(&rsCriticalSection);
```

Figure 11: Improved approach to computing the search result

In this way, each thread performs the search independently and only needs to access the critical section when it adds the result to global variable gMatched.

After the modification, we tested the application and found that the time it spent on a 32M file was reduced to *255.500ms*.

Now we have completed one iteration of tuning. We achieved improved performance after this iteration. Next we run the Thread Profiler again to see whether there are any other issues.

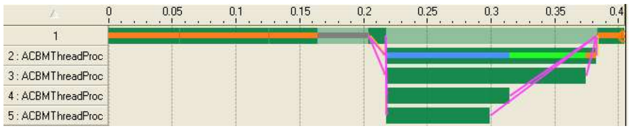


Figure 12: Thread Profiler Timeline view

In Figure 12, we can see that the serial execution of different threads has been solved. Notice that in the timeline view, the thread that ends last still has a small portion of serial execution, and the main thread needs to wait for all threads to exit and join the result. Instead of allowing the main thread to idle, we may let the main thread share a portion of data and do the search too. After the modification, the time spent is reduced to *252.643ms*. See the timeline view in Figure 13.

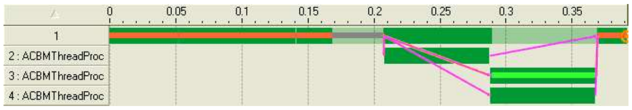


Figure 13: Thread Profiler Timeline view

We can see that there are no more overhead and transitions between different threads; Thread No. 3 is fully utilized during its life cycle. However, there are actually only two threads running at a time; the other two threads are just waiting. Remember, the execution environment is a dual-core system so there won’t be more than two threads running at one time in a CPU. Delegating more than two threads may only incur extra overhead, due to thread creation and scheduling. We therefore change the `THREAD_NUM` to 2. The time spent is now reduced to *248.702ms.*

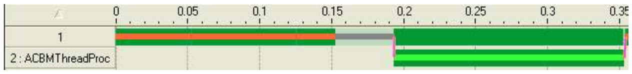


Figure 14: Thread Profiler Timeline view

Now the threads are all well utilized, and Figure 14 shows that there are no transitions and imbalances any more. The performance scalability compared to serial processing is *383.353ms/248.702ms = 1.54* (the ideal is 1.867). Since the performance for the recent tuning iterations is not improved dramatically, we decide to accept the improvement.

From the tuning process presented above, we can see that the number of threads impacts the overall performance. In our example, the application works best when there are two threads, and the application happens to run on a dual core system. However, it doesn't mean that all applications running on a dual core system have the best performance when they have two threads. In the AC_BM algorithm, ACBMThreadProc only does some searches and calculations. However, for some other applications, the thread may need to wait for some resources, such as network connection or data loading. If this is the case, such a thread can relinquish the CPU and let another thread that is not waiting for resources to have the CPU. So in the case of this application, it may perform better if it has more threads than the number of CPU cores. There is no easy way to determine the number of threads for an application especially when the application is large and complex. Further, it is not easy to determine which task should be assigned to each thread to get the best performance. It is better to use the Thread Profiler to get the execution information about each thread and discover the tuning opportunities.

PARALLEL COMPUTING WITH INTEL® TBB

Challenges of Parallelizing with the Win32 Threading Library

While the Win32 threading library gives programmers great flexibility by giving them detailed control over threads, the library brings challenges for them too: programmers must rewrite the common parallel programming utilities; troubleshoot threading bugs, and try to avoid thread-safe issues. There is no way to maintain load balance and achieve good scalability unless the programmers perform these tasks manually, and these tasks require a comprehensive understanding of the threading method, application, and the features of the system.

Intel Threading Building Blocks

To make it easier for programmers to realize parallelism, Intel TBB provides a high-level generic implementation of parallel patterns and concurrent data structures. With TBB, you specify task patterns instead of threads. Logical tasks are mapped automatically onto physical threads, thus hiding the complexity of operating system threads.

Implementation

According to the analysis detailed in the "Parallel with the Win32 Threading Library" section, the AC_BM algorithm can be parallelized by using data decomposition. We can separate the text into several blocks, calculate the matched number in each block, and sum the total matched number from all the blocks. Intel TBB provides the template parallel_reduce to do this kind of decomposition.

Figure 15 and Figure 16 show the source code of the implementation with Intel TBB. Figure 15 shows the definition of *Search* task that performs the pattern matching operations. Figure 16 shows the main() function.

```
struct Search {
   int value;
   Search() : value(0) {}
   Search( Search& s, split ) {value = 0;}
   void operator()( const blocked_range<int>&
range ) {
   int textLength = range.end() -
range.begin();
   unsigned int matched_indexs[0xffff],
connection_matched_indexs[0xffff];
   int matched, connection_matched;
   int nmax_index = 0xffff;

   matched = acbm_search(ptree, (unsigned
char *)(text+range.begin()), textLength,
matched_indexs, nmax_index);
   value += matched;
    /* Ajust the value for the adjacent part
    of the  divided text portions */
   if(range.begin() > 0){
   textLength = (ptree->max_depth - 1) * 2;
   connection_matched = acbm_ connection(…);
   value += connection_matched;
        }
   }
   void join( Search& rhs ) {
      value += rhs.value;
   }
};
```

Figure 15: AC_BM search task

```
int main(int argc, char* argv[])
{
   // Initilization and read file…
   …
   task_scheduler_init init;
   Search total;
   int grainsize;
   //Set the grainsize
   …

  parallel_reduce( blocked_range<int>( 0,
textLength, grainsize), total );
   matched = total.value;
   printf("total match is %d\n", matched);
        return 0;
}
```

Figure 16: Main() function

```
Template<typename Range, typename Body>

void   parallel_reduce(const   Range&   range,
Body& body);
```

Figure 17: Declaration of `parallel_reduce()`

Figure 17 shows the declaration of the parallel_reduce() function. A parallel_reduce performs parallel reduction of body over each value in range. Grain size specifies the biggest range considered indivisible, i.e., Intel TBB splits the range into two subranges, if the size of the range exceeds the grain size. A too-small grain size may cause scheduling overhead within the loop templates and counteract the speedup gained from parallelism. A too large grain size, on the other hand, may unnecessarily limit parallelism. For example, if the grain size is so large that the range cannot be split, then the application will not be parallelized. Therefore, it is important to set the proper grain size. Normally, setting grain size to 10,000 is high enough to amortize the scheduler overhead; the correct size also depends on the ratio between the task number and processor number.

From the result of the Intel Thread Checker, there are no data races and thread-safety errors. Figure 18 shows that there are no transitions or imbalances between different threads either.

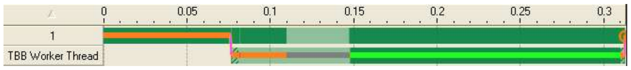


Figure 18: Thread Profiler Timeline view

Figure 19 shows the performance for different grain sizes when the test file is 32M. The application achieves the best performance when the grain size equals 8M.

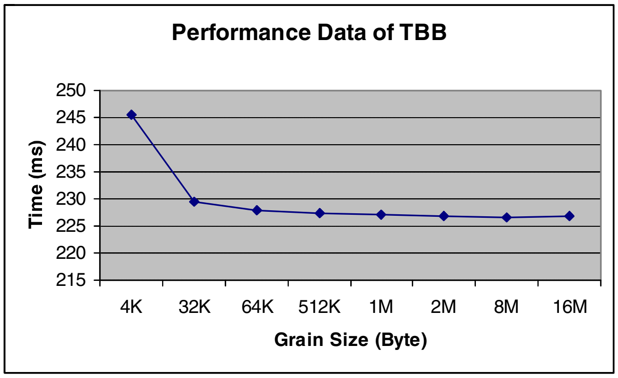


Figure 19: TBB performance of different grain sizes

RESULTS

Figure 20 shows the performance of Serial, Win32, and Intel TBB implementation of the ACBM search algorithm when the test file is 32M. Both Win32 and Intel TBB achieve better performance than the serial version.

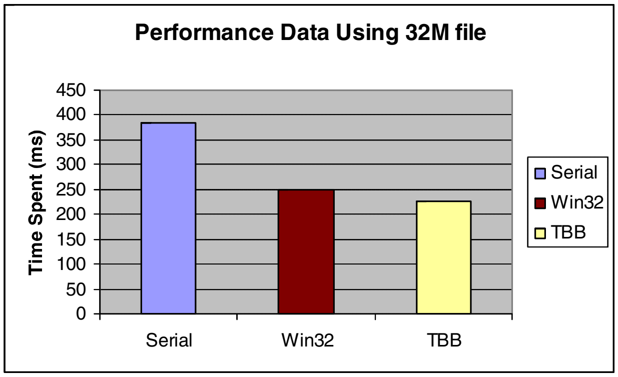


Figure 20: Performance data for different implementations

In order to evaluate the performance gain of parallelizing with Win32 and Intel TBB, we test the two applications with different file sizes and calculate the corresponding scalability. Figure 20 shows that both Win32 and Intel TBB attain better scalability as the file size grows. Intel TBB has better performance than Win32 for all the different file sizes. When the file size is 256M, Scalability$_{tbb}$ = 1.714 and Scalability$_{win32}$ = 1.580, Also, the average Scalability$_{tbb}$ = 1.655, and the average Scalability$_{win32}$ =1.549. Considering the ideal scalability is 1.867, Intel TBB demonstrates good scaling performance on a dual core system. Since the AC_BM algorithm can be used in the IDS detection engine, which is the core module of an IDS, parallelizing the algorithm can significantly improve the performance of the IDS.

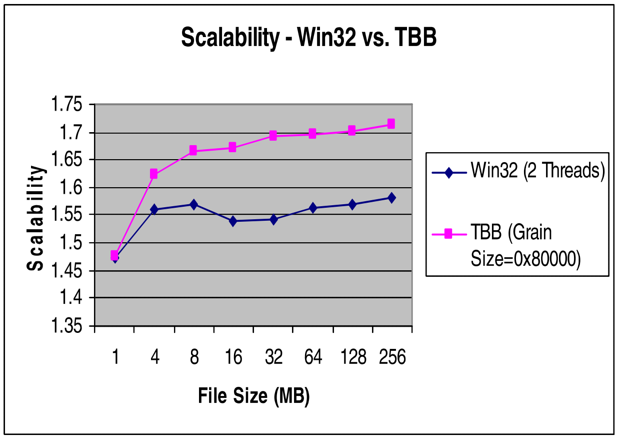


Figure 21: Scalability comparison between Win32 and Intel TBB

From the test results, we can see that Intel TBB demonstrates better performance than Win32 for the AC_BM algorithm. The reason is that with Intel TBB, we specify tasks instead of threads. A task can be assigned to a thread dynamically; Intel TBB selects the best thread for a task by using the task scheduler. If one thread runs faster, it is assigned to perform more tasks. However, with the Win32 Threading Library, a thread is assigned to a fixed task, and it can not be reassigned to other tasks even though it is idle.

From a developer's point of view, Intel TBB is easier to use. Intel TBB can dynamically decide the most suitable number of threads according to the platform and workload. Developers don't need to consider the granularity of the task by themselves. Further, when the application is migrated to another platform, Intel TBB automatically selects the best number of threads and assigns the tasks to the right threads to gain the best performance. Another advantage of using Intel TBB is that developers don't need to consider threading issues such as data race and inappropriate synchronization when they use the templates provided by Intel TBB, such as parallel_reduce, parallel_while, etc.

CONCLUSION

As multi-core processors become mainstream in the marketplace, software development needs to go parallel to fully exploit the performance potential of multi-core systems. For the rule-based IDS such as Snort, the parallel pattern matching algorithm in its detection engine can greatly improve the performance of intrusion detection, which in turn improves the performance of the overall IDS. As the IDS becomes more and more important in network security, and the efficiency of its detection engine contributes more to the overall performance of the IDS, it is worth it to parallelize the detection engine to attain the greatest performance gain in a multi-core environment.

Intel Threading Analysis Tools help developers write high-performance parallel software in all of the phases of the development cycle. By parallelizing the Aho-Corasick_Boyer-Moore algorithm using both the Win32 threading API and Intel TBB, we show how to parallelize and tune applications with Intel threading tools. We also show how to apply the parallel methodology to the real application during the analysis, design, and tuning phases. After performance tuning, both threading methods achieve much better performance than their serial counterparts. The Intel TBB attains a greater performance gain compared to the Win32 threading API, and it provides a more generic threading model to ease the implementation of parallelism.

ACKNOWLEDGMENTS

Thanks to Albert Hu for providing great advice on the paper content and organization. Thanks to Fane Li also for providing technical support on the parallel methodology.

REFERENCES

[1] Shwetha Doss and John O'Neill, Ph.D., "Best Practices for Developing and Optimizing Threaded Applications," *Go Parallel*, at http://www.devx.com/go-parallel/Article/33534.

[2] Michael Voss, I., "Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms," *DevX.com*, at http://www.devx.com/cplus/Article/32935.

[3] Ananth Grama and Anshul Gupta, *Introduction to Parallel Computing*, Addison Wesley, Chapter 3, Boston, 2003.

[4] "Getting Started with the VTune™ Performance Analyzer," Intel Corporation, at http://fm1cedar.cps.intel.com/softwarecollege/CourseDetails.asp?courseID=17.

[5] "Intel® Threading Profiler Getting Started Guide," Intel Corporation, at http://fm1cedar.cps.intel.com/softwarecollege/CourseDetails.asp?courseID=179.

[6] "Intel® Threading Checker Getting Started Guide," Intel Corporation, at https://fm1cedar.cps.intel.com/SoftwareCollege/CourseDetails.asp?courseID=178.

[7] "Intel® Threading Building Blocks Reference Manual," Intel Corporation, at http://www3.intel.com/cd/software/products/asmo-na/eng/294797.htm.

[8] C Jason Coit, Stuart Staniford, Joseph McAlerney, "Towards faster pattern matching for intrusion detection or exceeding the speed of snort," *DARPA Information Survivability Conference and Exposition*, 2001.

[9] Brian Caswell and Jeremy Hewlett, "Snort User Manual," at http://www.snort.org.

AUTHORS' BIOGRAPHIES

Lihui Wang joined Intel's IT Flexible Services Group in 2006. She works on system and application software development for Intel product groups. Her current work is on Service Delivery Operation. She received B.S. and M.S. degrees in Computer Science from Sichuan University. Her e-mail is lihui.wang at intel.com.

Xianchao Xu joined Intel's IT Flexible Services Group in 2005. He is part of the Research Engineering team and works on the development of wireless and TXT research prototypes. He also works on enabling Active Management Technology (AMT) technology and has taken part in the development of AMT configuring software. He received his B.S. degree in Computer Science from the University of Science and Technology of China and his M.S. degree in Computer Architecture from the Institute of Computing Technology, the Chinese Academy of Sciences. His e-mail is james.xu at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

Intel® Performance Libraries: Multi-Core-Ready Software for Numeric-Intensive Computation

Ilya Burylov, Performance Library Lab, Intel Corporation
Michael Chuvelev, Performance Library Lab, Intel Corporation
Bruce Greer, Performance Library Lab, Intel Corporation
Greg Henry, Performance Library Lab, Intel Corporation
Sergey Kuznetsov, Performance Library Lab, Intel Corporation
Boris Sabanin, Performance Library Lab, Intel Corporation

Index words: mathematics, library, parallel software, multi-core, vector math, BLAS, LAPACK

ABSTRACT

In this paper we present the Intel® Math Kernel Library (MKL) as a mathematical software package for scientific and technical computation designed for ease of use in environments that can vary greatly. Ease of use includes the build environment (use with different compilers), optimal performance on multiple platforms (automated selection of code based on the end-user system), optimal performance (optimization of an algorithm), interfaces to other libraries (FFTW), and effective use of multi-core processors through parallelization. We also discuss how this concept of ease of use will be expanded to provide more flexibility in the use of the library without greatly expanding its size.

Much of the paper is devoted to the optimization and parallelization of the library, critical in this era of multi-core processors. We discuss some of the methods used to improve performance that largely focus on cache utilization and minimization of table look-aside buffer (TLB) misses. Specifically, we look at the parallel performance of Basic Linear Algebra Subroutines [3] (BLAS), LAPACK [1], the Vector Math Library (VML), and a sparse linear solver (PARDISO). We include a brief section on a second application library, Integrated Performance Primitives (IPP), which complements the MKL in media applications.

INTRODUCTION

The Intel® Math Kernel Library (MKL) is a math library for use in scientific and engineering applications supporting a number of different mathematical areas:

Linear algebra. Basic Linear Algebra Subroutines (BLAS), LAPACK, ScaLAPACK, sparse BLAS, iterative sparse solvers, preconditioners, direct sparse solver (PARDISO)

Signal processing. FFTs, cluster FFTs

Vector math. Vector Math Library

Statistics. Vector Statistics Library with random number generators

PDEs. Poisson, Helmholtz solvers, trigonometric transforms

Optimization. Trust region solvers

Other. Interval linear solvers, multi-precision integer arithmetic

Among the key guidelines for the development of the library are using optimized math software for computationally demanding algorithms; threading and parallelizing these algorithms to make full use of multi-processor, multi-core [2], and multi-computer systems, making the library easy to use, and maintaining a high quality. Our focus in this paper is mostly on performance but we also introduce the paper with a discussion on ease of use.

A number of the features of the library do not relate to math functionality but contribute to ease of use. Some of these are:

- Designing the library to be compiler-independent eliminates the need for compiler-specific versions and allows C language programs to link to the Fortran portions of the library without the usual Fortran run-time libraries. Perhaps it is more correct to state that all compiler dependencies have been isolated (as will be explained in the discussion of the layer model of the library).

- Providing competitive performance on non-Intel® processors so software vendors can use a single library in their products for Intel® architecture computers.
- Parallelizing those parts of the library where parallelization makes sense. Most of the library functions could be parallelized but would not improve in performance if parallelized. Most of this paper deals with parallel performance on multi-core processors.
- Using interface files to map FFTW to MKL FFTs, other files to map older MKL FFTs to the more recent FFTs as well as using Java interface examples for various parts of the library.

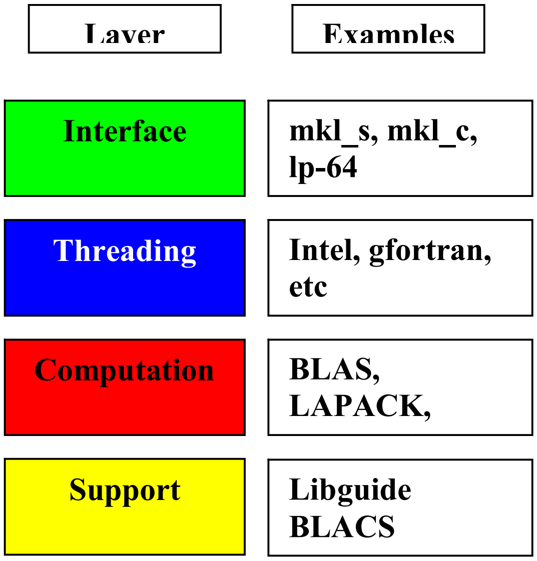
To further enhance usability, future versions of MKL will introduce a "layer model" (see Figure 1). This version will have four layers: interface, threading, computational, and run-time, or compiler-specific, library layer.

The first layer already exists for the 32-bit Windows* version but will be ubiquitous in the library. This layer allows MKL to accommodate different interfaces, including, for instance, gfortran. This and some other Fortran compilers handle complex return values differently than the Intel compiler for the Intel® 64 Architecture-based processors on Linux*. This difference can be dealt with through an interface file without duplicating the rest of the library. Similarly, the basic library for a 64-bit operating system (OS) will use 64-bit integers going forward, but LP64 (32-bit integers for a 64-bit OS) will be accommodated with a layer.

An area that has been problematic, and will be more difficult going forward, has been the intermingling of user threaded code with MKL, where the user's program is compiled with a non-Intel compiler. The second layer deals with this mismatch. All MKL threading is function based, so the threaded portion will be compiled with different compilers (Intel and gfortran, for instance) and the threaded portion provided as a layer. By turning threading off during compilation of the threaded software, a non-threaded layer will create a sequential version of the library. By linking in the appropriate threaded layer, multiple threading environments will be supported, including a sequential version of the library, with just a small increase in the size of the package.

The third layer is the computational layer. This layer does all the computations and includes processor-specific code that is chosen at run time.

The fourth layer contains support files such as libguide, the threading library for Intel® compilers, and the BLACS, which are specific to compilers and message passing interface (MPI) versions.



Layer	Examples
Interface	mkl_s, mkl_c, lp-64
Threading	Intel, gfortran, etc
Computation	BLAS, LAPACK,
Support	Libguide BLACS

Figure 1: Layer model for MKL

In the rest of this paper we focus on performance for multi-core processors. Fortunately, many of the methods needed to achieve scaling with multi-core processor systems are similar to those used in shared memory parallel systems, at least for many of the functions of MKL. However, because of the shared caches of multi-core processors there are additional opportunities for threading functions such as VML, as explained in one of the performance sections.

We discuss parallelization and optimization for several different areas supported by the Intel® libraries in this order: BLAS, LAPACK, sparse linear solvers, VML, and codecs from IPP. Other key functions such as FFTs are not discussed. Especially in the cases of the BLAS and LAPACK, the contribution of the MKL developers is to take extant code and optimize it, including parallelizing it where that makes sense.

The fundamental problem for much mathematical software is how to structure the problem in such a way that the caches can be effectively used. Before looking at these problems it is useful to look at the problem from a data consumption versus data supply rate point of view.

Consider the Intel® Core™2 Duo processor, with a dual core running at 3.0 GHz performing the dot product. If we assume that one vector can be kept in cache, at what rate must the memory system supply data to keep just one dual-core processor busy? Each processor can do two double-precision multiplies per clock or four multiplies per clock, requiring 32 bytes (8 bytes per double precision word) per clock. At 3 GHz, this is 96 GB/second. For a dual-socket system (Woodcrest) the system must provide 192 GB/s to keep all four cores busy. On a Clovertown system the number of cores doubles again and the demand, at the same frequency, goes to 384 GB/s.

Choose any realistic memory bandwidth and divide it into the rate at which the processor can consume the data and you will have an estimate of the number of times a datum must be used once it is in cache in order to keep all the cores busy.

Much of the optimization efforts of MKL are centered on how to get that reuse factor high as well as how to deal with the many architectural complexity issues. In the following sections we discuss some of the problems and solutions for performance in MKL and briefly in IPP.

BLAS

Libraries are often an easy way to improve performance of an application. In the Introduction, we discussed the broad range of functionality that MKL offers as well as some of the ways the design is intended to make its use easy. Applications linked with MKL will see improvements in performance, especially if run on multi-core systems, through the many threaded functions of the library.

The real issue is how MKL takes advantage of performance features such as SIMD hardware, and why multi-core processing exacerbates performance-sensitive issues. We start by describing single core performance optimization and move onto parallelization. If the single-core performance is far from optimal, it logically follows that the multi-core performance may not be ideal either.

Were MKL limited to a single set of functions, that set would be the BLAS because of its importance as a building block for higher order linear algebra functionality. The BLAS encapsulates several important dense linear algebra kernels.

“Levels” is an important notion of the BLAS philosophy. Examples of Level 1 algorithms include taking an inner product of two vectors, or scaling a vector by a constant multiplier. Level 2 algorithms are matrix-vector multiplication or a single right-hand-side triangular solve. Level 3 algorithms include dense matrix-matrix multiplication. If we assume a vector is length N or a matrix is order N, then the number of floating point operations (flops) for a Level 1, Level 2, and Level 3 algorithm are O(N), O(N^2), and O(N^3), respectively. The data movement, however, is O(N), O(N^2), and O(N^2), respectively. This last fact is crucial for optimization and threading performance. This makes the number of floating point operations per data item moved O(1), O(1), and O(N), respectively.

Memory performance is inadequate to directly support the computational speed of the processor. This gap has increased over the years and multi-core processors accelerate the mismatch between memory system performance and the data demands of the processor. To deal with this discrepancy, processors use a memory hierarchy. Each level of the memory hierarchy boasts a different latency and bandwidth. We consider the highest level machine registers. Register data movement keeps pace with processor clock rates. The next level is the first-level (L1) cache (small size) followed by the second-level (L2) cache (larger). Some machines also have a third-level (L3) cache (largest). Finally, at the bottom, there is the machine memory. This is often pictured as a pyramid, as shown in Figure 2.

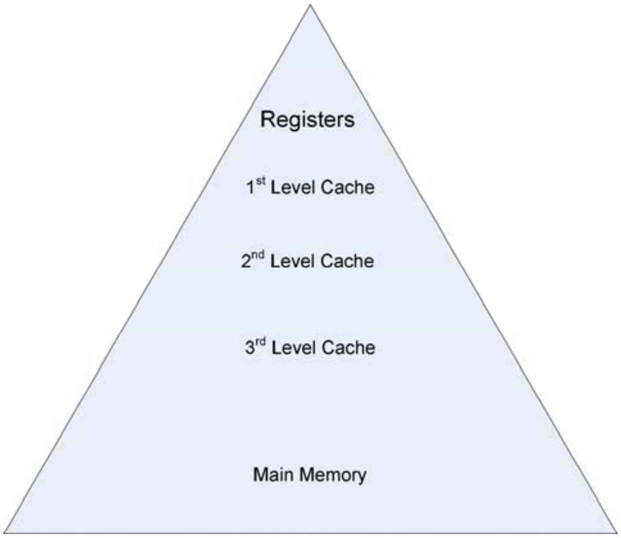


Figure 2: Memory hierarchy pyramid

The closer to the top of the pyramid, the more valuable the resource is and the greater its performance in terms of bandwidth and reduced latency. The challenge for the developer is to keep data in the fast memories long enough to amortize the cost of getting the data there. Blocking algorithms along with data organization ensure that more work gets done at the faster top of the pyramid. MKL blocks algorithms such as the Level 3 BLAS, where the amount of work, O(N^3), can be much greater than the amount of data movement, O(N^2).

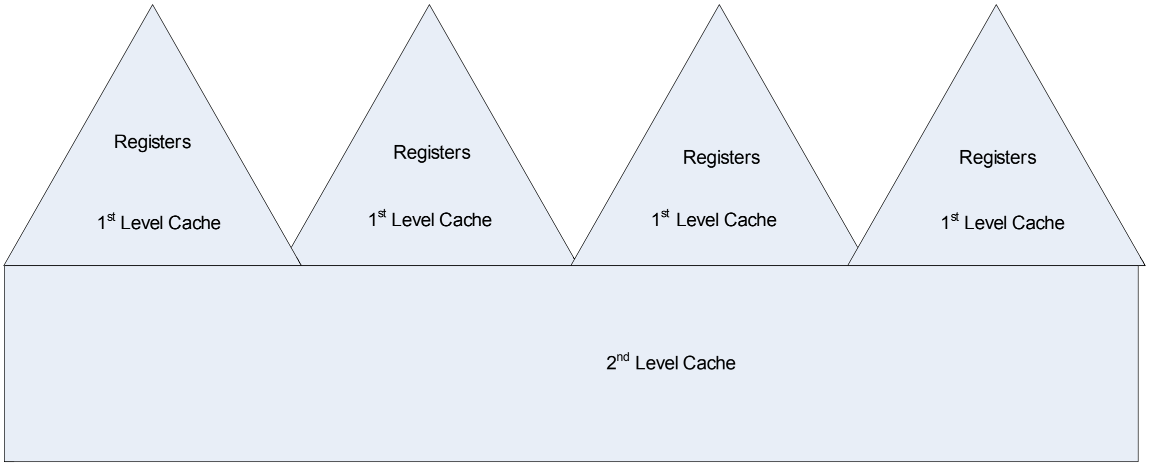


Figure 3: Shared cache top of pyramid

While this situation has existed in architectures for many years, the recent advent of multi-core processing merely adds to the complexity of the problem both because parallelism is not mandatory and because of the sharing of caches between cores. If two or more cores share a secondary cache, for instance, the familiar top of the pyramid suddenly looks like Figure 3. The bottom of the pyramid remains the same as in Figure 2.

What we typically see is a large cost for moving elements from main memory, compared to the very fast capacities of Figure 2. The complexity of the memory system—mapping the memory onto the cache—includes the use of an additional cache called the table look-aside buffer, or TLB. Each memory page mapped to the cache has a TLB entry.

When data are referenced, they may be in the L1 cache, L2 cache, or in memory. In addition, the page may or may not be in the TLB. Each miss—L1, L2, higher order cache, TLB—is increasingly expensive to retrieve. While the processor can hide some cache misses, TLB misses will cause stalls while the page address for the data is found and loaded.

In addition to the cache structure we have already outlined, most caches have a given associativity set, meaning how addresses are shared in their mapping to a given cache line. There is also a dependency on the cache replacement policies that determines when cache lines are evicted from the cache. There are other features of caches that will affect the performance of the processor: bank structure, how and when data are written back to memory, and so on.

All of these issues are accounted for either explicitly (by design) or implicitly (by automated searches through design space) for key MKL functions such as the BLAS. The result is code that is tuned for a single core. Now we need to parallelize the code.

One of the most important considerations is where to thread an application. If an algorithm from LAPACK calls the Level 3 BLAS, there is now a choice of where to thread. One can parallelize at the LAPACK level, the BLAS level, or both. We have consistently found it to be the case where parallelizing at the LAPACK level yields the greatest advantage.

Figure 4 illustrates this for the LAPACK function DGETRF, which performs LU factorization and is the basis for the LINPACK benchmark. The chart shows the ratios of performance for threading at the LAPACK and BLAS levels, with the BLAS-level performance being 1.0. Problem sizes are 1,000 times the abscissa values.

As the figure shows, for smaller sizes, the higher-level threading is up to 80% faster. But even at 30,000 equations, the LAPACK-level threading is nearly 10% faster on eight threads and 5% faster on two and four threads.

LAPACK

In the previous section we discussed the factors that go into the optimization of functions and showed how choosing the right level for parallelization can have a substantial impact on parallel performance as the number of cores increases, using LU factorization (DGETRF) as an example. The MKL has threaded and optimized many of the most important LAPACK functions. The problem is usually the same: how to feed the arithmetic units, which translates into how to get data into the caches and then to reuse them sufficiently to accommodate the substantial differences in the rate of consumption by the floating point hardware and the rate of supply by the memory subsystem.

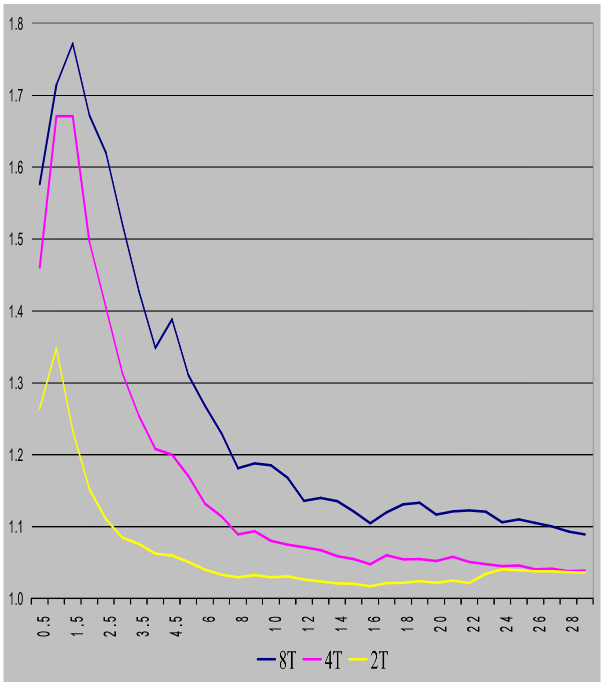


Figure 4: LAPACK vs. BLAS-level threading

LAPACK largely replaced LINPACK and employs blocked algorithms instead of the vector algorithms of LINPACK, making it much better suited for cache-based architectures. However, there are many areas where LAPACK code can further employ Level 3 BLAS [4] instead of the lower-level functions, which can improve cache usage. That, in turn, improves parallel performance, including performance on multi-core systems. We provide several examples of how increasing the use of higher-level BLAS substantially improves the performance of the MKL implementation of LAPACK over the reference implementation.

One of the important linear algebra applications is double-sided decompositions like singular value decomposition (SVD) or Symmetric Eigenvalue Decomposition. In MKL we block the chains of plane rotations using Level 3 BLAS, resulting in remarkable improvements in performance of up to about 18x. Figure 5 compares the resulting threaded symmetric solver DSYEV against the reference implementation, with performance improvements of up to approximately 18x. In this chart, the MKL performance[1] is threaded using eight threads, computing all eigenvectors.

A second example employing a blocking algorithm implementation that allows the use of higher-order BLAS are the routines operating on packed storage format. This optimization requires the allocation of additional workspace of size N*NB (where N is the size of the problem, and NB is the block size, usually around 64). Use of workspace is common in other LAPACK functions and the cost, in terms of memory usage, is small.

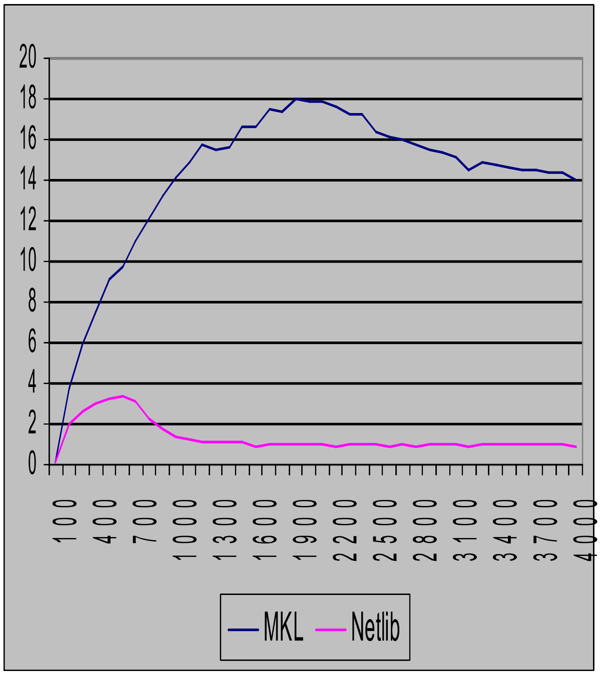


Figure 5: DSYEV improvements via Level 3 BLAS

In the case of the Cholesky solver performance on packed storage format, the performance improvement again is around 18x on the same system as for DSYEV, as shown in Figure 6.

While restructuring of the LAPACK code to use Level 3 BLAS improves performance markedly, more advanced techniques must be employed to minimize dependencies on the sequential code that remain after employing Level 3 BLAS.

[1] 2.4 GHz, Dual-socket, Quad-Core Intel® Xeon® processor 5300 Series 1067 MHz front-side bus. 2x4 MB L2 cache.

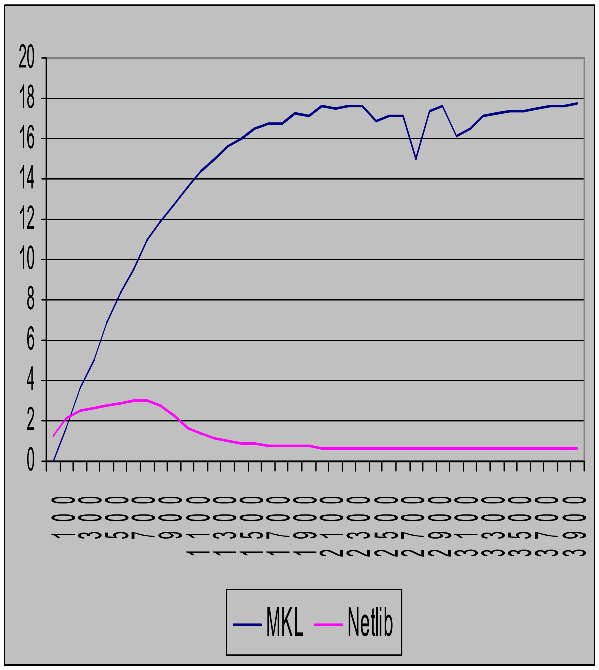


Figure 6: Packed-format Cholesky factorization

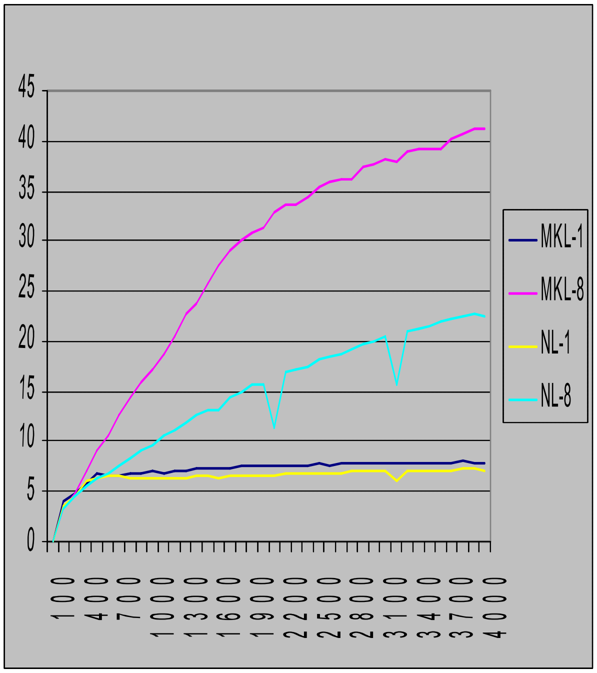


Figure 7: DGETRF-level versus BLAS-level threading

In such functions as LU and QR factorization [5], a look-ahead technique is used that allows the next block factorization to begin before the matrix has been fully updated, which increases concurrency. Figure 7 looks at DGETRF performance on an 8-core system comparing MKL versus netlib performance. As the chart shows, there are optimizations in MKL that improve the performance even on one thread vis-à-vis the reference implementation.

VECTOR MATH LIBRARY (VML)

As we suggested earlier, the main issue in threading various math functions is not so much whether they can be threaded (are operations separable) but rather whether there are sufficient operations on the data once they are in cache to permit other cores/processors to also get data to work on. In other words, this all comes down to a memory bandwidth issue.

The transcendental functions of VML typically require 10-50 cycles per element (CPE), typically with one input and one output value per element. Taking this into consideration we can roughly estimate the break-even point of threading by the following inequality:

$$S/T+O < S \rightarrow N*CPE/T+O < N*CPE \rightarrow N > O * T / (CPE * (T-1)),$$

where S = CPE*N and N is the vector length – is the number of cycles to execute a particular function in serial mode, O is the number of clocks for overhead for starting threads (it really depends on T, the total number of threads used) and CPE which is the cpe of the function in serial case. One can see that with increasing CPE (more complex functions) the shorter vectors can be effectively parallelized. The greatest difficulty here is to make an estimation of O.

Our computations show that O, measured in cycles, depends mostly on the number of sockets, the number of cores, and whether hyperthreading is turned on or off. This inequality, estimation of O, and a table of CPE values for each function are used in order to choose the number of threads for a particular function call during runtime.

Figure 8 shows the speedups for three VML functions on a Woodcrest system (dual socket, dual core) compared to single-thread performance on the same processor.

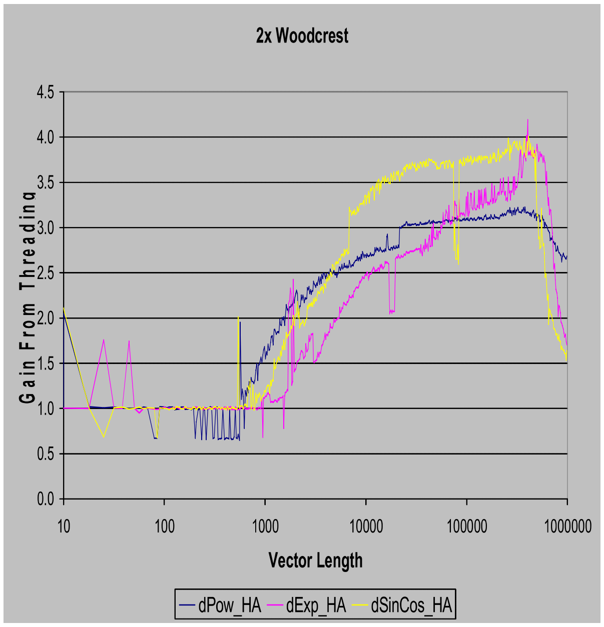


Figure 8: VML scaling on selected functions

Though VML can choose some particular number of threads it is difficult to do this accurately:

- *Performance is often data dependent*. For example, the dCbrt function (cubic root for double precision vectors) has 27.96 cpe for uniformly distributed data on the interval [-10000;10000], but 15.00 cpe if the vector is all zeros.
- *Data location*. If the input/output vectors are in cache, scaling and performance will be much better than if the data are in memory.
- When several different successive vector functions work with the same vectors, a different number of threads can be chosen, and as a result data might stay in the wrong cache if the cache is not shared.
- The influence of overhead might be significantly lowered by using threading at a higher level (i.e., if the user calls the VML functions from a threaded application).

In summary, for VML, multi-core shared cache architectures have opened opportunities for threading that did not exist previously, but the performance is dependent on factors the VML developer can only partly control. It is likely that in most cases calling VML functions from a threaded application will result in better performance than invoking the threaded VML.

SPARSE LINEAR ALGEBRA

Solving large sparse linear systems of equations is often a stumbling block in many scientific problems. MKL offers several approaches for solving such problems. The PARDISO solver is a sparse direct supernodal solver that is thread-safe, high-performing, and memory-efficient. Using this tool, you can solve symmetric and non-symmetric sparse linear systems of equations on shared-memory multi-processors. However, there is a point where the memory requirements for large systems can become prohibitively high, and the PARDISO/DSS (direct sparse solver) will not work. This is where MKL iterative sparse solvers come in: these solvers can provide a remedy, because only a few working vectors and the primary data need be stored.

MKL iterative solvers are based on a reverse communication interface (RCI) scheme that makes the user responsible for providing certain operations for the solver (for example, matrix-vector multiplications). To simplify the usage of MKL iterative solvers and gain additional performance, MKL offers sparse BLAS functions, which is a set of functions that perform a number of common vector and matrix operations for the most popular sparse storage schemes: compressed sparse row (CSR), compressed sparse column (CSC), diagonal, coordinate (COO), skyline, and block sparse row formats. Most MKL sparse BLAS routines are threaded using OpenMP. As in the case of the VML, for instance, performance on sparse BLAS is improved when the data are in the common cache for the cores and those BLAS are threaded.

Like dense matrices, the performance of MKL PARDISO/DSS and MKL sparse BLAS depends on the details of the machine architectures, but unlike dense problems, the performance of these components also depends on the structure of the matrix, because the distribution of the nonzero elements in a sparse matrix determines the memory access patterns. However, many physical problems expose a well-behaved sparse structure, or the rows can be re-ordered to yield a better structure. PARDISO uses approximate minimum degree ordering and METIS reordering techniques for getting permutations to minimize fill-in and the associated memory requirements. Internal storage for the matrix factors in PARDISO is a block format. Most of the computations are done with the help of MKL Level 3 BLAS and LAPACK. The usage of Level 3 BLAS and supernode pivoting coupled with supernode partitioning and synchronous computations allows PARDISO to achieve high-gigaflop rates and nearly linear speedup on multi-core platforms.

There are differences in optimization of Level 2 and Level 3 sparse BLAS on many core platforms, and some optimization problems are similar to the problems of dense Level 2 and Level 3 BLAS (e.g., low locality in Level 2 routines). For Level 3 sparse BLAS, reorganizing the computations to perform the entire set of multiplications as a single operation produces significantly better performance. It is natural to expect that performance and scalability of Level 3 sparse BLAS are better than those of Level 2 sparse BLAS. MKL sparse BLAS routines for the block sparse row format that exploit the benefits of data blocking have better data locality and vector instructions: for example, the SSE2 instruction set can be applied even for Level 2 sparse BLAS in this case. Similar optimizations are done for the diagonal and skyline format, because the elements of the source vector as well as destination vector are accessed sequentially. The Level 2 sparse BLAS operations for point entry sparse formats, such as the compressed sparse row (CSR) or coordinate formats (COO), are the most difficult area for optimization, because the elements of the source vector are accessed in a discontinuous way that leads to poor temporal locality. However, it appeared that even Level 2 sparse BLAS can be threaded effectively at least on the latest Intel® multi-core platforms. In addition, many well-known methods have been used for optimization of MKL Sparse BLAS. Among these are blocking, prefetching, OpenMP, etc., which allow for better performance of Sparse BLAS on multi-core architectures.

INTEGRATED PERFORMANCE PRIMITIVES (IPP)

IPP is a multi-functional library highly optimized for Intel architecture. IPP covers 15 functional domains that can be recognized by a suffix in the library file names. For example, functions with IPPs in their names are signal processing functions, note suffix “s.” There are more than two-thousand functions processing 1D signals/data of different data types: real and complex, signed and unsigned, floating point, and integer. The other libraries in IPP are image processing “i,” JPEG primitives “j,” audio coding “ac,” color conversion “cc,” string processing “ch,” cryptography “cp,” computer vision “cv,” data compression “dc,” small matrix operations “m,” realistic rendering “r,” speech coding “sc,” speech recognition primitives “sr,” video coding “vc,” vector math “vm.”

IPP is optimized for several Intel architectures: IA32, IA64, Intel® 64, and IXP. Within each architecture are optimizations for specific processors. For instance, within IA32 architecture there are specific optimizations for the Pentium® 4 and Intel Core 2 Duo processors, among others.

IPP is optimized at three levels: algorithmic, effective use of SIMD instructions (SSE2, SSE3), and parallelization at both the primitive and component levels. Primitive-level threading is the threading implemented in IPP functions. Not every function in IPP is parallelized because of the overhead added by threading. However, the good news here is that IPP is by design a set of build blocks and applications that developers can easily use to thread their application by calling the primitives on different threads.

Component-level threading is threading provided in such components as video codecs, the H264 encoder and decoder; the jpeg viewer, and the IPP implementation of well-known data compression libraries, ZLIB and GZIP. These components, as well as others, are shipped with IPP as IPP samples given in their source codes.

An example of algorithm optimization is the median filter in the Signal and Image processing domains. Table 1, for instance, illustrates the results, in clocks-per-element, of IPP optimization of the median filter compared with the LEADtools library.

Table 1: IPP compared to LEADtools on median filter

Spatial filter with mask 5x5	Function	cpe
LEADtools	L_MedianFilterBitmap	345
IPP	ippiFilterMedian_16s_C3R	35

CPU optimization with the SIMD instruction set, which is done for many functions in IPP, also gives a performance gain that can be measured by comparing the performance ratio numbers of the C version of the library to the CPU specific library, such as optimizing for the Intel Core 2 Duo processor. Table 2 illustrates the performance advantage of multi-core threading on MPEG4 decoding.

Table 2: Speedup on threaded MPEG4

Stream[2]	Resolution	Frames	Bitrate MB/s	FPS 1T	FPS 2T	Ratio
1	1280x720	IPB	4.0	199	328	1.65
2	720x576	IP	4.7	298	411	1.38
3	640x476	IP	2.2	671	841	1.25
4	640x480	IP, OBMC	1.0	650	650	1.6

[2] Stream 1: preakness_59.94fps_Xvid_4Mbs_CBR.avi; Stream 2: Boss.avi; Stream 3: Taxi.avi; Stream 4: Term2.divx

SUMMARY

The Intel MKL is part of a suite of tools offered by Intel to help developers create software efficiently and to achieve high performance. For MKL, the goal has been to provide an easy-to-use software package to aid in the development of mathematical software. Achieving that goal has a number of facets, some of which we have touched on in this paper: functionality, compiler independence, performance, and the most recent efforts in performance, focusing on helping the user get the full benefits available from Intel multi-core systems. We have discussed in general terms some of the approaches taken by library developers to achieve the performance goals including threading at a higher level of functionality (LAPACK) and improving the locality of reference for data in LAPACK codes through more effective use of the Level 3 BLAS, and so on.

As the complexity and core counts for microprocessors continue to grow, MKL (and IPP) will optimize functions that impact performance in key application areas ensuring full and effective use of those processor developments.

ACKNOWLEDGEMENTS

Neither MKL nor IPP would be possible without the creative and disciplined efforts of the teams that develop these software packages. We acknowledge their contributions that make these features, functions, and performance possible. Both libraries have been largely created by groups of developers with strong backgrounds in computer science and mathematics in Russia.

REFERENCES

[1] Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenny, A., and Sorenson, D., *LAPACK User's Guide. 3rd ed*., SIAM, Philadelphia, PA., 1999.

[2] Buttari1, A., Dongarra, J., Kurzak, J., Langou, J., Luszczek, P. and Tomov, S., "The Impact of Multicore on Math Software," at http://icl.cs.utk.edu/projectsfiles/sans/pubs/para-multicore-2006.pdf

[3] Dongarra, J., DuCroz, J., Duff, I. Hammarling, S., "A set of Level 3 Basic Linear Algebra Subprograms," *Technical Report, ANL-MCS-TM-88*, Argonne National Laboratory, Argonne, ILL, 1988.

[4] Lang, B, "Using Level 3 BLAS in Rotation-Based Algorithms," *SIAM Journal on Scientific Computing*, Volume 19, Number 2, pp. 626–634, 1998.

[5] Starzdins, P., "A comparison of lookahead and algorithmic blocking techniques for matrix factorization," *TR-CS-98-07, The Australian National University*, July 1998.

AUTHORS' BIOGRAPHIES

Ilya Burylov is a Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. His interests include multi-core threading technologies, numerical analysis, and partial differential equations in hydrodynamics. He received his M.S. degree from the Perm State Technical University. His e-mail is ilya.burylov at intel.com.

Michael Chuvelev is a Senior Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. He graduated from Moscow Institute of Physics and Technology with an M.S. degree in Applied Mathematics. At Intel he has specialized on code parallelization and linear algebra software optimization, particularly of LAPACK, ScaLAPACK, and sparse solvers. He is currently focusing on LAPACK optimization on SMP systems, especially on multi-core systems. His e-mail is michael.chuvelev at intel.com.

Bruce Greer is a Principal Engineer in the MKL team in the Developer Products Division of the Software Solutions Group. He was manager of MKL from 1995 until May of this year. He received an M.S. degree in Physics from Georgia Tech. His professional interests are in code optimization. Despite his handicap, he has a passion for golf. His e-mail address is bruce.s.greer at intel.com.

Greg Henry is a Principal Engineer in the MKL team in the Developer Products Division of the Software Solutions Group. His research interests are linear algebra, parallel computing, numerical analysis, scientific computing, and all things relevant to MKL. He received his Ph.D. degree from Cornell University in Applied Mathematics and started working at Intel in August 1993. Greg has three children and a wonderful wife, and writes novels as a hobby. His e-mail is greg.henry at intel.com.

Sergey Kuznetsov is a Senior Software Engineer in the Performance Library Lab of the Intel Software and Solutions Group. His research interests include parallel numerical linear algebra, sparse matrix computations, parallel algorithms and eigenvalue problems. He received his Ph.D. degree from the Novosibirsk State University, Russia. His e-mail is sergey.v.kuznetsov at intel.com.

Boris Sabanin is a Principal Engineer and the Engineering manager of the Intel Integrated Performance Primitives library in the Intel Systems and Solutions Group. He is focusing on the design, development and optimization of signal processing functions. His e-mail is boris.sabanin at intel.com.

GLOSSARY

BLACS: Basic Linear Algebra Communication Subprograms – a set of functions developed for ScaLAPACK which isolate the communications used by the software from the communication layer such as MPI. Used throughout MKL cluster software.

BLAS: *Basic Linear Algebra Subprograms* – a set of dense vector, vector matrix and matrix math functions useful in creating higher level functions such as solvers.

CODEC: COder/DECoder – used for encoding or decoding digital data streams such as video or audio.

DSS: Direct Sparse Solver – solves a system of equations in an *a priori* known number of operations in contrast to iterative sparse solvers for which the number of operations is data dependent.

FFT: Fast Fourier Transform – algorithms to convert, for instance, a time series into a frequency series in an efficient way.

FFTW: Fastest FFT in the West – a publicly available software package to create highly optimal FFTS. See http://www.fftw.org.

IMSL – A large commercial math software package. See http://www.vni.com

LAPACK: Linear Algebra PACKage – a set of solvers for systems of equations, eigensolvers, etc, using blocked algorithms that make effective use of the Level 3 BLAS.

LINPACK – Predates LAPACK and based on vector operations. Also a benchmark solving systems of linear equations.

METIS – A set of programs for partitioning unstructured graphs. See http://glaros.dtc.umn.edu/gkhome/views/metis

MPI: Message Passing Interface – A widely used distributed memory (cluster) communication package.

NAGLIB – A large math software package similar to IMSL. See http://www.nag.com

NETLIB – A repository of software packages such as BLAS, BLACS, LAPACK, LINPACK, ScaLAPACK and others. See http://www.netlib.org

PARDISO: PARallel DIrect SOlver – A parallel direct solver from University of Basel and licensed by MKL. See http://www.pardiso-project.org

PDE – Partial Differential Equation.

ScaLAPACK: Scalable LAPACK – cluster versions of much of LAPACK.

SIMD: Single Instruction Multiple Data – hardware to perform multiple arithmetic operations simultaneously on a single instruction, such as the SSE2 and SSE3 instructions.

The Foundations for Scalable Multi-core Software in Intel® Threading Building Blocks

Alexey Kukanov, Performance, Analysis and Threading Lab, Intel Corporation
Michael J. Voss, Performance, Analysis and Threading Lab, Intel Corporation

Index words: threading building blocks, threading, scalability, parallelism, software

ABSTRACT

This paper describes two features of Intel® Threading Building Blocks (Intel® TBB) [1] that provide the foundation for its robust performance: a work-stealing task scheduler and a scalable memory allocator.

Work-stealing task schedulers efficiently balance load while maintaining the natural data locality found in many applications. The Intel TBB task scheduler is available to users directly through an API and is also used in the implementation of the algorithms included in the library.

In this paper, we provide an overview of the TBB task scheduler and discuss three manual optimizations that users can make to improve its performance: continuation passing, scheduler bypass, and task recycling. In the Experimental Results section of this paper, we provide performance results for several benchmarks that demonstrate the potential scalability of applications threaded with TBB, as well as the positive impact of these manual optimizations on the performance of fine-grain tasks.

The task scheduler is complemented by the Intel TBB scalable memory allocator. Memory allocation can often be a limiting bottleneck in parallel applications. Using the TBB scalable memory allocator eliminates this bottleneck and also improves cache behavior. We discuss details of the design and implementation of the TBB scalable allocator and evaluate its performance relative to several commercial and non-commercial allocators, showing that the TBB allocator is competitive with these other allocators.

INTRODUCTION

Performance-oriented developers now face the daunting task of threading their applications. Introducing parallelism into an application is a large investment. It is therefore imperative to implement a scalable solution, one that continues to increase performance, as the number of available cores and threads increases.

Intel TBB is a C++ template library that is designed to assist developers in porting their applications to multi-core platforms. The TBB library provides generic parallel algorithms [18] and concurrent containers [19] that enable users to write parallel programs without directly creating and managing threads. These algorithms are tested and tuned for the current generation of multi-core processors, and they are designed to scale as the core count continues to increase.

To provide efficient performance today and continued scalability tomorrow, the library is designed to support fine-grain parallelism through tasks. Tasks are user-level objects that are scheduled for execution by the TBB task scheduler. The task scheduler maintains a pool of native threads and a set of per-thread ready pools of tasks. At initialization, the TBB scheduler creates an appropriate number of threads in the pool (by default, 1 per hardware thread) and maintains the ready pools using a randomized work-stealing algorithm [2, 3].

In this paper, we describe the design of the TBB task scheduler and several scheduling optimizations users can keep in mind while coding their applications. In the Results section, we explore the scalability of TBB applications and highlight the impact of these scheduling optimizations on performance.

The task scheduler is complemented by the Intel TBB scalable memory allocator. In this paper, we provide an overview of its design and look at the tradeoffs. We compare its performance to several other commercial and non-commercial allocators.

RELATED WORK

The Intel TBB task scheduler is inspired by the early Cilk scheduler [2, 3]. Cilk is a parallel extension of the C programming language that defines additional keywords and constructs. The Cilk project was a descendant of the Parallel Continuation Machine (PCM)/Threaded-C [13].

Both Cilk and the Intel TBB schedule lightweight tasks onto user threads. The Chare Kernel [14] is a portable set of functions that allows users to express parallelism in terms of small tasks (chares) with the runtime transparently managing resources. Unlike Intel TBB and Cilk, however, the Chare Kernel is targeted toward message passing machines.

Mainstream languages, such as those supported by the .NET CLR also recognize the need for thread pools, where users can submit tasks without the need to explicitly manage threads [15]. However, in the .NET CLR these thread pools are targeted at general-purpose applications and are not tuned for compute-intensive applications.

The McRT research program at Intel presented a software prototype of an integrated runtime library for large-scale chip-level multiprocessing (CMP) platforms [17], including a highly configurable, user-level scheduler. It can be used to realize a variety of co-operative scheduling strategies, including work stealing.

The design of the Intel TBB scalable allocator is based on contemporary research in scalable memory allocation [8, 9] and utilizes best-known design solutions; it has common roots with Hoard [8], LFMalloc, Vam [10], Streamflow [11] and other state-of-the-art concurrent and sequential allocators. The TBB scalable allocator is a productization of the scalable memory allocator developed as part of the McRT research program [7, 17].

THE TBB TASK SCHEDULER

The Intel TBB task scheduler is a *work-stealing scheduler.* The design of the TBB scheduler is inspired by the early Cilk scheduler, which Blumofe and Leiserson [2, 3] proved has optimal space, time, and communication bounds for well-structured ("fully strict") programs.

In a system that uses work-stealing, each thread maintains a local pool of tasks that are ready to run. Using local pools avoids the contention that may arise with the use of a global task queue. When executed, a task performs work and also may create additional tasks that are placed in the local pool. If a thread's pool becomes empty, it attempts to steal a task from another random thread's pool. This approach is in contrast to static scheduling methods where threads are assigned work up-front and from other dynamic scheduling methods where a central pool of tasks (or iterations) is maintained.

Blumofe and Leiserson [2, 3] showed that the expected parallel runtime of applications scheduled by the Cilk scheduler is $E[T_P] = O(T_1/P + T_\infty)$, where T_1 is the "work" or sequential time of the application, and T_∞ is the critical path length. This optimal bound shows that as P→ ∞, the expected time is only limited by the critical path length (the sequential part) of the application.

To achieve these same optimal bounds, the TBB task scheduler also uses a randomized work-stealing algorithm. An overview of its implementation is provided in the following section.

An Overview of the Task Scheduler Design

The TBB task scheduler evaluates *task graphs*. A task graph is a directed graph where nodes are tasks, and each node points to its *parent,* which is another task that is waiting on it to complete, or NULL. Each task has a *refcount* that counts the number of tasks that have it as their parent. Each task also has a *depth*, which is usually one more than the depth of its parent. The work of the task is performed by a user-defined function `execute` that is encapsulated within the task object.

To assist in providing an overview of the Intel TBB task scheduler, we use calculation of the n^th Fibonacci number as a running example. A serial implementation of our Fibonacci example is shown below:

```
long SerialFib( long n ) {
 if( n<2 )
  return n;
 else
  return SerialFib(n-1)+SerialFib(n-2);
}
```

The function `ParallelFib`, shown below, uses the TBB task API to construct the root node of a task graph, an object of type `FibTask`. When this task's function `execute` is called, it will create two child tasks, also of type `FibTask`. Child `a` will calculate fibonacci(n-1) and child `b` will calculate fibonacci(n-2). When each of these tasks is executed, they will in turn recursively spawn child tasks as follows:

```
class FibTask: public task {
public:
 const long n;
 long* const sum;
 FibTask( long n_, long* sum_ ) :
     n(n_), sum(sum_) {}

 task* execute() {
  if( n<CutOff ) {
   *sum = SerialFib(n);
  } else {
   long x, y;
   FibTask& a = *new( allocate_child() )
       FibTask(n-1,&x);
   FibTask& b = *new( allocate_child() )
       FibTask(n-2,&y);
   set_ref_count(3);
   spawn( b );
   spawn_and_wait_for_all( a );
   *sum = x+y;
  }
  return NULL;
 }
};

long ParallelFib( long n ) {
 long sum;
 FibTask& a = *new(task::allocate_root())
     FibTask(n,&sum);
 task::spawn_root_and_wait(a);
 return sum;
}
```

For performance reasons, TBB requires users to set task refcounts explicitly with the `set_ref_count` call, instead of atomically incrementing it in `allocate_child`. The refcount should be set for a task before spawning any of its children.

Each task that spawns children waits at the `spawn_and_wait_for_all` call until all of its children complete. An additional guard reference is required for this, as shown in the above example by using the refcount of 3, while there are only 2 child tasks. A thread that enters a `spawn_and_wait_for_all` is free to execute other ready tasks while it waits.

In `ParallelFib,` after completing the wait call, the results of the child tasks are summed and returned. When `n<CutOff,` no additional child tasks are created and instead the leaf task will directly call SerialFib.

Figure 1 shows a snapshot of a task graph that might be created by an execution of ParallelFib. Tasks with non-zero reference counts (A, B, and C) must wait for their child tasks to complete before proceeding. The leaf tasks are ready to run.

As mentioned previously, the TBB library maintains a pool of threads, each of which has its own pool of ready tasks. Each per-thread task pool is implemented as an array of lists of tasks. A task goes into a pool only when it is deemed ready to run, i.e., it has been spawned and has a refcount of 0. Figure 2 shows a snapshot of a pool that corresponds to the task graph in Figure 1. Tasks A, B, and C do not appear in the pool because they have non-zero refcounts and therefore are not ready to run.

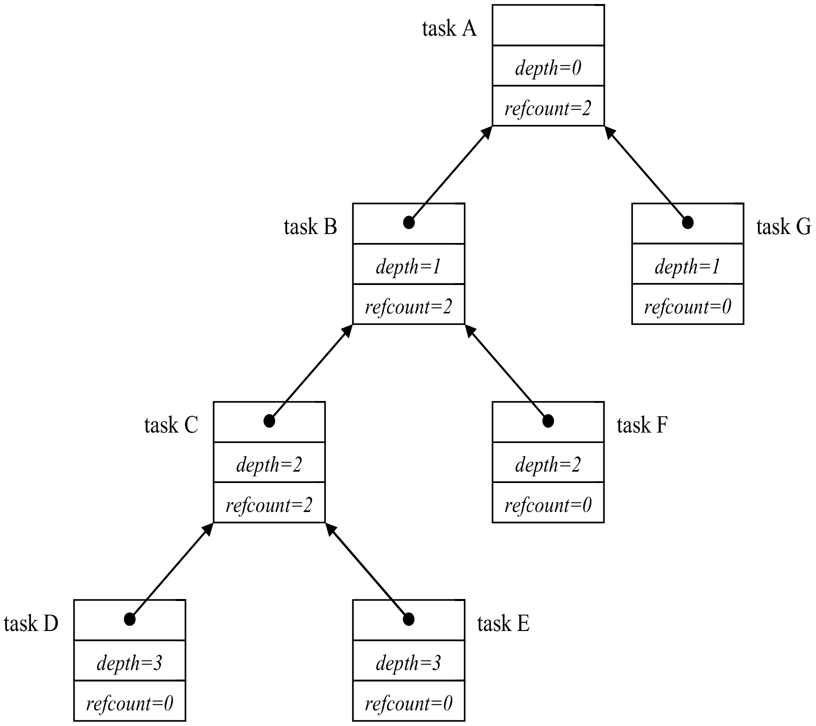


Figure 1: Intermediate task graph for the Fibonacci example

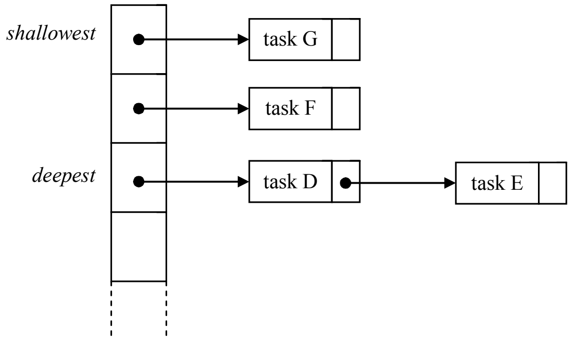


Figure 2: A pool of ready tasks that corresponds to the graph in Figure 1

Breadth-First Theft and Depth-First Work

The TBB task scheduler's fundamental strategy is "breadth-first theft and depth-first work." The breadth-first theft rule raises parallelism sufficiently to keep threads busy. The depth-first work rule keeps each thread operating efficiently once it has sufficient work to do.

A depth-first execution of a graph is the most efficient when performing a sequential execution because it provides better temporal locality and limits the space required for storing tasks. The deepest tasks are the most recently created tasks, and therefore are hottest in cache. When they complete, their parents can then execute, and although the parents are not hot in the cache, they're warmer than the tasks above them. A depth-first execution also limits the space required for storing tasks. When executing a node, only the nodes that lie along the path from the root to that node need to exist in memory.

Depth-first execution of a graph, however, limits parallelism. In contrast, always executing the shallowest tasks first leads to a breadth-first unfolding of the tree. This creates an exponential number of nodes that coexist simultaneously, providing ample tasks to steal but also excessively consuming memory.

To balance efficient execution and parallelism, the TBB scheduler therefore uses the "breadth-first theft and depth-first work" rule.

Each thread in the TBB thread pool executes a worker routine that actively looks for ready tasks to execute. A thread will first take the task at the front of the *deepest* list of its *own* pool[1]. If there are no ready tasks in its own pool and there is at least one non-empty task pool, it will then steal from the front of the *shallowest* list of *another* randomly chosen pool. If the chosen pool is empty, the thread tries to steal from another randomly selected thread until it succeeds.

Scheduling Trade-offs and Optimizations

The Intel TBB task scheduler was inspired by the Cilk scheduler. Cilk is a parallel extension of the C programming language that defines additional keywords and constructs. Since Cilk requires a modified C compiler, it can rely on the compiler to perform Cilk-specific transformations and optimizations.

TBB on the other hand is a C++ template library and can be compiled using any standard-compliant C++ compiler. While this makes TBB more portable, it also means that correctness and performance cannot depend on any TBB-specific compiler passes. The TBB task API has therefore been designed to allow users to perform certain scheduling optimizations "manually" to achieve increased performance when necessary. The most important of these optimization opportunities are discussed below and their impact is evaluated in the Experimental Results section.

Minimizing Stack Use with Continuation Tasks

As mentioned before, TBB uses a "breadth-first theft and depth-first work" approach. However, this approach can sometimes cause the processor stack to overflow.

For example, consider the case when a task enters a `spawn_and_wait_for_all.` The task cannot continue until all of its children complete. On entering the wait, the calling thread is released to execute or steal other tasks. If it steals the shallowest task from another thread, it then begins a depth-first execution of this stolen tree.

However, the initial task that entered the `spawn_and_wait_for_all` is kept on the processor stack to maintain its local storage and instruction pointer. The newly stolen tree then begins to unfold on top of the waiting subtree on the processor stack. This situation could occur repeatedly, causing the stack to overflow.

To avoid this situation, the TBB task scheduler forces a thread to only steal tasks that are deeper than any waiting task. While this limits stack growth, it also limits the choice of tasks to steal and therefore might limit parallelism.

To avoid restricting the choice of tasks to steal while at the same time limiting stack space growth, the TBB task interface allows developers to specify *continuation tasks*. A task can replace itself in the graph with a continuation task and then return, freeing up its stack space. When the children complete, the continuation task is spawned to finish the work delegated to it by the parent.

To use a continuation task `c`, the children of a task are allocated as children of `c` and not the task itself. Like

[1] Optimizations will be discussed later that allow tasks to directly return a next task to execute, bypassing the task scheduler.

other tasks, c becomes ready to run when its children complete and will only then be spawned. The code for spawning children using "continuation-passing" for our Fibonacci example is shown below:

```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& a = *new( c.allocate_child() )
    FibTask(n-2,&c.x);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
c.set_ref_count(2);
c.spawn( b );
c.spawn( a );
return NULL;
```

The implementation of class FibContinuation (not shown) inherits from class task, and sums c.x and c.y into sum in its execute function. The benefit of this approach is that after spawning children tasks in FibTask, the execute function returns, removing itself from the stack. Only the tasks that are actively executing are on the processor stack.

While there are benefits to the use of continuation tasks, there are also downsides. When using continuation tasks, all live state passed from parent to child cannot reside in the parent or its execute stack frame, since the parent may be destroyed before the child completes. Therefore additional care may be needed to properly encapsulate the live state of the computation. Also continuation passing requires the creation of an additional task object. In fine-grain tasks, this additional runtime overhead of task creation might be noticeable.

Reducing Overheads: Scheduler Bypass and Task Recycling

Luckily once an algorithm is using continuation tasks, it can also make use of two other overhead reducing techniques: *scheduler bypass* and *task recycling*.

With scheduler bypass, a task's execute function explicitly returns the next task to execute. Since the next task is known, the more complex logic to select a task is avoided in the scheduler's code. To use scheduler bypass in our Fibonacci example, the child task a is not spawned but is instead returned as shown below:

```
~~c.spawn( a );~~
return &a;
```

Once continuation passing and scheduler bypass are in use, it also becomes possible to recycle task objects. Normally when a task returns from its execute function, the task object is automatically deallocated. However, a user can choose to recycle a task object, making it live beyond the return and avoiding the repeated allocation and deallocation of task objects. Recycling a task as one of its own children is shown below for Fibonacci:

```
FibContinuation& c =
   *new( allocate_continuation() )
    FibContinuation(sum);
FibTask& b = *new( c.allocate_child() )
    FibTask(n-1,&c.y);
recycle_as_child_of(c);
n -= 2;
sum = &c.x;
c.set_ref_count(2);
c.spawn( b );
return this;
```

As shown in the Experimental Results section, scheduler bypass and task recycling often more than make up for the extra overhead added from the allocation of a continuation task.

SCALABLE MEMORY ALLOCATION

Until recently, mainstream client applications have targeted single-processor PCs. Therefore state-of-the-art general-purpose memory allocators such as Doug Lea's dlmalloc [4] have evolved to optimize for the sequential case. They were designed with two main principles in mind: efficient use of memory space and minimization of CPU overhead. Unfortunately, design decisions made to achieve these principles often hinder these allocators from providing good parallel performance.

Even the best sequential allocator can easily become a performance bottleneck in a parallel application. To ensure correctness, access to its heap must be properly protected. Using a single global lock for protection would amount to serializing all allocations. Detlefs et al. [5] showed that real applications spend up to 20% of execution time in memory allocator routines (even more with inefficient allocators). According to Amdahl's law [6], an application that is 20% sequential can never achieve more than a 5x speedup, even when using an infinite number of cores. Serializing allocations is therefore clearly not a scalable solution. Though more advanced schemas were developed to adapt dlmalloc for multi-threaded applications [12, 16], their scalability is also limited [8, 9, 12].

In addition, while space and CPU efficiency remain considerations in the design of a scalable memory allocator, they are not as important as before. The larger memory sizes available in the average PC and the growing speed gap between CPU and memory bring other considerations to the forefront, such as cache locality and prevention of false sharing.

Unfortunately, malloc implementations supplied by widely used C runtime libraries such as glibc and the Microsoft Visual C++* RTL still do not provide proper scalability for multi-threaded applications. As Intel TBB aims to ease the development of efficient and scalable parallel applications, it is unable to rely on these by-default allocators, and therefore provides its own scalable memory allocation library.

The TBB Scalable Allocator

The TBB scalable allocator is a productization of the scalable memory allocator developed as part of the McRT research program at Intel [7, 17].

In TBB, we improved the McRT code for better portability (for example, we had to rework the parts depending on other components of the McRT library) and addressed the performance of some corner case situations that were ignored by the research project. However, the major structure of the TBB scalable allocator is the same as the McRT design.

Figure 3 shows the high-level design of the TBB scalable allocator.

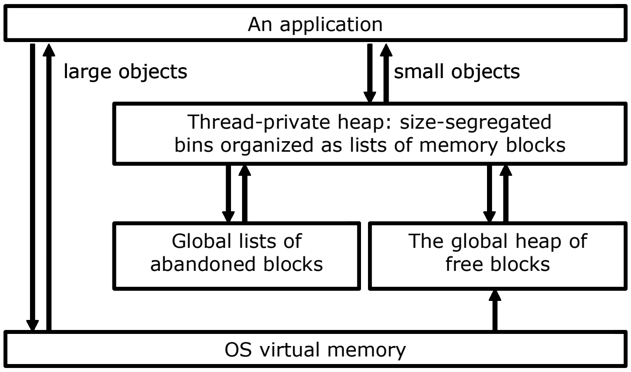

An application
large objects
small objects
Thread-private heap: size-segregated bins organized as lists of memory blocks
Global lists of abandoned blocks
The global heap of free blocks
OS virtual memory

Figure 3: High-level design of the scalable allocator

The allocator requests memory from the OS in 1MB chunks and divides each chunk into 16K-byte aligned *blocks*[2]. These blocks are initially placed in the global heap of free blocks. Currently, requested memory is never returned to OS (except for large allocations as described below), so the allocator carefully ensures that memory is reused. New blocks are only requested when a thread can't find any free *objects* in the blocks of its own heap and there are no available blocks in the global heap.

As in some other allocators, requests for large objects are redirected straight to OS virtual memory services. In the TBB allocator, the border between large and "regular" sizes lies slightly below 8K. However, we found that for better competitiveness, memory pieces of 8K to ~64K size should also be cached; explicitly managing these sizes is part of the future work for TBB.

Like many other widely used concurrent allocators, the TBB allocator uses *thread-private heaps*. Such a design has proven to cut down on the amount of code that requires synchronization, and reduce false sharing, thus providing better scalability. Each thread allocates its own copy of heap structures and accesses it via thread-specific data (TSD) using corresponding system APIs.

The heaps are *segregated*, i.e., they use different storage bins to allocate objects of different sizes. A memory request size is rounded up to the nearest object size. This technique provides better locality for similarly-sized objects that are often used together (for example, imagine an application traversing over a list or a tree). In the TBB allocator, the difference between consecutive object sizes in general does not exceed 25%, so internal fragmentation remains reasonable.

Figure 4 illustrates the internal design of a bin. A bin only holds objects of a particular size, and it is organized as a double-linked list of blocks. At each moment, there is at most one *active block* used to fulfill allocation requests for a given object size. Once the active block has no more free objects, the bin is switched to use another block.

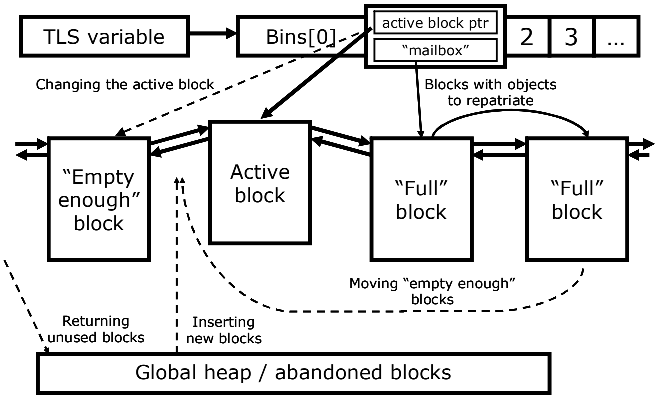


Figure 4: Design of a storage bin in a thread-private heap

Unlike in other allocators, the active block may be located in the middle of the list; *empty enough*[3] blocks are placed before it, and *full* blocks are placed after it. This design

[2] Following the authors of the McRT malloc [7], we will use terms "object" and "block"; in other literature, they can be called "block" and "superblock," respectively.

[3] A block counts as "empty enough" if the share of allocated objects drops below the predefined threshold.

minimizes the time to search for a new block if the active one is full. When enough objects are freed in a block, the block is moved right before the active block and thus becomes available for further allocation. A block with all its objects freed returns back to the global heap; new blocks are taken from there as required.

The design decisions made at higher levels allow certain optimization techniques for object allocation. With thread-local heaps, the common allocation path does not contain synchronization apart from the TSD access managed by the OS; the same is true for deallocation of a thread's own objects, as shown below.

Size segregation and aligned blocks have made per-object headers needless; all information required to free an object can be easily obtained via the *block header*. As a result, objects are tightly packed in the block (as shown in Figure 5), which leads to a potentially smaller memory footprint and better cache locality.

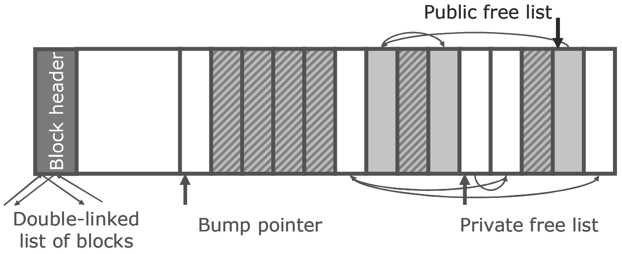


Figure 5: Structure of a memory block containing allocation objects of a specific size

Berger et al. [8] proved that allocators with pure private heaps cause unbounded memory blowup in producer-consumer applications. To avoid this, memory should be returned to the heap it was allocated from. In the TBB scalable allocator, an object is naturally returned to its enclosing block. However doing so means that a *foreign thread*[4] can interfere with operations of the owning thread, possibly leading to slowdown. To avoid that, two separate free lists are used for objects returned by the owner and by foreign threads.

Allocation requests are usually served from the *private free list* and so do not require synchronization; only when the request cannot be satisfied this way are the public free lists inspected. Unlike in McRT malloc [7], we do not make repatriation of objects completely non-blocking due to portability restrictions and stricter requirements; we use fine-grained locks that are distributed as much as possible.

EXPERIMENTAL RESULTS

In this section, we present performance data to evaluate the performance of both the Intel TBB task scheduler and the scalable memory allocator. All results were collected on a server system with two Quad-Core Intel® Xeon® processors X5355[5] running Red Hat Enterprise Linux 4 (update 4). We present data using 1 through 8 threads to show performance on both a small number of cores as well as to show the scalability beyond the number of cores available in a single multi-core processor today.

Performance of the Task Scheduler

In this section, we present the scalability of several benchmarks, highlighting the impact of continuation passing, scheduler bypass, and task recycling on the performance of each application.

Methodology

To evaluate the performance of the TBB scheduler as well as the impact of the manual optimization described above, we show results for applications using TBB without scheduling optimization (TBB); using only continuation passing (TBB+C); using continuation passing and scheduler bypass (TBB+CB); and using continuation passing, scheduler bypass, and task recycling (TBB+CBR). For each benchmark we show the speedups relative to an optimized serial implementation that does not use TBB.

Benchmark Descriptions

We use four applications to evaluate the performance of the task scheduler:

- **fibonacci.** The Fibonacci benchmark corresponds to the running example provided above in the description of the task scheduler. In our benchmark runs, we calculate the 50th Fibonacci number, with serial cutoffs of 12 and 20.
- **parallel_for.** This microbenchmark uses an Intel TBB parallel_for algorithm to iterate over a range of 100 million integers applying an empty loop body to each element. In the TBB library, all three scheduling optimizations are used by default. To allow the performance impact of the various optimizations to be

[4] A thread returning a memory object to the block owned by another thread.

[5] Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

measured, the implementation of parallel_for was modified to allow the selective disabling of specific optimizations.

- **sub_string_finder.** This benchmark is an example that is provided with the TBB library. The application calculates, for each position in a string, the location of the largest substring found elsewhere in the string that matches a string starting at the current position. The code uses the modified parallel_for described above to isolate the impact of the scheduling optimizations.
- **tacheon.** Tacheon is a 3D ray tracer that is distributed as another example with the TBB library. The code also uses the parallel_for algorithm modified to allow selective disabling of optimizations.

Benchmark Results

Figure 6 shows the performance of the Fibonacci example when executed on 1 through 8 threads on the aforementioned server. In the tests we used serial cutoffs of 12 and 20. When calculating the 50th Fibonacci number, the overhead of task creation is small when using a cutoff of 20, as shown by the speedup of 1 when using 1 thread. The scalability for this case is also excellent, with a speedup of nearly 8 on 8 threads.

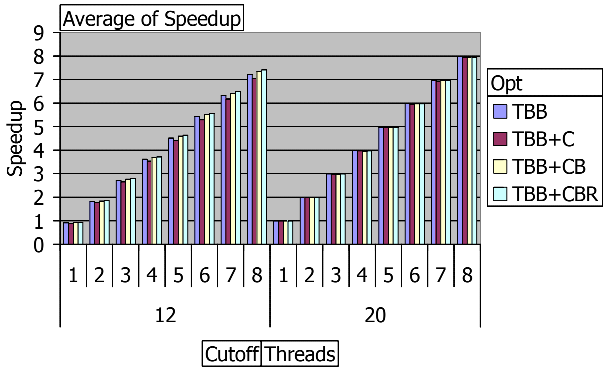


Figure 6: The speedup of the Fibonacci example when using different scheduling optimizations and serial cutoff values. The performance on 1 through 8 threads is reported for each configuration.

With a cutoff of 12, however, finer-grain tasks are created resulting in a noticeable scheduling overhead and a speedup of only 0.93 on 1 thread. With measurable overheads, the impact of the scheduling optimizations can also be seen. As discussed above, the use of continuation passing may provide additional opportunities for stealing but requires the allocation of additional task objects, often resulting in a slowdown. This effect is clearly seen in the TBB+C bars in Figure 6. However continuation passing also enables the scheduler bypass and task recycling optimizations, which when combined, result in speedups beyond the simple TBB case. On 8 threads, the speedup increases from 7.2 with no optimizations to 7.4 with all optimizations, an increase of approximately 3%.

The performance of the parallel_for microbenchmark is shown in Figure 7. The parallel_for algorithm creates tasks that apply a user-provided body to subranges of the user-provided range. When using the parallel_for algorithm, developers may explicitly specify a grainsize or choose to use the *auto_partitioner*. If a grainsize is specified, the default parallel_for algorithm recursively divides the provide range until the subranges are less than the grainsize. Tasks are created that apply the body to these subranges. If the auto_partitioner is used, the library adaptively tries to select a good partitioning of the range.

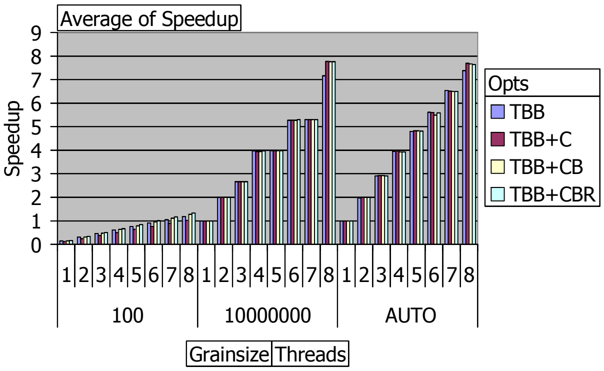


Figure 7: The speedup of the parallel_for benchmark when using different scheduling optimizations and grainsizes. The AUTO configurations use the auto_partitioner to divide the loop iterations; the other configurations use the provided grainsize parameter with the simple_paritioner. The performance is reported on 1 through 8 threads.

In Figure 7, results for a grainsize of 100, a grainsize of 10,000,000, and the auto_partitioner are shown. Again, for a large grainsize (and the correspondingly large-grain tasks) the overhead of the scheduler is negligible and the speedup on 8 threads is close to 8. Interestingly, the lack of available parallelism limits speedup even for large tasks, as demonstrated by the speedup increase with continuation passing over the base unoptimized case.

The fine-grain tasks, of only 100 iterations of an empty loop body, show high overhead (a speedup of 0.15 on 1 thread and 1.19 on 8 threads). Again because of the visibility of overheads, the impact of scheduler bypass and task recycling is clear on 1 through 8 threads. The speedup of 1.19 on 8 threads is improved to 1.34 when all three optimizations are applied.

Figures 8 and 9 present the performance of two larger, more realistic benchmarks. In both of these benchmarks, the performance using the default grainsize of 100 for sub_string_finder and of 50 for tacheon is measured as well as a grainsize of 1. In both applications, the scheduling overhead shown in the 1-thread case is small even when a grainsize of 1 is used. The scalability of both applications is also good, with a speedup of close to 8 for sub_string_finder and a speedup of 7.7 for tacheon.

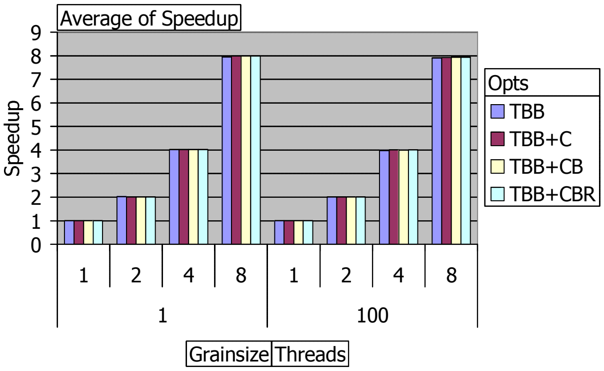


Figure 8: The speedup of the sub_string_finder example using different scheduling optimizations and grainsize parameters. The performance on 1, 2, 4, and 8 threads is presented.

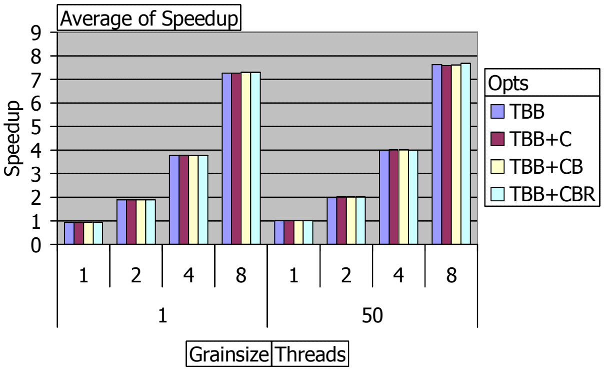


Figure 9: The speedup of the tacheon example using different scheduling optimizations and grainsize parameters. The performance on 1, 2, 4, and 8 threads is presented.

In summary, the scalability of the TBB scheduler is shown to allow linear speedups for several small benchmarks.[6] It is also clear that the overhead of the TBB scheduler is seen for fine-grain tasks (for example 100 iterations of an empty loop). When these overheads are visible, continuation passing alone often leads to a slowdown relative to the unoptimized case. However, continuation passing can be applied to enable scheduler bypass and task recycling, which are consistently shown to improve performance when scheduling fine-grain tasks.

Performance of Memory Allocation

In this section, we present the comparative performance data for the TBB scalable allocator and five other commercial and non-commercial memory allocators.

Memory Allocators being Compared

The TBB scalable memory allocator binaries were obtained from tbb20_010oss_lin.tar.gz package available at http://www.threadingbuildingblocks.org.

Other allocators in the comparison are these:

- The default memory allocator of GNU C runtime library (glibc) v2.3.4.
- Google's TCMalloc (google-perftools v0.92) from http://code.google.com/p/google-perftools built by gcc 3.4.6.
- Hoard v3.6.2 taken from http://www.hoard.org, also built by gcc 3.4.6.
- Memory Tuning System* (MTS) binaries provided by NewCode Technologies, Inc., http://www.newcodeinc.com.
- SmartHeap* for SMP binaries provided by MicroQuill, http://www.microquill.com.

Benchmark Description

When comparing memory allocators, it makes sense to use different tests that exercise different aspects of memory allocation routines. We used four benchmarks in our study: the Larson benchmark, the MTS demo test, and two internally developed microbenchmarks, speed-cross and false-sharing.

The **false-sharing micro-benchmark** was developed to check for the performance penalty due to false sharing induced by an allocator. Each thread repeatedly allocates a small object of a given size, then writes and reads every byte in the object many times in a loop and measures the time of the loop. The result is reported for every thread. If objects allocated by different threads share the same cache line, there should be a significant time penalty.

The **speed-cross micro-benchmark** was developed as a stress-test of the multi-threaded behavior of an allocator. Each thread repeatedly allocates a chunk of memory objects, touches each one by reading and writing a few bytes, and then transmits these objects in equal proportion to all other threads. Then each thread deallocates the objects it just received. Thus all objects are freed by foreign threads, and all subsequent allocations potentially

[6] The speedup of other applications will vary depending on application characteristics.

reuse these objects. The test reports average allocation and deallocation time per 1,000 objects.

Unlike the microbenchmarks intended to check specific aspects of memory allocation, the other two tests try to exercise memory in a more or less realistic way.

The **MTS demo test** was obtained from the MTS evaluation package. It attempts to mimic typical allocation behavior of applications by requesting few large objects, more medium-size objects, and significantly more small objects. The test measures elapsed time in seconds.

The **Larson benchmark** was originally developed by Larson and Krishnan [12] to model the allocation behavior of a multi-threaded server and test its throughput as the number of malloc and free pair operations per second. We took the benchmark from http://www.hoard.org.

Benchmark Results

The internal micro-benchmark data presented below were collected for objects of the machine word size, i.e., eight bytes on our test server.

The false-sharing benchmark demonstrates that of all the tested allocators, only the glibc allocator induces false sharing. Figure 10 shows execution time for various numbers of running threads as the percent of difference from the single-threaded run. While the test slowed down by 40-50% when executed with the default allocator, the difference is within 10% for all of the other allocators.

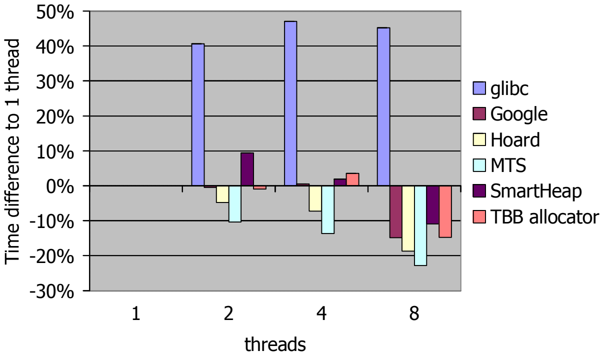


Figure 10: The difference in execution time of the false-sharing benchmark, running on 2, 4, and 8 threads, to the time of the single-threaded run, for various memory allocators

In addition, the test was faster with multiple threads which is especially well observed for eight threads. This effect could be possibly explained by decreased thread migration between cores when the number of active threads increases.

The speed-cross benchmark heavily stresses the allocators by freeing every object in a thread other than the one it was allocated; it's truly a worst-case test. In Figure 11, the summary time[7] of malloc and cross-thread free operations is shown for various numbers of threads. For allocators returning memory pieces to the heap of the allocating thread, the internal contention increases with the growing number of threads. The chart demonstrates that the TBB scalable allocator keeps being faster than the others with a growing number of threads and increasing contention.

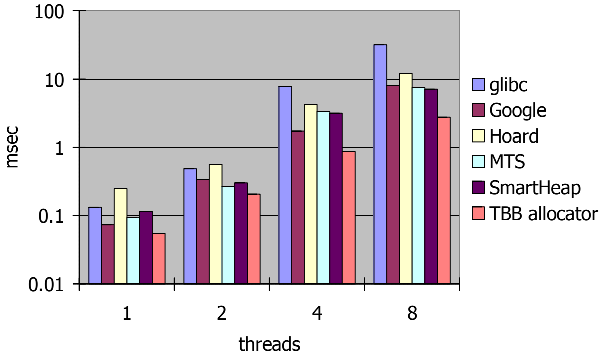


Figure 11: The average time to allocate and free 1,000 objects in the speed-cross benchmark is presented for 1, 2, 4, and 8 threads. The chart uses logarithmic scale.

Figure 12 demonstrates the elapsed time to run as reported by the MTS demo test.

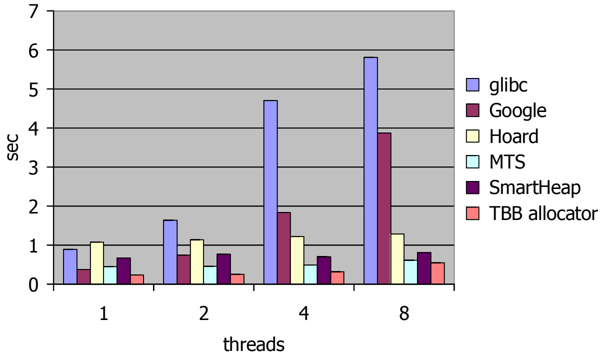


Figure 12: The elapsed time of the MTS demo test running on 1, 2, 4, and 8 threads, for various memory allocators

It is clearly seen that the test slows down as the number of threads increases for both the glibc malloc and Google's allocator; obviously their performance does not scale in this test. Other allocators scale well enough, though the test performance drops faster with the TBB allocator than with Hoard and the two commercial allocators. We are

[7] Due to nature of the benchmark, it separately collects data for malloc and free, then sums them up.

currently investigating the source of this performance drop.

The Larson benchmark results are shown in Figure 13. The benchmark parameters were set to allocate small size objects of 8 to 100 bytes. The TBB allocator scales linearly in this test with the best speedup slope. With 8 threads, it provides a 6x increase in throughput. Also note that the glibc malloc experienced a drop in throughput with multiple threads running. As in the other tests before, the Larson benchmark gives additional proof that the default glibc allocator can be a bottleneck in parallel code.

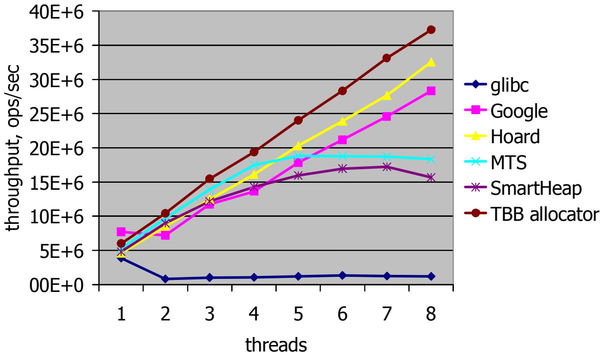


Figure 13: The throughput, in allocations per second, of the Larson benchmark running on 1 through 8 threads for various memory allocators

To summarize, all examined allocators except the glibc malloc showed their eligibility for parallel applications, and there is no single winner. While not always the best, the TBB scalable allocator performed competitively in all our tests.

Combined Performance of the Task Scheduler and the Scalable Allocator

In this section, we show the impact of the scalable allocator combined with an analysis of the impact of the task-scheduling optimizations. For this analysis, we use the tree sum example application provided in the TBB library distribution.

The tree sum application first generates a binary tree that contains nodes each holding a float value. It then performs a summation of the values in the tree. Both phases are done in parallel using TBB tasks, with a serial cutoff value below which the subtrees are allocated or summed sequentially.

Figure 14 shows the performance of the tree allocation phase of the benchmark when using both the scalable allocator and the default malloc implementation. Results are provided for a serial cutoff of both 100 nodes and 10,000 nodes.

First, it is clear that the allocation phase scales as additional threads are used only when the scalable allocator is employed. The performance of the standard malloc version degrades as additional threads are used.

Second, the impact of the scheduling optimizations is again demonstrated by the finer grained tasks (a cutoff of 10 nodes). There is an initial loss for employing continuation passing, but this loss is mitigated by the additional application of scheduler bypass and task recycling. And as expected, the larger tasks of 10,000 nodes show negligible impact from the optimizations.

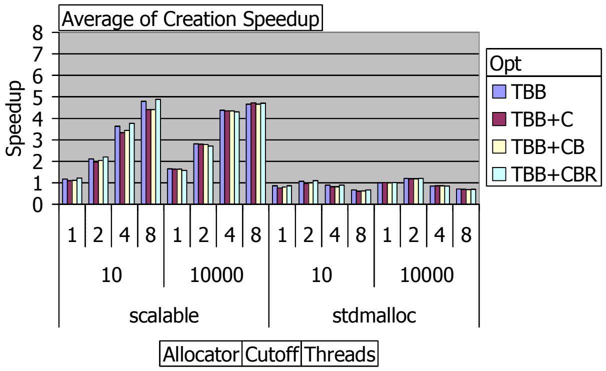


Figure 14: The speedup of the tree allocation phase of the tree_sum example using the scalable allocator and the default malloc implementation. The impact of the various scheduling optimizations is also shown. The performance on 1, 2, 4, and 8 threads is shown when using a serial cutoff of 10 and 10,000.

Figure 15 shows the performance of the summation phase of tree sum. Because of the locality and false-sharing benefits of the scalable allocator, the performance and scalability of the computation are also better than with the standard malloc implementation. The impact of the manual scheduling optimizations is also seen here for fine-grain tasks, and it is negligible for large-grain tasks.

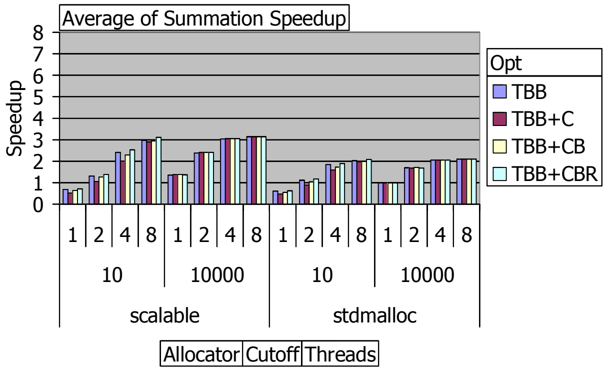


Figure 15: The speedup of the tree summation phase of the tree_sum example using the scalable allocator and the default malloc implementation. The impact of the various scheduling optimizations is also shown. The performance on 1, 2, 4, and 8 threads is shown when using a serial cutoff of 10 and 10,000.

CONCLUSION

Intel Threading Building Blocks is a C++ template library designed to raise the level of abstraction for parallelism as developers port their code to multi-core platforms. Starting with the 2.0 version, Intel TBB is also provided at www.threadingbuildingblocks.org as an open-source project licensed under the GNU Public License.

Two key features of the library are its work-stealing task scheduler and scalable memory allocator. Both of these systems reduce the need of users to understand the many complex issues related to multi-core performance and scalability.

In the TBB Task Scheduler section, we provided an overview of the task scheduler design and outlined several manual optimizations that users can perform to improve the performance of the scheduler when executing fine-grain tasks.

In the Scalable Memory Allocation section, we described the motivation for and implementation of the scalable memory allocator, highlighting the design characteristics that decrease synchronization, increase locality, and avoid false sharing.

In the Experimental Results section, we explored the performance of a number of benchmarks on a server with two Quad-Core Intel Xeon processors. We showed that the overhead of work stealing is low for large-grain tasks, and that the manual optimizations described in this paper offer a small but noticeable improvement when scheduling fine-grain tasks.

In our evaluation of scalable memory allocators, the TBB scalable allocator was shown to be competitive with several commercial and research allocators.

In an analysis of an example that studied the combined effects of the scheduling optimizations and the scalable allocator, the use of the scalable allocator showed a large impact for both small- and large-grain tasks. The scheduling optimizations were shown to have a small performance impact for the small-grain tasks and a negligible impact on the scheduling of the larger-grain tasks. This confirms the assertion that memory allocation can sometimes be a limiting factor in the scalability of parallel applications and that a scalable allocator can remove this bottleneck.

With the growing availability of multi-core platforms, it is becoming imperative for performance-oriented developers to thread their code. Intel TBB, built on its work-stealing task scheduler and scalable memory allocator, offers an exciting solution to ease the burden of this transition.

ACKNOWLEDGMENTS

We first must acknowledge the architect of Intel Threading Building Blocks, Arch Robison, who did much of the background research and design of the task scheduler. We also acknowledge the other members of the TBB development team: Elena Gavrina, Chris Huson, Anton Malakhov, Andrey Marochko, Alexey Murashov, Anton Pegushin, Vladimir Polin, and Dave Poulsen. As mentioned in the paper, the TBB scalable allocator is a productization of the McRT allocator, so we acknowledge the contributions of Richard Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. We thank NewCode Technologies, Inc. and MicroQuill for providing evaluation copies of their memory allocators. We also thank the reviewers of this paper for their many useful and insightful comments.

REFERENCES

[1] James Reinders, *Intel Threading Building Blocks*, O'Reilly Media, Inc, Sebastopol, CA, 2007.

[2] Robert D. Blumofe and Charles E. Leiserson, "Scheduling Multithreaded Computations by Work-Stealing," in *Proceedings of the 35th Annual IEEE Conference on Foundations of Computer Science,* Sante Fe, New Mexico, November 20–22, 1994.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (PPoPP '95), Santa Barbara, California, July 19–21, 1995.

[4] Doug Lea, “A Memory Allocator,” at http://gee.cs.oswego.edu/dl/html/malloc.html

[5] David Detlefs, Al Dosser, and Benjamin Zorn, “Memory Allocation Costs in Large C and C++ Programs,” *Software Practice and Experience,* 24(6), pp. 527–542, June 1994.

[6] Gene Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities,” *AFIPS Conference Proceedings,* (30), pp. 483–485, 1967.

[7] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg, “McRT-Malloc – A Scalable Transactional Memory Allocator,” in *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pp. 74–83, Ottawa, Canada, June 2006.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson, “Hoard: A scalable memory allocator for multithreaded applications,” in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems,* pp. 117–128, November 2000.

[9] Maged M. Michael, “Scalable Lock-free Dynamic Memory Allocation,” in *Proceedings of the ACMSIGPLAN 2004 Conference on Programming Language Design and Implementation*, pp. 35–46, Washington, D.C., June 2004.

[10] Yi Feng and Emery D. Berger, “A Locality-Improving Dynamic Memory Allocator,” in *Proceedings of the Third Annual ACM SIGPLAN Workshop on Memory Systems Performance*, pp. 68–77, Chicago, IL, June 2005.

[11] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos, “Scalable Locality-Conscious Multithreaded Memory Allocation,” in *Proceedings of the 2006 ACM SIGPLAN International Symposium on Memory Management*, pp. 84–94, Ottawa, Canada, June 2006.

[12] Paul Larson and Murali Krishnan, “Memory Allocation for Long-Running Server Applications,” in *Proceedings of the First International Symposium on Memory Management,* pp. 176–185, Vancouver, BC, October 1998.

[13] Michael Halbherr, Yuli Zhou and Christopher F. Joerg, “MIMD-style parallel programming with continuation-passing threads,” in *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications,* Capri, Italy, September 1994.

[14] W. Shu and L. V. Kale, “Chare Kernel – A Runtime Support System for Parallel Computations,” *Journal of Parallel and Distributed Computing,* 11(3), Academic Press, pp. 198–211, 1991.

[15] Jeffrey Richter, “.NET: The CLRs Thread Pool,” *msdn Magazine,* 18(6), June 2003.

[16] Wolfram Gloger, “Dynamic Memory Allocator Implementations in Linux System Libraries,” at http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html

[17] Bratin Saha et al., “Enabling scalability and performance in a large scale CMP environment,” in *Proceedings of the 2007 conference on EuroSys*, pp. 73–86, Lisbon, Portugal, March 2007.

[18] Michael Voss, “Demystify Scalable Parallelism with Intel Threading Building Block’s Generic Parallel Algorithms,” DevX.com, Jupiter Media, October 2006, at http://www.devx.com/cplus/Article/32935.

[19] Michael Voss, “Enable Safe, Scalable Parallelism with Intel Threading Building Block’s Cocurrent Containers,” DevX.com, Jupiter Media, December 2006, at http://www.devx.com/cplus/Article/33334.

AUTHORS’ BIOGRAPHIES

Alexey Kukanov is a Senior Software Engineer in Intel’s Performance Analysis and Threading Lab. Since joining Intel in 2000, he worked on a few software products. Finally, his interests in C++, library development and multi-threading have been happily combined in the TBB project where he is now one of leading developers. Alexey received an M.S. equivalent degree in Applied Math from Nizhny Novgorod State University. When he has some free time he likes playing billiards and volleyball. His e-mail is alexey.kukanov at intel.com.

Michael Voss is a Senior Staff Software Engineer in Intel’s Performance Analysis and Threading Lab, where he is currently one of the lead developers of Intel Threading Building Blocks. Michael is also an adjunct professor in the Department of Electrical and Computer Engineering at the University of Toronto, where he taught from 2001–2005. His interests include languages, tools, and compilers for parallel computing. Michael received his Ph.D. and M.S.E.E degrees in Electrical Engineering from Purdue University in 2001 and 1997, respectively. His e-mail is michael.j.voss at intel.com.

Methodology, Tools, and Techniques to Parallelize Large-Scale Applications: A Case Study

Knud J. Kirkegaard, Software and Solutions Group, Intel Corporation
Mohammad R. Haghighat, Software and Solutions Group, Intel Corporation
Ravi Narayanaswamy, Software and Solutions Group, Intel Corporation
Bhanu Shankar, Software and Solutions Group, Intel Corporation
Neil Faiman, Software and Solutions Group, Intel Corporation
David C. Sehr, Software and Solutions Group, Intel Corporation

Index words: multi-core, parallelism, threading, data dependence, privatization

ABSTRACT

Multi-core processors are now mainstream, while many-core architectures are arriving. Yet getting general-purpose software ready to take full advantage of the available hardware parallelism remains a challenge. There are, in fact, very few success stories of semi-automatic parallelization of large-scale integer applications outside the high-performance computing (HPC) and transaction processing domains. In this paper, we report on such a success story: threading the Intel® C++ Compiler [3] which resulted in an average 2x speedup in compiling a range of CPU2000 benchmarks. We present the methodology and tools that enabled us to achieve this success. We believe our approach is generally applicable to threading a large class of applications.

INTRODUCTION

In this paper we focus primarily on the techniques used to parallelize an application, the tools that facilitate the parallelization, and the new insights this approach yielded. Techniques that proved helpful in our work are at the core of a comprehensive solution suite Intel is developing to assist software developers discover and exploit parallelism in their applications. The generally applicable source changes necessary to make the compiler thread safe are also categorized and described. As expected, good software engineering principles such as modularization, data abstraction, and information hiding ease the process of threading an application. We also describe how we automated repetitive source changes. To make it feasible to apply all the source changes necessary for an application of this size, where the threaded loop spans hundreds of modules covering hundreds of thousands of lines of code with extensive use of macros, semi-automated script tools were developed. It is easy to get overwhelmed by the data dependence complexity and size when starting to thread existing serial applications, but as we hope to illustrate in this case study, with the help of Intel's threading tools and a systematic approach, it is possible to achieve large application threading with a reasonable amount of effort and time.

The threading effort, involving a small team over a relatively short period of time, successfully yielded a working parallelized compiler. Although work remains to be done in tuning the resulting application, we also discuss in this paper the impact of different thread scheduling algorithms and the speedups achieved. We also briefly discuss the issues involved in maintaining a thread-safe application.

DESCRIPTION OF THE APPLICATION

The Intel Compiler is a large non-numeric application that compiles C/C++ and Fortran applications for a variety of Intel® platforms including the IA-32 architecture, the Intel® 64 Architecture, and the Itanium® processor. Despite having evolved over the years to target new Intel® processors and platforms, parallelization of the compiler itself was not an initial design goal. As such, the compiler has characteristics similar to other large integer applications that need to be parallelized in order to take full advantage of multi-core platforms. At the Intel Compiler Lab, we parallelized the Intel Compiler and achieved great performance results. One of our goals was to fully understand the issues that application developers encounter when parallelizing a large-scale application.

We chose to thread the Intel C++ Compiler for a number of reasons.

First, we had detailed knowledge about the application. We strongly believe that to thread an application successfully, it is important to involve the application architects as they tend to know what to parallelize and what not to parallelize. Moreover, an in-depth knowledge of the application global data is crucial.

Second, the compiler has evolved over the years and therefore is a good proxy for real-world, legacy product applications. It is a mature integer application that was not initially designed to be thread safe.

Third, by choosing a non-numeric application outside the traditional high-performance computing (HPC) domain, we strived to address the challenges other application developers would encounter when undertaking a similar task. A particularly interesting challenge is that the potential parallel region spans hundreds of source modules containing millions of lines of code. In contrast, in typical HPC applications, the parallel loops are contained within one module or even just one function.

Finally, there is an inherent scalable parallelism in what compilers do. By using performance analysis tools and built-in timers in the application itself, we found that the region we intended to parallelize accounts for up to 80% of the application time in compiling a number of benchmarks. With infinite parallelism there is a theoretical speedup of 5x as dictated by Amdahl's law. If S is the fraction of the program that is serial and N is the number of available processors, the speedup through parallelism is 1/(S + ((1-S)/N)), and the theoretical speedup limit is 1/S. For example, if 80% of the application time is in the parallel region, then S equals 0.20, and assuming N → ∞, we get at best a 5x speedup through threading.

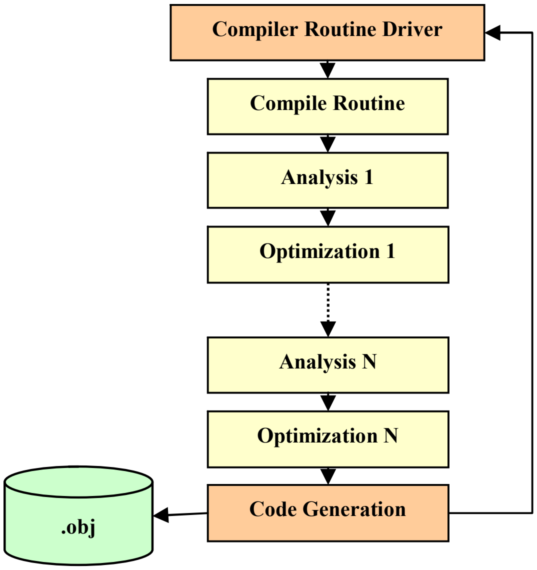


Figure 1: Serial execution of the compiler driver loop

The basic flow of the compiler is shown in Figure 1. After the front-end parses the input program into an intermediate representation, the compiler iterates over the functions of each module. At each iteration, the compiler translates the code of the corresponding function, applies a series of optimizations to the intermediate representation, and finally generates code for the function. We observe that each routine compilation is logically independent of each other; that is, we can change the order in which routines are compiled without affecting the correctness of the program and therefore it is legal to parallelize the loop that compiles each individual routine. This loop spans almost 200 source modules containing roughly half a million lines of mostly C source code. The flow of the parallelized compiler is shown in Figure 2.

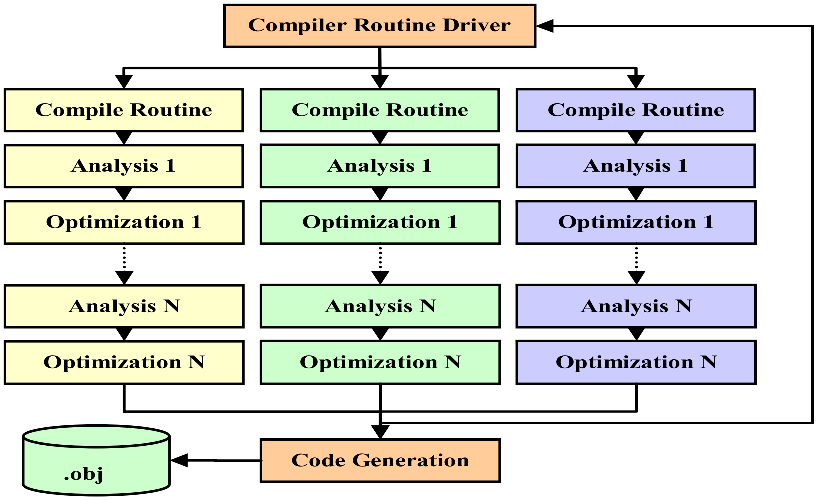


Figure 2: Parallel execution of the compiler driver loop

Table 1: The execution profile of the compiler across its loop hierarchy

%Ticks	Ticks	Entry	Exit	File:Line:Col	Function:Line
51.7	675952724	1	1	ip/placement.c:657:39	compiler_driver:276
18.0	235966016	14782	14782	fe/lexical.c:9589:8	get_token:9509
16.1	211070764	1	1	fe/decls.c:14518:7	translation_unit:14480
6.5	84369904	1	1	intrin/intrin.c:1536:5	intrin_process:1517
4.7	61261408	2	2	fe/code.c:3474:3	dump_routines:3423
4.5	59420636	87	87	il/verify.c:3014:5	verify:3011
4.2	54363924	217	217	fe/preproc.c:460:5	skip_endif:442
4.1	54079124	4228	4228	fe/lexical.c:6706:8	skip_space:6661
3.9	51635388	27	27	fe/lexical.c:4449:5	search_input:4416

Of course, we could have parallelized the compiler at a higher or a lower level. The highest level would simply be to compile the modules of an application in parallel, as with a parallel make file. This scheme is very simple to implement. However, it can easily run into load-balancing problems when the application's modules have widely varying sizes. It also fails when the build uses "link time compilation," an important feature of our compiler. Link time compilation pre-compiles the individual modules of the application into intermediate representations and then processes all the intermediate representations at once in a single execution of the compiler, making it possible to obtain the benefits of inter-procedural optimizations across the entire application.

At a lower level, we could have looked at smaller potential parallel regions, such as individual optimization phases. It might be easier to parallelize these phases than to parallelize the entire compiler driver loop, but any one piece would have accounted for only a small fraction of the total compilation time. Therefore, it would have been necessary to parallelize many smaller pieces to get any significant benefit from threading. Furthermore, working with the outermost driver loop allowed us to learn more about the problems of threading very large applications.

THE THREADING METHODOLOGY

We followed a threading methodology that consists of the following four basic steps:

1. Discovering parallelism
2. Expressing parallelism
3. Debugging the threaded code
4. Tuning the threaded code

In the first step, the application architect needs to discover the parallelism that is available in the application. Tools that provide loop-profiling capabilities can be used. One would need to know the execution profile of the application across its loops. This includes both the loops in the program control-flow graph as well as the loops in its call graph. In our case, a significant majority of the execution time of the compiler, as explained before, is spent in the body of the compiler driver loop. As contributing architects of the compiler, we knew where that loop was, but we found a loop profiling capability of the compiler generally helpful for threading. Through an option, the compiler instruments the generated binaries with timing instructions before and after program loops and functions. The execution time profile of the compiler across its loop graph is shown in Table 1. This option may also be provided through dynamic instrumentation tools such as the Intel VTune™ Performance Analyzer [5]. The application architect can go through the application loops in a top-down fashion ordered by the total contribution of the loop to the execution of the application. If, intuitively, the loop has parallelism potential, then the architect would need to know how many data dependence violations would be violated should that loop be parallelized.

Data Dependence

The notion of *data dependence* captures the most important properties of a program for parallel execution at all levels [1, 6]. At the loop level, the dependence relation is defined in three categories as follows.

1. If an iteration of a loop writes to a memory location that is later read in another iteration of the loop, we say that the second iteration is *flow-dependent* on the first iteration.

S1: x = …
S2: … = x

2. If the first iteration reads from a location that is later modified in another iteration of the loop, we say that the second iteration is *anti-dependent* on the first iteration.

   S1: … = x
   S2: x = …

3. Two iterations of a loop are *output-dependent* on each other if both write to the same memory location.

   S1: x = …
   S2: x = …

Data dependence relations are often called *hazards* or *data races*. Flow dependence, anti-dependence, and output dependence relations are equivalent to Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW), respectively.

A loop that contains no dependence relations can be parallelized. On the other hand, parallelizing a loop that contains any of these dependence relations may cause invalid results. However, it can be shown that if a loop contains only anti- and output-dependence relations, it can be parallelized with the proper code change [1].

Therefore, in order to parallelize an application, the application architect needs a tool to identify the dependence relations between its possible threads of execution such as various iterations of its loops. In our threading experience, we used the Intel® Thread Checker [2, 4], a software tool that helps developers detect the race conditions [7, 8] in their threaded applications. Among its many features, Thread Checker has a mode of operation, called *projection mode*, which is particularly helpful for parallelization. In this mode, the user can mark a sequential loop as a parallel loop. Thread Checker will run the code sequentially, but with some additional bookkeeping to reveal the race conditions that would occur should that loop actually run in parallel. This mode is extremely helpful in parallelization as it allows the sequential application to run to completion while the information about its possible threaded execution is being collected. More specifically, in spite of the data dependence violations in the parallel execution of the application, Thread Checker's projection mode does not crash due to such violations. We marked the compiler driver loop as a parallel loop and ran it under the control of Thread Checker on a small test program that included a single file with a few functions, conditional statements, and loops.

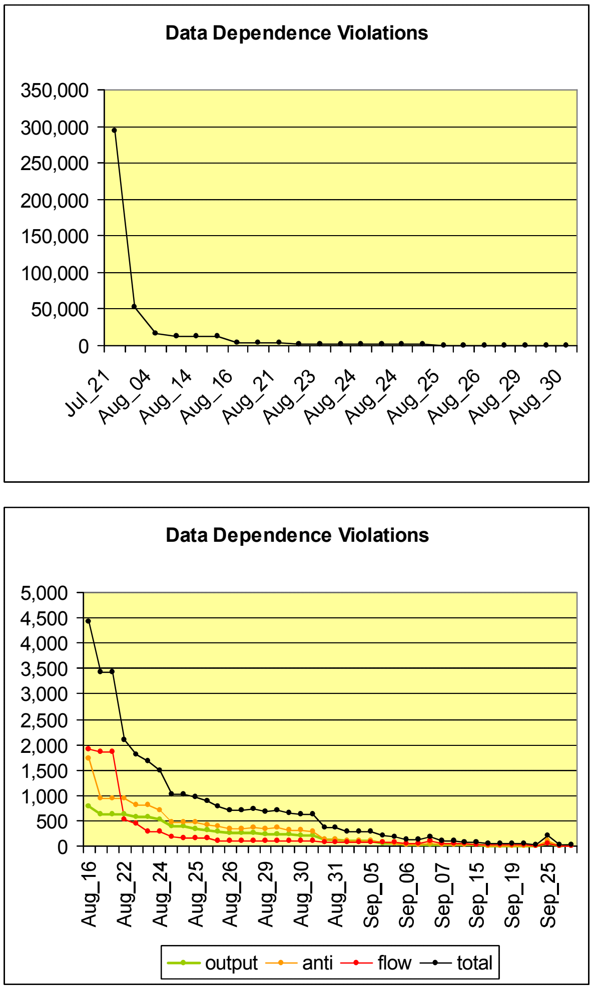


Figure 3: Progress of elimination of dependence violations over time

We also used the Intel Compiler code-coverage tool to make sure that our simple test resulted in reasonably good coverage of the compiler source code. In particular, we made sure that most of the critical components of the compiler including its various optimizations were exercised when compiling our test program. One should note that dynamic analysis tools, such as Thread Checker, typically provide information only about what occurs in a particular instance of program execution as opposed to static tools that may be able to provide information about what can possibly happen in the program execution in the general case. Thus, the lack of dynamic dependence violations does not necessarily imply thread correctness. The use of the code-coverage tool alleviates this problem to some extent. If one does not observe any dependence violation in a piece of code, and the coverage information reveals that the code was not in fact executed, then nothing can be inferred about the possible dependence relations in that piece of code. The first run of our

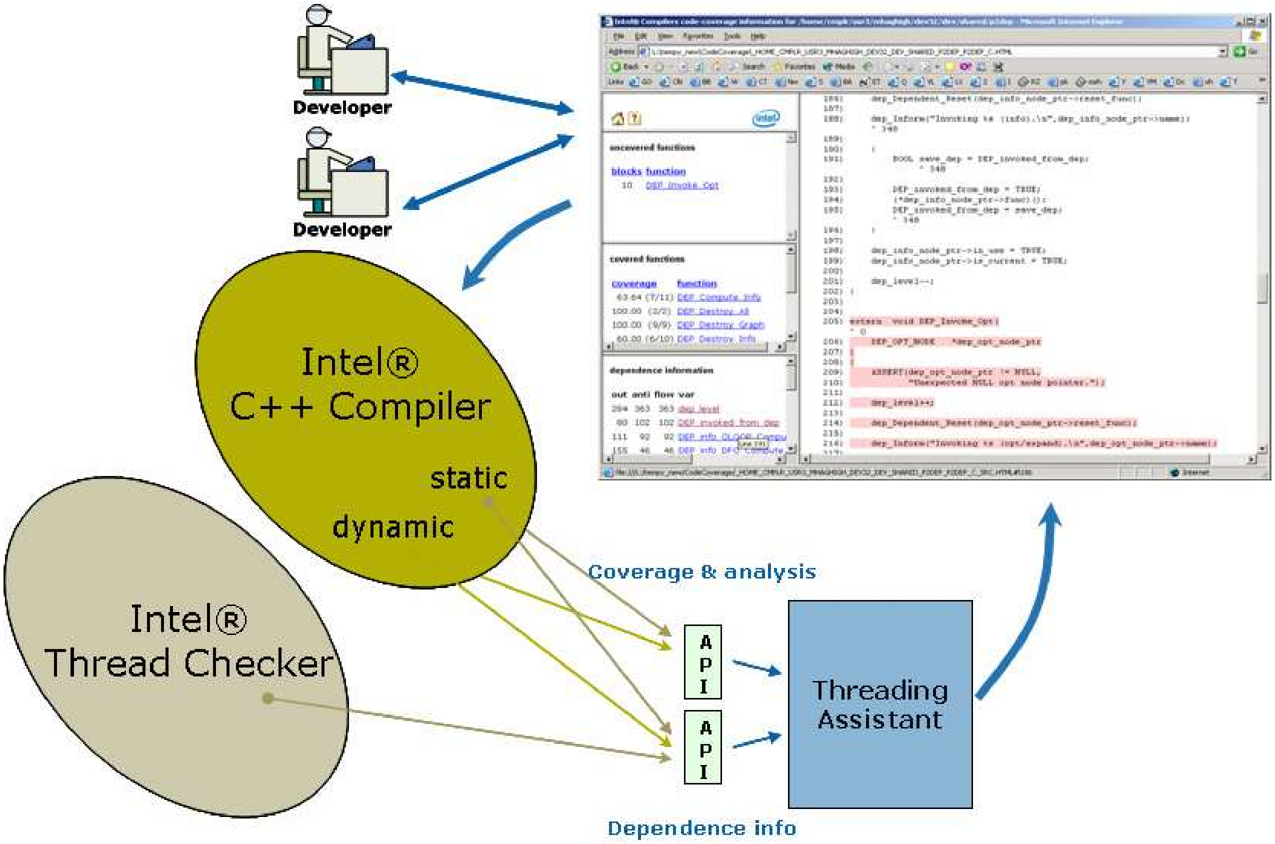


Figure 4: Flow of the iterative process of dependence-violation elimination

instrumented compiler under Thread Checker took several hours to complete and resulted in about 300,000 data-dependence violations. Such an error size is well above the comfort zone of most of the available race-detection tools.

Managing the Size Problem

The key to managing the large dependence problem size is controlling the precision of the generated analysis data. In the early phases of the threading effort, one may not need all the details about every individual dependence violation that is detected, including information about the source position of the two memory accesses that are involved and the call stack of each of them. At a later point, however, such information may actually be crucial to figure out the exact conditions under which the violation occurs.

Thread Checker already supported several useful filtering capabilities, such as filtering based on file names, variable names, and so on. It also summarizes the violations that have identical first memory access source position and base, and those that have identical second memory access source position and base. This filter effectively groups the violations that occur when processing the data in a given array in a single loop with the same dependence distance. In addition to the existing filters that Thread Checker supports, we developed a new filter that proved very effective at grouping the violations that map to different source files and functions and thus reduced the problem size dramatically. In this filter, we grouped together the violations whose base addresses were identical, irrespective of their source file positions and functions. One can think of this heuristic as projecting the dependence information based on its data structure as opposed to based on the code. We then picked the source position of the first such violation as the representative of that group of violations and summed all the violations in that group. Using this technique, we immediately realized that approximately 65% of the violations corresponded to the compiler memory pool data structure. What we lost in this filter is all the details about every individual violation, but what we learned was sufficient to guide us to make the pool thread safe and eliminate almost 200,000 dependence violations with a small number of changes to the source code. After fixing this problem, the subsequent instrumented runs not only have a much smaller problem size but also a much shorter turnaround time. The reason for this is that the runtime overhead of race detection depends on the number of violations, and by eliminating the violations in a prioritized fashion, we constantly speedup the process of the next iteration. Figure 3 shows the number of dependence violations over time.

The interactive development environment we created to assist us in the parallelization effort is illustrated in Figure 4. The main components of this platform are Intel Thread Checker, Intel Compiler, and Intel Compiler's code-coverage tool. In this framework the dynamic dependence diagnostics produced by the Intel Thread Checker and the dynamic code-coverage information generated by the instrumented binaries are combined with the static information provided by the Intel compiler to collectively assist the parallelization effort. The communication of information between these components is facilitated by means of well-defined APIs. The collected information is then assimilated by our Threading-Assistant analyzer to produce a compact set of dependence violation diagnostics and threading hints to the developers. The parallelization process is iterative and may require several iterations before all the dependence violations are eliminated and thread-safe code is obtained.

Making the Application Thread Safe

After identifying the loop to be parallelized in the *Discover* step of the threading methodology the application must be made thread safe with respect to that loop. Identifying all the global data with dependence relations and effectively privatizing them was by far the largest part of our effort and is a challenge for an application of this type. We spent about 10 person months to achieve thread safety for the compiler at the optimization levels chosen for our prototype project. In order to achieve this goal in the given project time frame, we not only relied on threading tools but also developed scripting tools to assist us in applying the needed source-code changes semi-automatically. Looking at the global data dependence violations and knowing the modular structure of the compiler, we found it useful to categorize statically allocated global data, as opposed to heap allocated global data, into three categories. For each category of global data we have a method for making the data thread safe:

Global Data with Dependence Relations

Initially, we attempt to rewrite the code in these data to eliminate the data dependence, and if that is not possible we have to apply locks to synchronize the access to the global data.

Global Data Defined Outside the Loop

This category includes global data that are defined outside the parallel loop and only read inside the loop. This is a thread-safe usage of global data and doesn't require any rewrite. The main issue is to ensure that the usage of the global data remains thread safe.

Global Data with Restricted Scope

This category consists of global data that could have been declared as constant or as stack variables. If it is possible to rewrite global data to be constant or as stack variables they become thread safe automatically. This, furthermore, improves the software engineering aspect of the application.

In the first category, where we have flow-dependence relations, we found there were many false flow-dependence relations that can be eliminated by privatizing the data and thereby improving the software engineering. One example is global data shared across loop iterations where the data need to be reset to proper initial values for each iteration of the loop. We found it useful to categorize the global data with data dependence relations into four sub categories:

1. Synchronized
2. Mutable
3. Persistent
4. Transient

The synchronized category includes the global data that require locks for controlled synchronized accesses. It is not possible to privatize such data without extensive changes. Examples of global data that require synchronization are input/output operation and heap allocation management.

The mutable category contains the global data that generally are defined before the parallelized loop, but they may be modified by an iteration of the loop (only to be reset to the original value before the next iteration). Mutable data are privatized by creating a thread-private copy of the data for each of the iterations. This has the additional advantage that there is no longer a need to restore values for the next loop iteration, if data were modified. Furthermore, this helps improve maintainability of the code by eliminating the code necessary to restore values.

The persistent category comprises the global data that are defined and used in each thread but do not have any cross iteration dependence relations. In the case of the compiler, examples of data in the persistent category include the intermediate representations for routine statements, expressions, symbol tables, control flow graphs, etc. The lifetime of the global data in the persistent category spans the entire thread. They are allocated and initialized after thread creation and freed before thread termination. Allocated persistent data are assigned and accessed through a thread-private pointer. In object-oriented terms, the state object is constructed after thread creation and destroyed before thread termination.

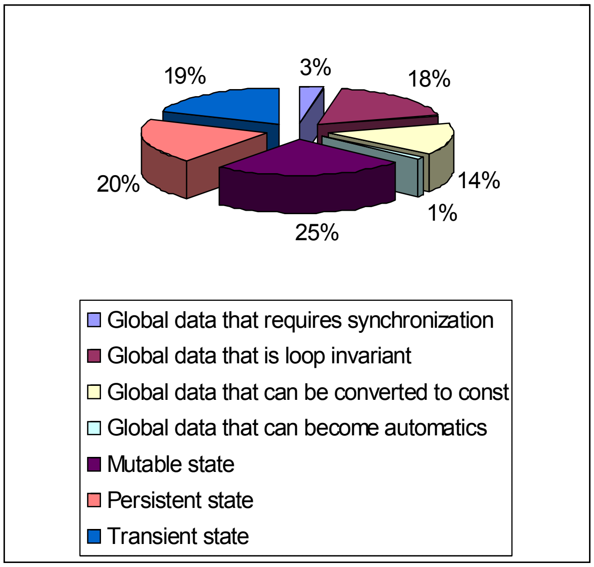


Figure 5: Breakdown of the global data

The transient category consists of the global data that are defined and used only within a certain phase of the threaded region, for example global data that are used to do constant propagation. Global data in the transient category are allocated on entry to a phase and freed on exit from that phase. In general, the transient state is allocated on the stack and is assigned and accessed through a thread-private pointer.

We chose to have a thread-private pointer for the persistent state as well as for each transient state for several reasons. First, on some systems there is a limit to the size of thread local storage; therefore, all the state objects could not be made thread-local. Furthermore, to make the source code changes manageable, it is convenient to have a thread-local pointer instead of adding state arguments to each routine to pass around persistent and transient state objects.

The compiler uses a lot of global state either as file-scope static variables or external global objects. In our case, global variable references are generally direct. Our semi-automatic source transformation tool takes a compiler-generated listing of global variables defined in each module and from that creates structures for transient and persistent state objects. The tool also automatically redefines those global variables as macros with the proper implementations; that is, a dereference through a thread-local pointer to a field in a state object. This relieves us from the tedious and error-prone task of manually modifying all references to those variables.

By creating persistent and transient state objects as structures, we also help improve software engineering by organizing global data into logical objects that have well-defined lifetimes.

Of all the global data that needed to be privatized we only had to create synchronized access for about 3% of the global data. The breakdown of the classification of the remaining global data is illustrated in Figure 5.

Another major task in working with data-race detection tools is training them to understand customized memory pool operations that behave like `malloc` and `free`. It is common to allocate a memory block and use it in an iteration and free it in the same iteration. When a subsequent iteration allocates memory, it may get part or all of the freed block. If the data-race detection tool is not able to recognize the malloc-free pattern, it may report a large number of false dependence violations. The Intel Thread Checker recognizes a class of such operations. It also provides a mechanism through which the user can communicate this information with its runtime. This is achieved by means of an API call that passes a starting memory address and by the number of bytes to be considered as newly allocated memory chunks. In this way, the application architect asserts that the dependence relations across the specified barrier can safely be ignored. This is a simple mechanism; yet, it is capable of handling very complicated memory pool management systems.

PERFORMANCE RESULTS

After our compiler was successfully threaded and debugged, we spent some time in tuning its performance. Of particular importance was the choice of thread scheduling. We conducted many experiments with various parallel-loop scheduling policies. From the parallel-loop scheduling schemes supported by OpenMP*, self-scheduling provided the best performance. In addition, we implemented a scheduling policy that consistently outperformed self scheduling. The policy took advantage of the information that the compiler has about the functions it needs to compile. As part of parsing the input file and creating the intermediate language, the compiler has a substantial amount of information about the structure and the size of each function. We used this information as a static estimate of the time it would take to compile each function. We then grouped together functions in as many chunks as the number of threads or available cores in such a way that the workload of each chunk is almost the same. Through this technique we avoided the load imbalance problem. Figure 6 shows the parallel speedup we achieved in comparison to the theoretical speedup limit. The results are based on our experiments on a 4-socket dual-core system–a total of eight processors. We also spent some time in making sure lock contention was reduced by proper choice of locking. We were pleased with the final parallel performance of the threaded compiler as it approached the theoretical limit of parallel performance as dictated by Amdahl's law. Figure 6 shows the speedup of the threaded compiler compared to the original sequential compiler when compiling the SPEC CPU2000 benchmarks.

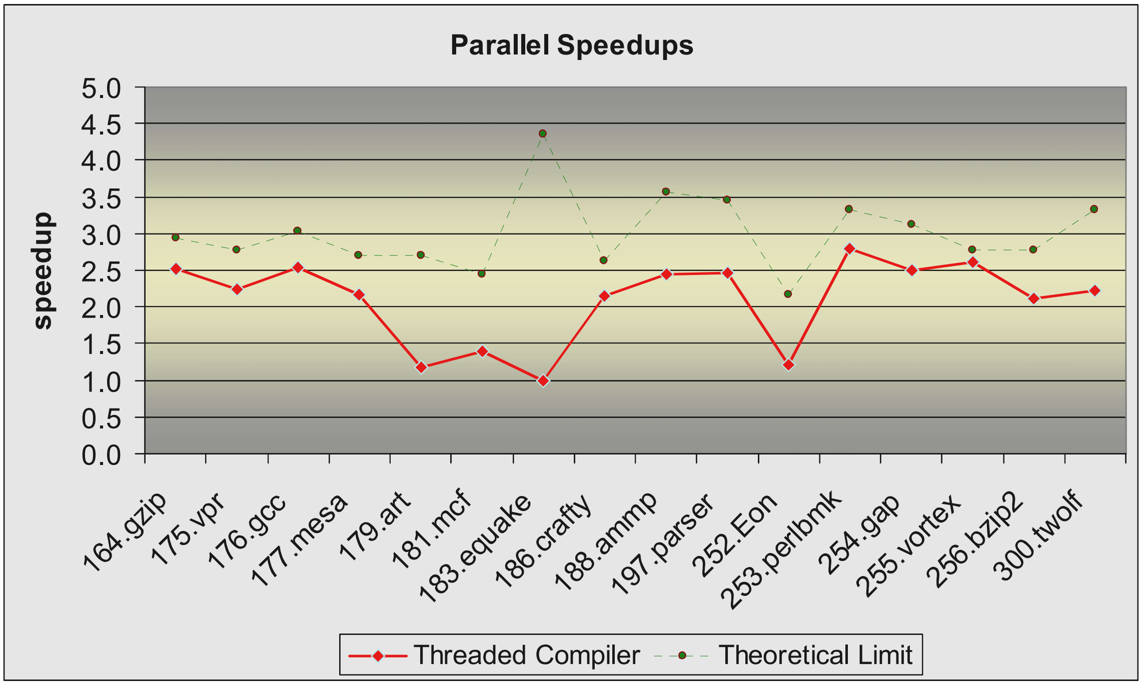


Figure 6: Parallel speedups of compiling CPU2000 benchmarks

CONCLUSION

We conclude that advancements in threading analysis tools have made parallelization of complex applications an easier task than what it was a decade or two ago. The overhead of the required instrumentation to perform the dynamic dependence checking has become affordable on modern microprocessors. Effective summarization and filtering of data dependence violations play a key role in managing the large problem size. We also found that semi-automatic mechanisms provide crucial help in accomplishing the repetitive and error-prone task of source code changes. Moreover, we found that good software engineering practices make threading easier.

ACKNOWLEDGEMENTS

We thank Zhiqiang Ma, Paul Petersen, and Victoria Gromova for their help with using and enhancing the Intel threading tools. Thanks also to Diana King, Sergey Kozhukhov, Suriya Madras-Subramanian, Xinmin Tian, and Ravi Ayyagari for their help with the static instrumentation of the Intel compiler. Throughout the entire project, we benefited from the mentorship of Kevin J. Smith.

REFERENCES

[1] Allen, R., and Kennedy, K., *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, San Francisco, CA, 2002.

[2] Banerjee, U., Bliss. B., Ma, Z., and Petersen, P., "Unraveling Data Race Detection in the Intel® Thread Checker," presented at the *First Workshop on Software Tools for Multi-core Systems (STMCS)*, in conjunction *with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, March 26, 2006, Manhattan, New York, NY.

[3] Intel® Compilers http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm

[4] Intel® Threading Analysis Tools http://www3.intel.com/cd/software/products/asmo-na/eng/threading/219785.htm

[5] Intel VTune™ Performance Analyzer http://www3.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm

[6] Kuck, D.J., R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *8th annual ACM Symposium on*

Principles of Programming Languages, pp. 207–218, Jan. 26-28, 1981.

[7] Lamport, L., "Time, Clocks, and the Ordering of events in a Distributed System," *Communications of the ACM, Vol. 21, No. 7*, July 1978, pp. 558–565

[8] Lee, E. A., "The Problem with Threads," *IEEE Computer Society, Computer*, May 2006, Volume 39, Number 5.

AUTHORS' BIOGRAPHIES

Knud J. Kirkegaard is a Principal Engineer in the Intel Compiler Laboratory. He currently works on compiler optimizations for the Intel architectures. Since he joined Intel, he has worked on scalar optimizations, interprocedural optimizations, and profile guided optimizations for IA-32, Intel 64, and the Itanium processor family. His current interests are in optimized C++ code, compiler architecture, and thread-safe applications. He has an M.S. degree in Information and Control Systems Engineering from Aalborg University, Denmark. His e-mail is knud.j.kirkegaard at intel.com.

Mohammad Reza Haghighat is a Principal Engineer at Intel and the architect of Intel Compiler's code-coverage and test-prioritization tools. He is a threading expert and the author of *Symbolic Analysis for Parallelizing Compilers*, a book based on his pioneering Ph.D. research at the University of Illinois at Urbana-Champaign in the early 90s. Mohammad was the lead developer of one of the first Java JIT-Compilers and also has extensive experience in the performance aspects of database systems. More recently, he has been doing advanced development in the emerging Web 2.0 technologies such as AJAX and PHP. His e-mail is mohammad.r.haghighat at intel.com.

Ravi Narayanaswamy is a Senior Staff Engineer in the Intel Compiler Lab. He is currently working on software transaction memory support in the compiler. His previous role at Intel included porting of the compiler to various platforms. He was also involved in various optimizations in the compiler. He has an M.S. degree in Environmental Engineering and in Computer Science, both from Southern Illinois University, Carbondale. His e-mail is ravi.narayanaswamy at intel.com.

Bhanu Shankar is a Staff Engineer at Intel's Performance, Analysis and Threading Lab. His primary areas of interest include compilers, performance tools for HPC, and multi-threaded architectures. Bhanu received his Ph.D. degree from Colorado State University. His e-mail is bhanu.shankar at intel.com.

Neil Faiman is a Senior Staff Software Engineer in the Intel Compiler Lab, working on the development of tools to help assist users with threading their applications. Neil has been with Intel for five years. He came to Intel from Compaq, where he was the Intermediate Language architect for the GEM compiler project. Neil has B.S. and M.S. degrees from Michigan State University. His e-mail is neil.faiman at intel.com.

David Sehr heads the Advanced Tools team in the Software and Solutions Group at Intel. He was named a Senior Principal Engineer in 2003. At the time this work was done, David was the compiler architect and leader of the advanced development team in the Intel Compiler Lab. David received his Ph.D. degree from the University of Illinois at Urbana-Champaign in 1992, working under the direction of David Padua and Laxmikant Kale. His interests include threading, performance tools, language implementation and compilation, and static analysis. His e-mail is sehr at google.com.

THIS PAGE INTENTIONALLY LEFT BLANK

Future-Proof Data Parallel Algorithms and Software on Intel® Multi-Core Architecture

Anwar Ghuloum, Corporate Technology Group, Intel Corporation
Terry Smith, Corporate Technology Group, Intel Corporation
Gansha Wu, Corporate Technology Group, Intel Corporation
Xin Zhou, Corporate Technology Group, Intel Corporation
Jesse Fang, Corporate Technology Group, Intel Corporation
Peng Guo, Corporate Technology Group, Intel Corporation
Byoungro So, Corporate Technology Group, Intel Corporation
Mohan Rajagopalan, Corporate Technology Group, Intel Corporation
Yongjian Chen, Corporate Technology Group, Intel Corporation
Biao Chen, Corporate Technology Group, Intel Corporation

Index words: parallel programming, data parallelism, forward scalability

ABSTRACT

Developers face new challenges with multi-core software development. The first of these challenges is a significant productivity burden particular to parallel programming. A big contributor to this burden is the relative difficulty of tracking down data races, which manifest non-deterministically. The second challenge is parallelizing applications so that they effectively scale with new core counts and the inevitable enhancement and evolution of the instruction set. This is a new and subtle change to the benefit of backwards compatibility inherent in Intel® Architecture (IA): performance may not scale forward with new micro-architectures and, in some cases, may regress. We assert that *forward-scaling* is an essential requirement for new programming models, tools, and methodologies intended for multi-core software development.

We are implementing a programming model called the *Ct* API that leverages the strengths of data parallel programming to help address these challenges of multi-core software development. In this paper we describe how Ct is designed for minimal effort by the developer, while providing forward scaling on multi-core IA. We describe how Ct's design and implementation evolved from the initial prototype, based on co-traveler feedback, and we provide examples of how Ct can be used. We demonstrate how a sampling of key application spaces can be easily written using Ct to achieve high performance. Finally, we discuss how these ideas can be transitioned into mainstream software development tools.

INTRODUCTION

The data parallel style of programming [3][9][10][15] is best encapsulated in programming models in which collections of data elements are operated on en masse using various operators. For example, if a programmer wishes to sum the elements of two vectors (or matrices, trees, or sets) together, she simply writes an expression that adds these collections free of the bookkeeping and overhead typically associated with threaded programming (i.e., A = B + C).

Lately, data parallelism has (re-)emerged as an important topic in multi-core application development for a number of important technical reasons. First, many algorithms, including much of what is considered "low-hanging fruit," are appropriately characterized as data parallel in nature. Second, data parallel programming models offer the elusive, yet highly desirable, property of determinism, which effectively eliminates data races as a class of programmer errors. Put simply, this means that the programmer writes code that behaves the same way regardless of the number of cores on which it is executed. Third, data parallel programming models are generally highly portable, offering the possibility of building parallel applications that adapt to new micro-architectures.

Another highly prized characteristic of data parallel programming models is a predictable and relatively simple performance model. This allows the programmer to consider performance in software design without focusing on the specifics of the underlying architecture. A related consequence of this characteristic is that data parallel algorithms provide a means to future-proof applications. As previously mentioned, a significant challenge to programming for multi-core architectures is *forward-scaling* performance in applications on evolving multi-core architecture. The performance of parallel applications is very sensitive to core count, vector ISA width (e.g., SSE), core-to-core latencies, memory hierarchy design, and synchronization costs[1]. Software development tools must abstract these variations so that software performance continues to reap the benefits of Moore's law. The built-in performance model of data parallel programming naturally accomplishes this. Figure 1 illustrates how a compiled Ct binary can dynamically be reoptimized for these changing parameters.

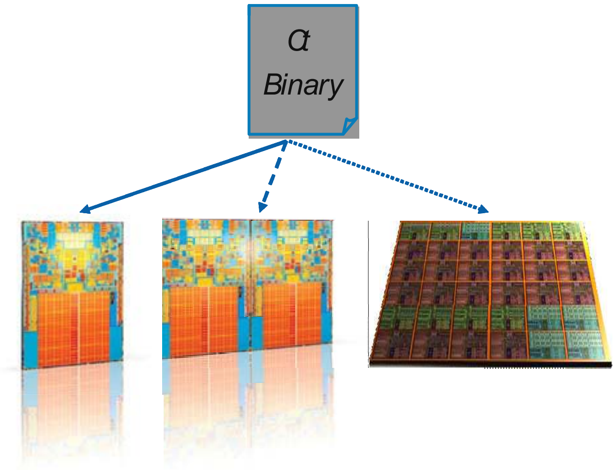


Figure 1: Forward scaling with Ct

An important goal of Ct is to extend the benefits of data parallel programming to less structured task parallel programming. We also aim to address highly object-oriented application designs. Because of this, we have developed a set of technologies to go well beyond basic data parallelism. For example, the underlying model of parallelism used by Ct is a sophisticated implementation of fine-grained concurrency and synchronization that we can progressively expose through the evolving Ct API.

In the next section of our paper we describe key factors and trends driving modern software development and how they are impacted by multi-core programming. Following that, we describe parallel programming models and show where data parallelism lives from a taxonomic point of view. We then describe the Ct API and its implementation in detail and conclude with examples of Ct in action for typical algorithms.

SOFTWARE DEVELOPMENT DRIVERS

Software development takes fundamentally different processes and paths in different market segments. We believe that it is essential to understand these variations to adequately solve multi-core development challenges.

These are the key factors driving the adoption of parallel programming for multi-core architecture:

- *Productivity*: In most market segments, programmer productivity is a major factor in adopting new methodologies for programming, regardless of the benefits. Programmer productivity directly impacts cost and time-to-market, the latter of which is often driven by seasonal milestones. Productivity is adversely impacted by (newly introduced) parallelism-related bugs, performance tuning, and porting to increasingly parallel architectures.
- *Performance*. Raw performance, as measured directly by frame rate in a game or indirectly as new features enabled, is a first-order concern for most ISVs. However, there is frequently (though not always) tradeoffs against productivity-driven metrics like time-to-market.
- *Incremental adoption*. It is probably unreasonable to expect a company with an investment in several hundreds of thousands (or even millions) of lines of code to rewrite this code completely for parallelism. Rather, incremental adoption of parallelism features is the most likely scenario for the typical software developer. This carries with it several interoperability burdens: legacy binary libraries, existing code, and legacy threading APIs. For example, many developers use OpenMP or MPI to parallelize their code. It is paramount that new bridging technologies for parallel computing work well with these components.
- *Object-oriented design methodologies*: These can be viewed as another legacy interoperability issue, but their uniqueness and pervasiveness warrants separate consideration. In the last couple of decades, highly abstracted, objected-oriented programming styles have prevailed in the general software engineering community. The reasons are obvious: increasing abstraction levels facilitate more generic programming methods that increase code reusability. In many instances (C++, for example), programmers have found unexpected ways to use highly abstracted libraries through template meta-programming (Ct

[1] These are not necessarily orthogonal parameters.

itself leverages this!). This trend, however, runs counter to what the compiler and performance optimizer needs to see to generate high-performance parallel code: i.e., well-defined regions (typically, loops) of compute intensive execution. Any mainstream parallelism features must integrate smoothly into these programming methodologies.

This all boils down to the following seemingly untenable requirement: *Developers want a useful high-level programming model that introduces no parallelism-related bugs, yields high performance, and interoperates smoothly within the designs of their existing code base.*

In the next section, we describe how to characterize programming models in a way that serves this requirement.

DATA PARALLELISM BASICS

Parallel programming takes on many flavors. Traditionally, parallel programming models have been compared using dimensions such as message passing versus shared memory, or task (or control) versus data parallelism. However, the portability and expressive power of a particular manifestation of a programming model can transcend these issues. For example, some programming models are amenable to implementation on both shared memory and message passing systems. Also, many algorithms can be equally expressed using either task or data parallelism.

Despite the numerous formal and informal attempts to classify parallel programming models in this vein, we have chosen to measure success by specifically addressing the issues we raised in the previous sections. Our goal is to demonstrate all of these characteristics in our design:

- *Expressive power*. This is the ability to succinctly express different parallel algorithms in a model. For example, task parallel models support data parallel algorithms, though data parallel models cannot easily express some forms of task parallel algorithms. For a given application class, one style of programming model is likely to be prevalent.
- *Determinism*. A deterministic model has no possibility of data races introduced by the programmer, eliminating this new class of bugs. This directly impacts programmer productivity, though tools may mitigate this.
- *Performance transparency*. At the lexical level, it is possible to predict performance to varying degrees of accuracy. This often has a greater impact on programmer productivity, as it requires significant effort and low-level architectural understanding to tune performance on highly parallel architectures.
- *Portability*. Architectural portability is closely related to the requirements for forward-scaling multi-core applications. As the core count is scaled in multi-core architectures and new ISA enhancements are introduced, portable models are necessary to reliably leverage these features.

Expressive Power

Data parallel programming models allow the programmer to specify parallelism implicitly as operators on collections of data. For example, if a programmer wants to add to arrays of data in element-wise fashion, a data parallel programming model would be able to find parallelism roughly proportional to the amount of data in each array. So, if the arrays have 1,000 elements each, this comprises 1,000 independent (and potentially parallel) operations. To perform this computation, the data parallel model's implementation may choose to use parallel threads or tasks and vector instructions at its discretion.[2]

In the early days (the 60s and 70s) of parallel computing, this style of data parallelism was prevalent in languages like APL [3][15] and in the loop-y programming styles of Fortran (where the compiler did the heavy lifting with little guidance from the programmer).

The typical base data type in a data parallel programming model is an array or vector. Sometimes, these can be multi-dimensional. This has been the cornerstone of most models, but it can limit expressiveness. For example, flat or multi-dimensional vector-based models were most readily useful for dense linear algebra and signal or image processing applications. Moreover, complex computation patterns, like recursive subdivision or divide-and-conquer, were severely constrained in these models. Still, a large swath of applications found these models useful.

The key to broadening the applicability of data parallel models is to become more generically "collection-oriented." That is, by adding more types of collections that are supportable, the model becomes more expressive. For example, in the late 80s and early 90s, APL2 [4][14] and Nesl [11][18] added support for *segmented* vectors (see also [17] for a latter day example), which allowed the programmer to represent both irregular data structures and control flow. Per the former, sparse linear algebra was productively programmed using Nesl. Per the latter, divide-and-conquer algorithms like *quicksort* and *quickhull* were easily programmed. Paralation Lisp [19] and CM-Lisp [12][13] added support for indexed vectors,

[2] This computation can also be expressed as a task parallel computation, where we would "spawn" tasks for each of the 1,000 additions, followed by a synchronization to ensure that the computation is completed.

allowing even more complex data structures (including additional sparse representations) to be represented. Ct builds on these algorithms.

There are limitations to the applicability of the data parallel model. For example, applications that require tasks that make asynchronous updates to shared data will generally not map well onto this model. A Web server is a very good example of such an application. It is important to note that most applications require a variety of parallel programming models, so despite the prevalence of data parallelism for these applications, other flavors of parallelism are often required.

Determinism

The data parallel model generally relies on a compiler and/or runtime to manage task creation and usage of vector instruction; there is no explicit thread spawning or synchronization necessary, so data races are non-existent as far as the programmer is concerned. Though the data parallel model can provide fairly sophisticated data movement and communication primitives, it preserves this model.

For example, Ct provides many *collective communication* primitives, including the ability to perform a sum reduction on a vector. This entails summing all elements of the vector in parallel, which requires re-associating the computation. However, the programmer need only specify the reduction operator and leave the necessary threading and synchronization to the runtime. When considering nested or indexed vectors, the semantics of the operator are much more complex, but the programmer’s view is as simple as a flat vector reduction.

Performance Transparency

Though the data parallel model constrains expressiveness somewhat, this property and its high-level abstraction bespeak a relatively predictable performance model. When programming with threads and lower-level synchronization constructs, it is difficult to predict when serialization (intended and unintended) will happen. Moreover, it is extremely difficult to predict memory-related performance issues, since predicting the volume of data accessed and any potential conflicts between threads is often rendered intractable by the high level of abstraction used in modern software.

Operations on collections have the desirable properties that the programmer can predict relative performance behaviors based on collection size and operation complexity. For example, a 1,000 by 1,000 element 2 dimensional matrix generally introduces up to 1,000,000-way parallelism, meaning that for up to thousands of hardware threads, the computation is likely to be able to profitably scale. Furthermore, a collective communication primitive is likely to engender more synchronization than an element-wise operation (which often optimizes away to no synchronization). Though the exact performance is still difficult to predict, these higher-level tradeoffs allow the programmer to make good algorithmic choices.

Portability

Data parallel models have been mapped to a wide range of architectures, from massively parallel distributed memory architectures, to shared memory multi-processors, to deeply pipelined vector supercomputers, to GPUs. This portability is critical to the matching software requirements for evolving multi-core architecture.

This evolution is following several paths. First, the core count will increase, requiring ever increasing amounts of parallelism. Second, non-uniformity of memory access time between cores is increasing, meaning that typical memory access latencies will exhibit high variance to predict unless data partitioning is done carefully. Somewhat related to these two considerations, relative core-to-core synchronization costs will change, requiring re-optimization of code to make the best hide-related latencies. Third, we expect the instruction set improvements to continue, requiring quick adaptation to these enhancements.

The resiliency of data parallel models in many different operating environments is evidence of its ability to adapt to these changes. In particular, the programmer can expect that an algorithm written in a data parallel style will scale across generations of multi-core architectures, using ever-more cores and leveraging newer and wider vector ISAs while avoiding the pitfalls of unintended serialization through the memory hierarchy.

CT

Brief Ct Overview

Ct is a data parallel programming environment with predictable syntax based on C++ that provides distinct semantics and performance [6].

Unique among commercial data parallel programming models, Ct implements a *nested data parallel* model based on work on Nesl [18] and Paralation Lisp [19]. Ct’s nested data parallelism enables a far broader set of collections to be represented. For example, sparse matrices and trees are very difficult to represent in flat data parallel or streaming models. However, these fall out naturally in a nested data parallel model. Also, common divide-and-conquer algorithms, for example, KD-tree construction and sorting, are very difficult to express using flat data parallel and streaming models. These are readily expressed using nested data parallelism. Nested data parallel computations generally do not port efficiently to GPUs and streaming architectures, but they run efficiently on multi-core IA.

Unlike many of its data-parallel brethren, Ct also supports *deterministic task parallelism* on multi-core IA (inspired by [16]). Determinism guarantees that program behavior is identical, on one core or many cores. This essentially eliminates an entire class of programmer errors—data races.

TVECs

The basic type in Ct is a generic vector type, called TVEC. TVECs are allocated and managed in a segregated memory space that is accessible only by Ct operators, to ensure the safety of parallel operation on vectors. TVEC is polymorphic in terms of its base types and shapes.

The base types of TVECs are drawn from a set of typical pre-defined scalar (or value) types. Examples of base types include `I32` (32-bit integer), `I64` (64-bit integer), `F32` (Float), `F64` (Double), and `Bool` (Boolean). In future, Ct will also support the `Bit` type and user-defined base types, for example, C struct-like base type, `TSTRUCT`, and the C array-like base type, `TUPLE`, for more complicated application scenarios.

A TVEC may be declared as follows:

```
TVEC<F32> temp;

TVEC<F32> prices(option_prices, num_options);

TVEC<I8> red(image, length, 4/*stride*/);
```

The TVEC constructor copies data explicitly from the unmanaged C/C++ memory to the managed vector space, in the form of either plain element-wise copy, or the strided memory copy (`red` in the example above takes one byte from every four of the data stream). There are also exceptional cases when it is not preferable to copy the data all at once (because of long latency) or we do not want to copy at all. Thus, there are several TVEC traits that may be applied, including `Stream` for copying data in a streaming fashion, or `Direct` for not copying.

```
TVEC<F32, Stream> stream;

TVEC<F32, Direct> mapped_file;
```

Constant TVECs may also be constructed by factory methods. For example, an identity matrix, in the form of TVEC2D (a TVEC derivative for matrices), may be created as follows:

```
TVEC2D<F32> id = TVEC2D<F32>::identity(dim);
```

Nested data parallelism is a distinguished property for programming irregular data structures and algorithms. TVECs assume a number of shapes, including flat, multi-dimensional, irregular nested, and indexed forms. For example, a matrix TVEC could be constructed as follows:

```
TVEC2D<F32> matrix(data, width, height);
```

TVECs may also be associated with certain accuracy attributes, which may allow experienced programmers to influence the compiler's code generation. For example:

```
TVEC<F32, Default, 2/*ulp*/> data;

... = sqrt(data);
```

The above TVEC declaration specifies 2 ulp (*units-in-the-last-place*) as the tolerable accuracy threshold, which gives a hint to the compiler that the square root operator may be translated into a simpler code sequence with lower-order polynomials and less fix-up code. However, if 0.5 ulp is specified, the compiler may generate a more complicated code sequence that might be up to 60+% slower on some architectures.

When the computation on TVECs is completed, the computed results may be transferred back to the unmanaged space through the `copyOut` primitive.

Ct Operators

The only operators allowed on TVECs are Ct operators. Ct operators are functionally pure (free of side effects). That is, TVECs are passed around by value, and each Ct operator logically returns a new TVEC. For example:

```
scaled_red = red * 0.5; //a new TVEC is born
```

This property guarantees the safety of parallelism and the aggressive optimizations that make parallelism efficient.

The Ct API provides a broad range of Ct operators with rich functionalities. Operator overloading is used extensively to support a programming style, based on C++, particularly for the arithmetic, bitwise, and logical/comparison operators. For example, the '*' operator in the above example is overloaded to the TVEC multiply operator.

Basically Ct operators can be categorized into element-wise/vector-scalar, collective communication, and permutation operators.

Element-wise/vector-scalar operators are typically referred to as "embarrassingly" parallel, requiring no interactions between the computations on each vector element. An example of an element-wise operation is the addition of two vectors:

```
TVEC<F32> A = B + C; //"+" resolves to ctAdd
```

Note that this code generically performs an element-wise addition of two vectors, regardless of the "shape" of the two vectors (i.e., their length, dimensionality, irregularity).

Collective communication operators tend to provide distilled computations over entire vectors and are highly coordinated [3]. While they have a high degree of interference, they can be structured so that there is parallelism in colliding writes, and they typically scale in performance linearly with processor count, with little or no hardware support.

There are two kinds of collective communication primitives in general, namely reductions and prefix-sums (also called scans). Reductions apply an operator over an entire vector to compute a distilled value (or values, depending on the type of vector). Prefix-sums perform a similar operation, but return a partial result for each vector element. For example, an `addReduce` sums over all the elements of a vector if the vector is flat. More concretely, `addReduce([1 0 2 -1 4])` yields `1+0+2+(-1)+4=6`. Likewise, `addScan([1 0 2 -1 4])` yields `[1 1+0 1+0+2 1+0+2+(-1) 1+0+2+(-1)+4]`.

A *permutation* operator in Ct is any operator that requires moving data from its original position to a different position. An example is the `gather` operation, which uses an index array to collect values of a vector in a particular order; and the scatter operator does the reverse. Permutations run the gamut, from arbitrary permutations with arbitrary collisions (occurring when two values want to reside in the same location) to well-structured and predictable permutations where no collisions can occur. For collisions, it is recommended that programmers make use of the collective communication operators. An example of a well-structured (and efficient) permutation operator is `pack`, which uses a flag vector to select values from a vector in the source vector order. With proper hardware support on multi-core IA, these operators can be implemented fairly efficiently. In contrast, these operators could not be implemented efficiently on constrained architectures (for example, most GPUs do not efficiently support scatter operations).

Besides these built-in operators, Ct also supports generic user-defined operators through Ct functions. As implied by their name, Ct functions define a block of code that is applicable to a collection of vectors, which allows programmers to define new generic operators or functions for repeated application (mitigating compilation overhead). The following code defines a Ct function that performs a fused multiply-add:

```
F32 fma(F32 a, F32 b, F32 c)
        return a + b * c;
```

We use a map operator that takes as arguments the Ct function pointer and three vector arguments to apply this function in an element-wise manner:

```
map(fma, ta, tb, tc)
```

The implementation of the map operator employs compile-time type inference to prevent programmers from specifying improper arguments, such as TVEC<I32> (which is not conformant to this function's definition), or wrong numbers of arguments. Just like C/C++ routines, Ct functions are composable, greatly extending Ct's expressiveness.

Nested Vectors

Ct's support for nested vectors is a generalization that allows a greater degree of flexibility than is otherwise found in most data parallel models. TVECs may be flat vectors or regular multi-dimensional vectors. They also may be nested vectors of varying length, which allows for very expressive coding of irregular algorithms, such as other variants of sparse matrix representations, or byproducts of divide-and-conquer algorithms.

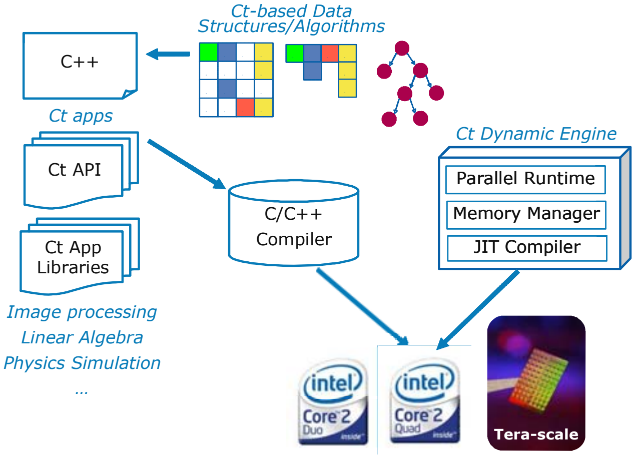


Figure 2: The usage and implementation of Ct

The vector value `[a b c d e f]` is a flat (or 1-dimensional) vector. The vector `[[a b][c d e f]]` holds the same element values, but is a vector of two vectors of lengths 2 and 4. The second vector might represent a partitioning of the first vector's data based on certain criteria (e.g., the relationship to a pivot value in a quicksort). Practically, the nested format enables a lot of irregular data structures and algorithms. Figure 2 gives a few such examples.

All Ct operators work on nested TVECs generically. The behavior of element-wise operators is the same for nested TVECs as for flat vectors. For example, `[[a b][c d e f]] + [[g h][i j k l]]` yields `[[a+g b+h][c+i d+j e+k f+l]]`.

[3] These operators are called collective communication operators in MPI and reductions in OpenMP, though neither provides the rich set of operations that Ct does. In functional languages, these are termed fold operations or list homomorphisms.

The power of nested versus flat TVECs is primarily differentiated through the behavior of collective communication and permutation primitives. Collective communication primitives applied to nested TVECs "respect the boundaries" of the subvectors by applying the operator to each subvector independently. For example, `addReduce([a b c d e f])` yields the singleton vector `[a+b+c+d+e+f]`, while `addReduce([[a b][c d e f])` yields the two-element vector `[a+b c+d+e+f]`.

IMPLEMENTING CT FOR FORWARD-SCALING

Figure 2 also illustrates the Ct execution model. The core of Ct-enabled applications is the use of Ct-based data structures and algorithms. In addition, Ct Application Libraries are a set of well-optimized higher-level APIs aiming to boost programmers' productivity for Tera-scale applications such as image processing, linear algebra, and physics simulation. The Ct libraries can be compiled by stock C++ compilers, such as Visual C++, Intel® C/C++, and Gnu C/C++ compilers, into an IA binary that is able to run on all multi-core IA platforms. This binary comprises of either IA32 or Intel® 64 Architecture instructions, which also include calls to the Ct Dynamic Engine. During the execution of the binary, the Ct Dynamic Engine is launched and provides the services essential to performance and forward-scaling. More specifically, the three major services are the Threading Runtime (TRT), Memory Manager (MM), and Just-In-Time (JIT) compiler. In particular, the TRT and JIT (especially the vector abstraction we will introduce called VIP) provide the basis for forward scaling across IA.

The Threading Runtime

The first key to forward-scaling is to *dynamically adapt to new architectural characteristics*. Threading and synchronization overhead is likely to change between processor generations, necessitating an ability to both select the task granularity and synchronization method dynamically. In fact, our approach is to isolate the architecture-dependent components of the Ct runtime to dynamically loaded libraries. Another aspect of forward-scaling is that data set sizes are likely to scale in the long run, but are generally unpredictable in phases of computation, especially for client applications such as games. In this case, the amount of data being processed is highly scene and gameplay dependent. As such, the runtime must be able to adapt its threading strategy to variable data sets.

The TRT provides a fine-grained threading model that is used to implement both data parallel and task parallel constructs. The underlying building block for this model is a *future*, which under the runtime semantics may represent a suspended closure or *thunk* (i.e., a function pointer and an argument list representing a potentially parallel function application), a thunk computation *in flight,* or a *computed* value (representing a successfully evaluated thunk). A handle to a *future* essentially represents a dependency on that suspended thunk's evaluation. This is inspired by the techniques for expressing and managing parallelism presented in [7][8]. Using this mechanism, many complex fine-grained synchronization patterns may be expressed; however, the TRT facilitates fine-grained synchronizations via a building block called a *join*. A join can be used to express a range of logical combinations of synchronization dependencies.

The TRT uses additional primitives called *bulkspawns* and *bulkjoins*, which essentially represent mapped future spawns and joins on collection-oriented arguments. Bulkspawn operations dynamically partition the collection into the *right number* of fine-grained tasks interlinked with fine-grained synchronizations. This is key to adapting to the core count and utilization, as well as cache footprint.

The Memory Manager

The Ct MM automatically manages the segregated Ct vector space. As such, it provides a set of lock-free memory allocation interfaces, as well as a reference-counting-based garbage collector to reclaim dead vectors automatically. The MM is responsible for allocated data format and, in conjunction with the TRT, partitions vectors for parallel operations (i.e., the TRT bulkspawn operations).

The Compiler

The Ct compiler has a slightly unconventional structure, notably in its dynamic nature. When executing Ct API calls, the dynamic engine constructs intermediate representations of the computation, deferring actual execution (and further optimization) until *later*. "Later" is bounded by the necessity to copy values back into native C/C++ space, though the engine may decide to compile code at intermediate steps, such as when back edges in control flow (i.e., loops) are detected. This intermediate representation (IR) building is the default mode of Ct code execution for new paths in the program. Otherwise, cached code is executed if the path followed is in the "Code Cache," or a cached IR is augmented.

Once the compiler is invoked, several phases with distinct objectives are invoked: the High-Level Optimizer (HLO), the Low-Level Optimizer (LLO), and the VIP Code Generator (VCG).

The HLO phase [5] performs architecture- and runtime-independent optimizations, such as sub-primitive decomposition (breaking up data parallel operators into more primitive patterns of parallelism), fusion (essential to coarsening the fine-grained concurrency of the Ct model as much as possible), scalarization, common sub-expression elimination, and copy propagation. These optimizations are all possible without introducing the details of run-time memory allocation and shape checks. This is left to LLO.

The LLO phase is still architecture independent, but unlike HLO, LLO does runtime-dependent optimizations. It has three primary objectives: 1) generate parallelized kernels using the TRT; 2) translate the optimized kernels (spawned in the TRT) into proper vectorized code; 3) generate architecture-independent representations, which we call the Virtual Intel® Platform, or VIP. Much of the difficulty of implementing and optimizing collective communication [1] and segmented operators [2] is deferred to this phase. VIP is an abstract instruction set that is based on IA32/Intel 64 Architecture, but that uses a generalized vector ISA to defer binding to a particular generation of SSE as late as possible.

One of the challenges we face for forward-scaling is the *near certainty of SSE extensions and enhancements*. Figure 3 shows the evolution of Intel SIMD ISA. The number of instructions has been increasing at the pace of 30 instructions per year on average since the introduction of MMX™ technology in 1996. Meanwhile, the data width of vector registers was also increased from 64 bits to 128 bits, and it can be reasonably expected to increase further at some point in the future. This is driven by the need to increase performance in the most power efficient way and extending SIMD ISA is one such mechanism. VIP, as a virtual ISA of the Ct Dynamic Engine, is designed to hide changes in SIMD ISA, and, via VCG, provide future-proof performance to Ct applications.

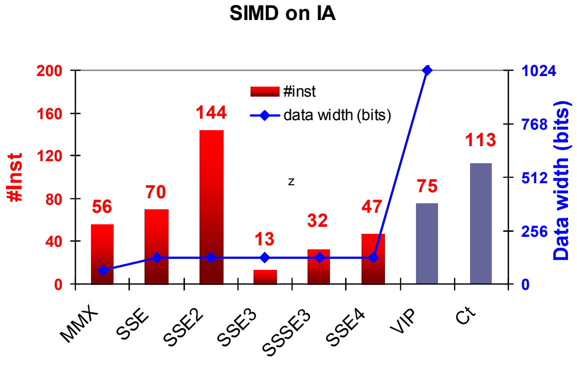


Figure 3: The evolution of IA SIMD ISA vs. Ct

The VCG phase is a state-of-the-art backend for VIP that dynamically selects the appropriate target ISA. When new SSE extensions are introduced, a new dynamically linked library can be made available that supports both legacy and new SSE extensions. No recompilation with a static compiler is necessary. In this way, applications can forward scale through vector ISA with the adaptivity of the VCG backend. VCG does classic loop-based optimizations, such as loop fusion, loop interchange, and array contraction. VCG also does architecture-dependent optimizations, such as register allocation and instruction scheduling.

By carefully layering architecture and run-time dependent optimizations in the Ct compiler, we can retarget the entire dynamic engine with great agility, including for the purposes of evaluating new micro-architectures (i.e., considering in-order architectures and evaluating new, throughput-oriented ISA extensions). This was done deliberately with forward-scaling in mind.

The dynamic compilation approach, especially the VIP layer, provides smooth migration paths to future SSE and IA-based SIMD instruction sets.

CT IN ACTION

In this section, we walk through some examples to demonstrate how Ct boosts the productivity and performance for a variety of application domains. We take a step-by-step approach, to make clear some guidelines for porting to Ct. In particular, we provide rules of thumb for translating sequential code to Ct code.

Black-Scholes

Option pricing is a computation-hungry, important application in modern financial engineering. Black-Scholes is a well-accepted analytical model for European option pricing. We use it here as an exemplar for C/C++ to Ct migration. The code below shows the sequential C code.

```
1  float s[N], x[N], r[N], v[N], t[N];
2  float result[N];

3  for(int i = 0; i < N; i++) {
4   float d1 = s[i] / ln(x[i]);
5   d1 += (r[i] + v[i] * v[i] * 0.5f) * t[i];
6   d1 /= sqrt(t[i]);
7   float d2 = d1 – sqrt(t[i]);

8   result[i] = x[i] * exp(r[i] * t[i]) *
9    ( 1.0f - CND(d2)) + (-s[i]) * (1.0f –
                                   CND(d1));
10 }
```

The code below shows its Ct counterpart. The two pieces of code are very similar (lines 4-9).

```
0  #include <ct.h>

1  T s[N], x[N], r[N], v[N], t[N];
2  T result[N];
3  TVEC<T> S(s, N), X(x, N), R(r, N), V(v, N),
T(t, N);
```

```
4  TVEC<T> d1 = S / ln(X);
5  d1 += (R + V * V * 0.5f) * T;
6  d1 /= sqrt(T);
7  TVEC<T> d2 = d1 - sqrt(T);

8  TVEC<T> result = X * exp(R * T) *
9   ( 1.0f - CND(d2)) + (-S) * (1.0f -
                                    CND(d1));
```

The only differences are these:

- Ct needs to include the `ct.h` header file (line 0).
- Ct adds the TVEC declarations (line 3).
- Ct exempts programmers from having to manipulate arrays with loops and subscripts (lines 3-9).
- The Ct version co-exists well with C++'s parametric polymorphism, allowing the code to be instantiated with different types `T`.

The desirable tradeoff here is that the coding overhead is small when migrating to Ct, but you get highly efficient vectorized, parallelized, and forward-scaling code. In contrast, a manually vectorized version using MMX/SSE intrinsics has 51 lines of code, and the manual parallelization using threads requires an additional 20+ lines of code. When the underlying hardware or OS changes, you may need to modify the code to use new instrinsics and change the number of threads or the threading primitives (e.g., pthreads).

The code below shows how a Cumulative Normal Distribution function, CND, is implemented in C and Ct, respectively. Though the C code can be translated into Ct easily, we use a Ct Application Library function, `Ct::Polynomial::eval`, to accelerate the polynomial evaluation. Our data show that for a 5th-order polynomial, the optimized Ct library yields ~3X speedup over the naïve implementation with negligible precision loss.

```
float CND(float d) { // The C version
  ... ...
  w = 0.31938153f * k -
    0.356563782f * k * k +
    1.781477937f * k * k * k -
    1.821255978f * k * k * k * k +
    1.330274429f * k * k * k * k * k;
  ... ...
}


template <typename T>
TVEC<T> CND(TVEC<T> d) { // The Ct version
 ... ...
 static T coefficients[] = {
        0.0f,
      0.31938153f,
      0.356563782f,
      1.781477937f,
      1.821255978f,
      1.330274429f};

 w = Ct::Polynomial::eval(k, 5/*order*/,
coefficients);
 ... ...
}
```

Rule of Thumb I:

```
for(int i = 0; i < N; i++)
 ... data[i] ...
➔
 ... data ...
```

Convolution

Convolution is a widely used function in many application domains ranging from signal/image processing to statistics, to geophysics. Compared with the very parallel Black-Scholes, the computation pattern of Convolution is slightly trickier. In particular, programmers have several mechanisms at their disposal and we will explore the tradeoffs between these approaches.

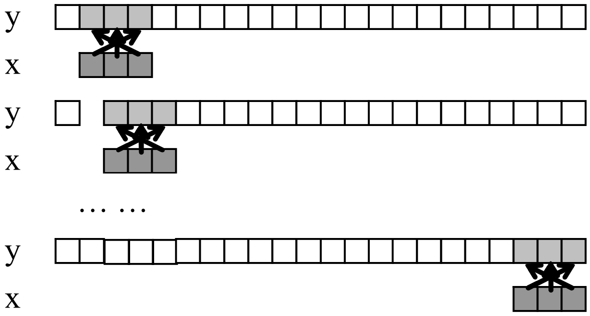


Figure 4: Convolution algorithm illustration (1 dimensional)

Figure 4 is a typical 1D convolution algorithm, where `y` is a data set, and `x` is a kernel sliding through the data set. The code below gives the C implementation [4]. As compared to Figure 4, you may find this loop structure is more complicated and the array access pattern is more irregular (particularly `y[i - j]` on Line 5).

```
float x[M], y[N], z[N];
for (int i = 0; i < N; i++) {
        z[i] = 0.0f;
        for (int j = 0; j < M; j++)
                z[i] += x[j] * y[i-j];
        }
}
```

Our first question is how to map the two-level loops to TVECs. Obviously we want to abstract the data set, `y`, to be a TVEC. In this regard, we peel the outer loop and change all the occurrences of `y` to `Y`, as shown below.

```
T x[M], y[N], z[N];
TVEC<T> Y(y, N), I = TVEC<T>::index(0, N);
TVEC<T> Z = TVEC<T>::constant(0.0f, N) ;
for (int j = 0 ; j < M; j++)
        Z += x[j] * Y[I-j];
```

[4] The example is just for illustration purposes, and we omitted some code for checking boundary conditions.

The second question is how to represent `y[i - j]` in Ct. Because `i` is a loop induction variable incremented by a step value 1, we map it to an identity vector, `I`, which results in `Y[I - j]`. Ct's C++ front-end reinterprets the `[]` operator into a gather operator, gathering values from `Y` according to an index vector, `I - j`.

This porting is straightforward but we can't claim this solution is ideal for performance, because the gather operator is expensive on most architectures. Experienced Ct programmers, when understanding the algorithm, may resort to a more lightweight operator, `shiftPermute`. If you look at Figure 4 from a different perspective, that is, the kernel `x` is fixed while the data set `y` is sliding leftward, the result is equivalent. The optimized implementation is shown in the code below.

```
T x[M], y[N], z[N];
TVEC<T> Y(y, N) ;
TVEC<T> Z = TVEC<T> ::constant(0.0f, N) ;
for (int j = 0 ; j < M ; j++)
        Z += x[i]*Y.shiftPermute(1);
```

It is worthwhile to mention that the Ct implementation can be extended to 2D convolutions with minimal effort.

Rule of Thumb II:

```
  for(int i = 0; i < M; i++)
   for(int j = 0; j < N; j++)
     ... data[i + a, j + b] ...
  ➔
    ... data.shiftPermute(a, b) ...
```

Sparse Matrix Vector Product (SMVP)

Linear algebra, particularly matrix operations, is quite common in high-performance computing, physics simulation, aspects of machine learning, and many Recognition, Mining, and Synthesis (RMS) applications. Sparse matrices are extremely useful for cases where the particular algebraic formulation of a problem sparsely populates elements in the matrix with meaningful values.

An example is large-scale physics simulations. In such cases, the logical size of a dense matrix might be 100s of megabytes, where a sparse matrix representation that only stores non-zero matrix elements would perhaps only hold 1 megabyte of data. Unlike dense matrices, whose control paths and data access patterns are highly predictable, sparse matrices are much more hard in terms of the diversity of data structures and the irregularity of algorithms. In this section, we use a common kernel in gaming and RMS applications, Sparse Matrix Vector Product (SMVP), to demonstrate how a sparse matrix multiplied by a vector is implemented with Ct.

We use a Compressed Sparse Column (CSC) format. The basic idea of CSC is to only store the non-zero elements of the matrix in the column order, and with each non-zero element, the programmer also stores the row index. Consider the sparse matrix below.

```
A =     [[0 1 0 0 0]
         [2 0 3 4 0]
         [0 5 0 0 6]
         [0 7 0 0 8]
         [0 0 9 0 0]]
```

The matrix is stored as three arrays:

- `nz_mval`: the nonzero values, in column major order.
- `row_idx`: the row indices for nonzero values.
- `col_ptr`: the column pointers. `col_ptr[i]` tell the values of the i-th column start from which index into the `nz_mval` array).

```
mval     = [2 1 5 7 3 9 4 6 8]
row_idx = [1 0 2 3 1 4 1 2 3]
col_ptr = [0 1 4 6 8 9]
vval    = [1 2 3 4 5] // the vector values
```

The schema for computing the SMVP is shown below.

```
for (c = 0; c < col_num; c++) {
 for (e = col_ptr[c]; e < col_ptr[c + 1];
                                    e++){
  int r = row_idx[e];
  product[r] += mval[e] /* A[r][c] */
        * vval[c];
 }
}
```

It is worth observing some of the computing patterns to comprehend the implications for porting to Ct:

- The two-level loops are more irregular than the aforementioned examples. Typically this kind of loop structure can be mapped to a two-level nested vector, as shown below

```
[
    [...],//col_ptr[1]-col_ptr[0] elems
    [...],//col_ptr[2]-col_ptr[1] elems
    ...
    [...],//col_ptr[col_num+1]-
            col_ptr[col_num] elems
]
```

- In the inner loop, `vval[c]` is used for `col_ptr[c+1] - col_ptr[c]` times, which can be viewed as a special kind of *gather* operation. In Ct, we have a dedicated operator, `distribute`, to replicate values of a vector for certain numbers of times specified by another vector.
- The expression `product[row_idx[e]]+= ...` implies that we are performing what is called a *combining-send*, or alternatively a *multi-reduction* or *combining-scatter*.

By comprehending the patterns, we have the Ct implementation presented below:

```
vval = vval.distribute(col_ptr);
  //=> [1 2 2 2 3 3 4 5 5]
TVEC<T> product = mval * vval;
  //=> [2 2 10 14 9 27 16 30 40]
product.applyNesting(row_idx,
                            ctNestedIndex);
  //=> [[2] [2 9 16] [10 30] [14 40] [27]]
product = addReduce(product);
  //=> [2 27 40 54 27]
```

Rule of Thumb III:

```
for(int i = 0; i < num_segs; i++)
 for(int j = seg[i]; j < seg[i+1]; j++)
   ... data[j] ...
➔
  data: [ [seg1], [seg2], ..., [segnum_segs] ]
```

Rule of Thumb IV:

```
for(int i = 0; i < M; i++)
  data[j] += ...
➔
addReduce(data);
```

Experimental Results

We measured the performance of Ct and sequential C implementations of representative data parallel operators and a set of real-world applications on an Intel® Xeon® processor E5345 [5] platform (two 2.33GHz quad-core processors, 4GB memory), and plotted the speedup of Ct over C in Figure 5 and Figure 6. To present the benefits from the JIT compiler optimizations and the TRT separately, we configured Ct to run with different numbers of cores. The C implementations are compiled with Visual C++ 2005 compiler.

Figure 5 compares Ct's performance vs. C (compiled with O3-level optimizations on the Intel C Compiler) for a few key operators. These are common building blocks in many-core applications, though their use and mix is varied. It provides a reasonable baseline for assessing scalability based on the mix of Ct operators in your application. These operators can be categorized into two classes: `add`, `max`, `sqrt`, and `exp` belong to element-wise operators, while `addReduce` and `addScan` are collective communication operators.

The Ct implementations of almost all element-wise operators achieve 7-8X scalability when adding the number of cores from 1 to 8. When considering only one core, the Ct Compiler's aggressive vectorization also makes significant difference:

- For `add`, Ct generated code achieves 2X speedup against a scalar implementation. Given SSE's vector width is 4, and the vectorized code is mixed with a lot of scalar code, the speedup is quite reasonable.
- For `max`, the Ct implementation takes advantage of SSE's `max` instructions, however, the C version uses control flow (e.g., `a > b ? a : b`) that are more challenging to vectorize. Thus the speedup reaches up to 11X.
- For `sqrt`, Ct generated code leverages SSE's `sqrt` instruction, while the C code relies on slow C runtime library implementation. As such, the Ct implementation achieves a significant speedup of 42X.
- SSE doesn't have direct support for `exp`. However, Ct still generates highly efficient SSE code sequence, based on a look-up table and interpolation-based method that outperforms the C runtime library-based implementation by 15X.

Unlike element-wise operators that are embarrassingly parallel, collective communication operators have more complicated inter-thread communication patterns. In the meantime, the collective operators also impose challenges on local code vectorization.

- The `addReduce` operator performs the summation of all elements of a vector. The Ct implementation achieves totally 93X speedup over the scalar implementation, where 12X comes from vectorization.
- The `addScan` operator requires more complicated communication patterns. Again, the speedup achieved by our optimized code is as high as 31X, where 5X is from code vectorization.

In the future, Ct's adaptive compilation strategy will play an even more important role when new, throughput-oriented ISA extensions emerge (such as mask, cast/conversion, swizzle/shuffle, and gather/scatter).

[5] Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

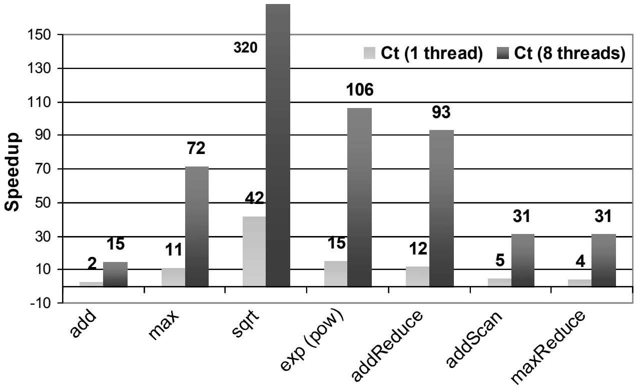


Figure 5: Performance of select Ct primitives vs. C compiled with optimizations

The applications being studied are listed in the table below. These span applications from high-performance computing, financial computing, image processing, through physics simulation. Black-Scholes, Binomial Tree, and Monte Carlo Simulation are three widely used option pricing models, 2D Convolution is a typical signal processing kernel, and the Narrow-Phase Collision Detection is a compute-intensive component of gaming physics.

Table 1: Applications characterized under Ct

Program	Description
Black Scholes	European Option Pricing (financial analytics)
Binomial Tree	American Option Pricing (financial analytics)
Monte Carlo	Asian Option Pricing (financial analytics)
Convolution	2D Convolution kernel (signal processing)
Collision Detection	Narrow-phase collision detection (game physics)

Figure 6 shows Ct's performance on Core 2 Quad machines, as compared to plain C code and hand-compiler tuned SSE code (using SSE intrinsics). The figure also indicates that Ct code has good scalability when increasing the number of cores from 1 to 8.

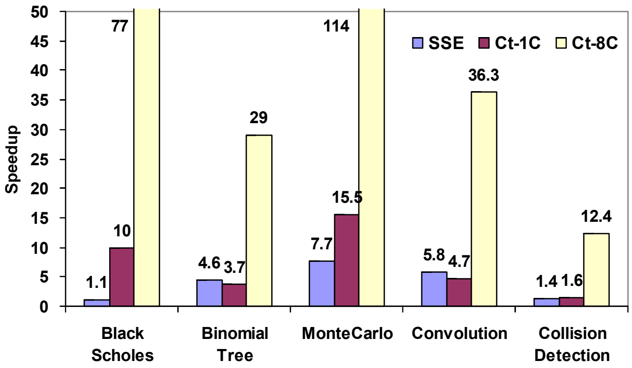


Figure 6: Application scaling with Ct

For single-thread performance, Ct achieves a speedup as high as 10X for Black-Scholes. Black-Scholes relies heavily on the performance of transcendental functions, namely `exp`, `log`, `rcp` and `sqrt`. SSE does provide efficient support for `rcp` and `sqrt`, while lacking support for `exp` and `log`. Typically, programmers fall back to a scalar loop and call corresponding C runtime functions for each element. Even though the rest of the program is well vectorized, the overall speedup of SSE over C is only 1.1X. Ct's 10X speedup is mainly attributed to JIT's use of vectorized implementation of such transcendental functions.

Monte Carlo Simulation has a 15.5X Ct-over-C speedup. Two factors contribute to the 8X speedup: first, Ct has a very fast, vectorized implementation of random number generator, while C has to resort to the C runtime function, `rand`; second, Monte Carlo Simulation heavily uses two transcendental functions, `sin` and `cos`, where Ct also has very efficient SSE-based implementations. Consequently, the speedup achieved by SSE is only 7.7X.

Single threaded Ct for Binomial Tree and Convolution achieve only 3.7X and 4.7X speedup, respectively, which is not surprising given that the two applications are not arithmetic intensive. Their SSE versions are slightly faster because the Visual C++ compiler uses a static compilation strategy that makes more aggressive optimizations affordable. An interesting note is that Binomial Tree suffers from many floating point underflow exceptions. Ct allows specifying lower numerical precision requirements. This enables the Ct Compiler to generate code under "flush-to-zero" mode, which speeds up the performance further by 3-4X. For cases when lower accuracy is not tolerable, we may specify `F64` (namely `double`) as the base type of `TVEC`. Although the SIMD data width is reduced to half, the underflow exceptions are totally removed, which speeds up the performance of the `TVEC<F32>` version by 2.5X. It is trivial for Ct programmers to get the speedup because only TVEC declarations are changed (i.e., they do not have to change a single line of code).

Note that this graph shows the performance when running the *same* Ct binary with different hardware configurations. Looking forward, Ct provides nice forward scalability. Note that relatively inexperienced C/C++ programmers can get these performance benefits (nearly) for free on stock machines. Porting C implementations to their Ct counterparts takes minor effort, and the Lines-of-Code of the two implementations are almost 1:1.

CONCLUSION

Future-proofing algorithms for multi-core architectures is an important way to continue to reap the performance benefits of Moore's Law scaling. Data parallel programming models offer a promising abstraction to use for forward-scaling, but it is often limited by too-narrowly defined types and operators. This severely limits the scope of applicability for such models. In Ct, we are attempting to build a system that delivers a more general data parallel (indeed, a deterministic task parallel) model while providing the essential framework for forward-scaling.

A serious challenge is acknowledging the realities of modern software development methods and assuring compatibility with legacy code and programming methodologies. We believe that using the Ct Dynamic Engine's particular flavor of adaptive compilation and run-time is the most effective way to extract chains of performance code without seriously compromising the language design or the large software investment by developers. More radical language redesigns are likely to appear at some point, but we view this more incremental approach as a flexible and highly productive way to leverage multi-core architectures while developing the basic parallel algorithms and design methods. In fact, we expect that future languages will almost certainly encompass some form (if not the exact form) of the ideas in Ct.

ACKNOWLEDGEMENTS

We thank other members in the Ct team, namely Zhanglin Liu, Dan Zhang, Jane Liu, Zhaohui Du, Josh Fryman, and Zhigang Wang for their **significant and indispensable** contributions to the work presented in this paper. We also wish to thank Joe Schutz, Shekhar Borkar, Jerry Bautista, and Justin Rattner for their support in this work.

REFERENCES

[1] Allan L. Fisher , Anwar M. Ghuloum, "Parallelizing complex scans and reductions" in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 135–146, June 20-24, 1994, Orlando, Florida, United States.

[2] Anwar M. Ghuloum, Allan L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *ACM SIGPLAN Notices*, v.30 n.8, pp. 58–67, August 1995.

[3] "APL: A Programming Language," at http://www.users.cloud9.net/~bradmcc/APL.html

[4] "APL2," at http://www.ibm.com/software/awdtools/apl/

[5] Byoungro So, Anwar Ghuloum, Youfeng Wu, "Optimizing data parallel operations on many-core platforms," *First Workshop on Software Tools for Multi-Core Systems* (STMCS), Manhattan, NY, 2006, pp. 66–70.

[6] "Ct: A Flexible Parallel Pprogramming Model for Tera-scale Architectures," at http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf

[7] Daniel P. Friedman and David S. Wise, "Aspects of applicative programming for parallel processing," *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.

[8] David A. Krantz, Robert H. Halstead, Jr., and Eric Mohr, "Mul-T: a high-performance parallel lisp," in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 81–90, 1989.

[9] Guy Blelloch, "Vector Models for Data-Parallel Computing," *MIT Press*. ISBN 0-262-02313-X. 1990.

[10] Guy E. Blelloch and Gary W. Sabot, "Compiling Collection-oriented Languages onto Massively Parallel Computers," *Journal of Parallel and Distributed Computing, Vol. 8, Issue 2*, pp. 119–134, 1990.

[11] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Journal of Parallel and Distributed Computing (JPDC)*, 21(1), April 1994.

[12] Guy L. Steele and W. Daniel Hillis, "Connection Machine LISP: Fine-grained Parallel Symbolic Processing," in *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, MA, August 1986.

[13] Guy L. Steele, "CM-Lisp Technical Report," *Thinking Machines Corporation*, 1986.

[14] "IBM, APL2 Programming: Language Reference," first ed., August 1984.

[15] Kenneth E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.

[16] Leslie G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 33(8):103–111, August 1990.

[17] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky and Wolf Pfannenstiel, "Nepal – nested data parallelism in Haskell," in *Proceedings 7th International Euro-Par Conference*, volume 2150 of Lecture Notes in *Computer Science*, pp. 524–534, Springer-Verlag, Manchester, UK, 2001.

[18] "NESL: A Parallel Programming Language," at http://www.cs.cmu.edu/~scandal/nesl.html

[19] "Paralation LISP–Embeds the paralation model in Common LISP," Available from *MIT Press*, (800)356-0343.

AUTHORS' BIOGRAPHIES

Anwar Ghuloum is a Principal Engineer with Intel's Microprocessor Technology Lab, working on diverse topics such as parallel language and compiler design, parallel architecture evaluation, optimizing memory system performance, and multimedia applications. Anwar received a B.S. degree in Computer Science and Engineering from the University of California, Los Angeles and a Ph.D. degree in Computer Science from Carnegie Mellon University's School of Computer Science in 1996. Before joining Intel, he co-founded and was the CTO of a fab-less semiconductor startup that designed parallel image and video processors for the consumer electronics market. Prior to that, Anwar developed novel predictive drug design software for early lead optimization using 3D surface pattern recognition techniques for a biotech startup. A recurring theme in Anwar's work has been to bridge high-level application knowledge and low-level parallel architecture constraints with careful parallel language and compiler design to achieve the optimal tradeoffs in productivity and performance. His e-mail is anwar.ghuloum at intel.com.

Terry Smith is a Business Development Manager within Intel's Corporate Technology Group. In his ten years with Intel he has focused on the management of emerging technologies, strategic marketing, and business development. His background includes the Executive MBA Program at the University of Texas-Austin and a B.S. degree in Math/CS from the University of Illinois.

Gansha Wu is a researcher with Intel's Corporate Technology Group. He leads a team researching advanced compiler and runtime technology for future Intel architectures. Gansha has been with Intel for seven years. His e-mail address is gansha.wu at intel.com.

Xin Zhou is a researcher with Intel's Corporate Technology Group. He leads a Ct programmability and workload study. Xin has been with Intel for five years. His e-mail address is xin.zhou at intel.com.

Jesse Fang is the Director and Chief Scientist of the Programming System Lab at Intel/CTG/MTL (Corp. Technology Group/Microprocessor Technology Lab). Before joining Intel in 1995, Jesse was manager of the Hewlett-Packet Research Lab compiler team for Itanium® Architecture. Before that, he was the manager of parallel/vector compilers at Convex and Concurrent Computer Corporation in 1989 and 1986, respectively. Jesse received his Ph.D. degree in Computer Science from the University of Nebraska-Lincoln before he did a post-Doctorate at the University of Illinois, Urbana-Champaign. Jesse received his B.S. degree in Math from Fudan University in Shanghai.

Peng Guo is an engineer in Intel's Corporate Technology Group and works on dynamic compilers. His research interests include dynamic compiler optimizations, and compiler/runtime interactions. He received his Masters degree in Computer Science from the Beijing University of Aeronautics and Astronautics. His e-mail address is peng.guo at intel.com.

Byoungro So is a Senior Research Scientist in Intel's Corporate Technology Group. His research interests include program analysis, high-performance computing, adaptive computing, parallelizing compilers, and performance optimizations. Before joining Intel, he worked for IBM T.J. Watson research center where he developed the Cell compiler and runtime. He received both his M.S. and Ph.D. degrees in Computer Science from the University of Southern California in 1998 and 2003, respectively. His email address is byoungro.so at intel.com.

Mohan Rajagopalan is a Research Scientist in Intel's Programming Systems Lab and leads the parallel runtime research for Ct. His current interests include runtime technologies for forward-scaling on emerging multi-core platforms, new programming models such as for reusable and incremental computation, and whole system optimization. Mohan received his Ph.D. degree from the University of Arizona in 2006. He was the recipient of the 2005 IEEE/IFIP Willam C. Carter Dissertation Award. His e-mail is mohan.rajagopalan at intel.com.

Yongjian Chen is an Engineer in Intel's Corporate Technology Group and works on dynamic compilers. His research interests include parallel language design, compiler/runtime technology to support parallel computation, and parallel architectures. He received his Ph.D. degree from Tsinghua University. His e-mail is yongjian.chen at intel.com.

Biao Chen is an Engineer in Intel's Corporate Technology Group and works on Ct memory management and Ct workload study. His research interests include emerging workloads and memory management. He received his Masters degree in Computer Science from the Beijing University of Aeronautics and Astronautics. His e-mail is biao.chen at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

Accelerating Video Feature Extractions in CBVIR on Multi-core Systems

Yurong Chen, Corporation Technology Group, Intel Corporation
Eric Li, Corporation Technology Group, Intel Corporation
Jianguo Li, Corporation Technology Group, Intel Corporation
Yimin Zhang, Corporation Technology Group, Intel Corporation

Index words: contend-based video information retrieval, multi-core, optimization, parallel computing, performance analysis

ABSTRACT

With the explosive increase in video data, automatic video management (search/retrieval) is becoming a mass market application, and Content-Based Video Information Retrieval (CBVIR) is one of the best solutions. Most CBVIR systems are based on low-level feature extractions guided by the MPEG-7 standard for high-level semantic concept indexing. It is well known that CBVIR is a very compute-intensive task, and the low-level visual feature extractions are the most time-consuming components in CBVIR. Nowadays, with the multi-core processor becoming mainstream, CBVIR can be accelerated by fully utilizing the computing power of available multi-core processors.

In this paper, we optimize and parallelize a set of typical visual feature extraction applications in CBVIR. The underlying optimization and parallel techniques are representative of those used in video-analysis applications and can be further used in other applications to maximally improve their performance on multi-core systems. We conduct a detailed performance analysis of these parallel applications on a dual-socket, quad-core system. The analysis helps us identify possible causes of bottlenecks, and we suggest avenues for scalability improvement to make those applications more powerful in real-time performance.

INTRODUCTION

Nowadays, with advances in video capture and storage techniques, the sheer amount of video data has exploded not only in enterprises but also in our homes. Concomitantly, there is an increasing demand for a system that can help end users to index massive amounts of video data for further search, browse, and management tasks. Digital home-usage media centers are coming into being for this very purpose. Most of these centers consist of two key ingredients: the Content-Based Video Information Retrieval (CBVIR) module and the computing platform.

CBVIR is a computational technique to index unstructured video information in terms of low-level audio/visual features [1]. MPEG-7 is an experimental standard acting as a guideline for low-level audio/visual feature extractions [2]. It includes a set of visual color, texture, shape, and motion descriptors. Since low-level visual feature extraction is the most time-consuming part in CBVIR applications, these applications are much more compute intensive than traditional video decoding/encoding applications. Although typically the indexing can be done in off-line mode, there are many more emerging scenarios that require a real-time or even super-real-time processing capability in a CBVIR system. With the boom in multi-core processors, we can take full advantage of the computing power of today's multi-core platform to accelerate the use of CBVIR applications [3].

In this paper, we optimize and parallelize a set of typical feature extraction applications on a multi-core system. Our results show most of them are much slower than real-time in their original implementations. After serial optimization, however, they become 3.3x faster, and only five of them are still slower than real-time. After the tailored parallelization, the six most compute-intensive applications obtain up to a 7.6x speedup on a dual-socket, quad-core system, which enables them to achieve super-real-time performance.

This paper is organized as follows. First, we briefly review several low-level visual descriptors under the guidelines of the MPEG-7 experimental standard. Next, we present our optimization and parallelization methodology for low-level visual feature extractions. Then, we show the performance analysis results of the typical feature extraction workloads.

CBVIR AND LOW-LEVEL VISUAL DESCRIPTORS

Video information differs from conventional text or numerical data in that video data require a large amount of memory and special processing operations. Video retrieval is based on how the contents of a sequence of images can be represented. Computational techniques that pursue the goals of indexing the unstructured visual information are called CBVIR [1, 4]. Generally, a typical CBVIR system includes two ingredients: the back-end for video indexing and the front-end for retrieval query processing. The back-end extracts low-level audio/visual features for video data indexing, while the front-end is a query engine that returns retrieval results based on the similarity between query example and indexed video data [4]. A typical system framework is illustrated in Figure 1.

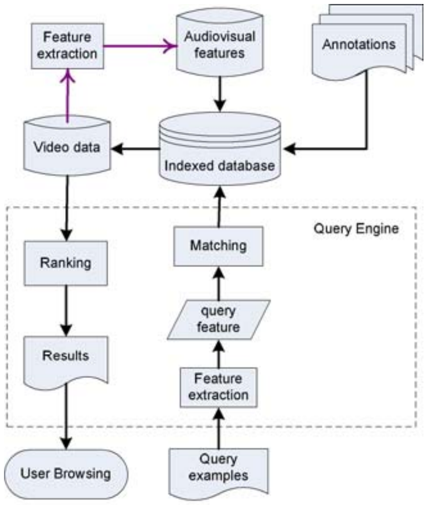


Figure 1: Framework of a typical CBVIR system

The well-known maxim “Garbage in, garbage out” means that good features will greatly improve the retrieval performance of a CBVIR system. Based on this point, the MPEG-7 standard, formally known as the “Multimedia Content Description Interface,” is proposed to guide content retrieval and feature extraction from video data. It includes a set of low-level color descriptors, texture descriptors, shape descriptors, and motion descriptors [2]. Since MPEG-7 is an experimental standard currently, the descriptors are only at the conceptual level. Therefore, in practice, most CBVIR systems just use MPEG-7 as a guideline for low-level feature extractions [1, 5]. In our experiments, we also use MPEG-7 as a guideline, and we briefly introduce the most-widely used visual features. In each category, we also select one or two typical features with detailed descriptions. These features are widely adopted and have very good retrieval performance [6].

Color Descriptors

Because of its expressive power, color is one of the first attributes used in image description, similarity, and retrieval tasks [7]. MPEG-7 divides color descriptors into several sub-categories: scalable color, color structure, color layout and so on [2]. In practice, there are four widely used color descriptors: Color Histogram, Color Moments, Color Coherence Vector (CCV), and Color Correlogram. The first two can be viewed as scale color descriptors, and the latter two can be viewed as structure color descriptors. In color histograms, overall color distribution can be captured in terms of histogram or low-order moments, but color histograms do not capture any spatial relationships among colors. The CCV is an extension of color histograms, in that it partitions pixels falling in each color histogram bin into coherent pixels and non-coherent pixels.

Color Correlogram is proposed to characterize how the spatial correlation of pairs of colors is changing with the distance [8]. It provides much better performance than color histograms, color moments, and the CCV [6, 8] and has been widely used in CBVIR systems [1, 5]. Color Correlogram extends the co-occurrence matrix method in texture analysis to the color domain. In short, a correlogram is a squared table where the entry at (i, j) specifies the probability of finding a pixel of color c_j at a fixed distance from a given pixel of color c_i. To catch more local spatial information, the co-occurrence can also be defined by banded neighborhoods: this leads to the banded color correlogram. In practice, {0, 1, 3, 5, 7} are the most popularly used banded distances.

Texture Descriptors

The textural features describe local arrangements of image signals in the spatial domain or the frequency domain by some spectral transforms. There are many kinds of texture features, such as the Gray-Level Co-occurrence Matrix (GLCM), edge histogram features, multi-resolution simultaneous autoregressive models (MRSAR), wavelet coefficients, and Gabor textures. Specifically, the GLCM is the sufficient statistics of Markov random fields with multiple pairwise pixel interactions. The Edge histogram feature is used to characterize non-homogeneous texture regions. The MRSAR is a random field texture model that characterizes the geometric structure and the quantitative strength of interactions among neighbors. At present, most promising features for texture retrieval are multi-resolution features obtained from orthogonal wavelet transforms or from Gabor transforms in the frequency domain [7].

MPEG-7 has three texture descriptors: homogeneous texture, texture browsing, and edge histograms. The first two are based on the Gabor transform [2]. The Gabor transform offers the best simultaneous localization of spatial and frequency information. It emerges as an important visual primitive, and it is widely applied in tasks like edge detection, invariant object recognition, and compression [9, 10]. The 2-dimensional (2D) Gabor filters are defined as a series of multi-scale and multi-orientation cosine modulated Gaussian kernels. The Gabor texture representation of images is derived by convolving the image with the Gabor filters and implementing the convolved image efficiently by using Fast Fourier Transform (FFT). The MPEG-7 standard suggests using 6-orientation and 5-scale Gabor filters for the homogeneous texture descriptor and the texture browsing descriptor, which yields one forward 2D FFT for the image and 30 inverse 2D FFTs for the frequency-domain results.

MRSAR is another texture feature studied in this paper, that models the texture as second-order, non-causal Markov random fields [15]. MRSAR uses a 21x21 window sliding across the input image with fixed pixel steps (seven pixels in our experiments) in three resolutions. The least squares estimations are carried out at each resolution independently. Together with the standard deviation of the error term, five parameters are estimated for each resolution and concatenated for a 15-dimensional feature vector. The final feature is the mean and covariance matrix of the 15-dimensional feature on all sliding windows.

Shape Descriptors

The object's shape plays a critical role in searching for similar image objects (e.g., texts or trademarks in binary images or specific boundaries of target objects in images, etc.). In image/video retrieval, one expects that the shape description is invariant to scaling, rotation, and translation of the object. Shape features are less developed than their color and texture counterparts because of the inherent complexity of representing shapes. MPEG-7 supports region-based and contour-based shape descriptors [2]. However, these kinds of shape descriptors rely on the shape quality of shape extraction processes.

Recently, shape context has been proposed as a global shape descriptor, and it has demonstrated great success in image matching, recognition, and retrieval [11, 12]. It contains two steps: shape extraction and feature formulation. In practice, the shape can be provided by boundary detector, edge detector, or segmentation boundary. Our implementation adopts the simplest Canny edge detector. For each shape point *p*, it calculates the distance *r* and orientation *θ* between the point *p* and other shape points, and then it quantizes the pair (*r*, *θ*) into nine bins of a log-polar coordinate as shown in Figure 2. The 9-bin histogram is used to represent features at point *p*. Finally, the histogram of each selected key point is flattened and concatenated to form the context description of the shape.

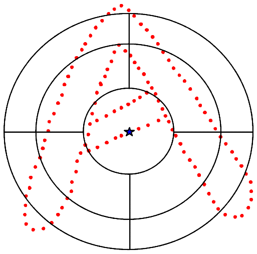


Figure 2: An example of shape context for the reference point

Localization Descriptors

Local descriptors for regions of interest have proved to be very successful in applications such as object recognition, image/video retrieval, and matching different views of object and scene [12]. They are distinctive, robust to occlusion, and do not require segmentation. The idea is to detect image regions that are covariant to a class of transformations, and these regions are then used as support regions to compute invariant descriptors. MPEG-7 contains a region locator and spatial-temporal locators [2]. In this paper we only discuss one of the most widely used localization descriptors: the scale-invariant feature transform (SIFT), which is a known invariant to changes in illumination, image noise, scaling, and small changes in viewpoint [13].

SIFT feature detection can be divided into four steps. The first step detects local extrema in scale-space. SIFT progressively blurs the input image with the Gaussian kernel, resulting in a series of blurred images. Then, each blurred image is subtracted from its direct neighbors (called scale space) to produce a new series of difference of Gaussian (DoG) images. Thereafter, a specific blob detection is conducted at each pixel in the image by comparing the pixel to its eight direct neighbor pixels and 18 neighbor pixels from direct neighbored blur images in the scale space. The second step localizes key points from the extrema in scale space by removing some lower-contrast and noise points. The third step assigns orientation for each key point, and computes histograms of gradient directions in a 16x16 window at each key point. The fourth step formulates the key point descriptor, which is a 128-dimensional vector of the normalized histogram.

Motion Descriptors

There are four motion descriptors: camera motion, motion trajectory, parametric motion, and motion activity in MPEG-7, which characterize 3-D camera motion parameters, temporal evolution of key points, the motion of regions, and the intensity or pace of motion, respectively [2]. Some MPEG video compression methods already encode macro-block level motion vectors. However, when the pixel-level or object-level motion estimation is required, we must resort to other techniques such as optical flow.

As motion can be represented as vectors originating or terminating at pixels in a digital image sequence, optical flow denotes a vector field defined across the image plane that can wrap images from previous to the next [14]. Estimating the optical flow is very useful in pattern recognition, computer vision, and other image-processing applications. In this work, we study the Lucas-Kanade method, which is known as the most popular two-frame differential method for optical flow estimation. This method tries to calculate the motion between two image frames that are taken at times *t* and *t*+*δt* at every pixel position. As a pixel at location (*x*, *y*, *t*) with intensity *I*(*x*, *y*, *t*) will have moved by *δx*, *δy*, and δ*t* between the two frames, optical flow assumes that parts of the objects are the same at the two time slices, i.e., $I(x+\delta x, y+\delta y, t+\delta t) = I(x, y, t)$. With first-order Taylor expansion of the left side, and omitting higher-order terms, we have the basic constraint $I_x V_x + I_y V_y + I_t = 0$. The Lucas-Kanade method assumes that the flow (V_x, V_y) is constant in a small window with *n* pixels, and then it yields *n* linear equations when taking the *n* pixels into the basic constraint. Since there are more equations than unknown variables (i.e., *n*>2), the system is over-determined and can be solved by the least squares method.

OPTIMIZATION AND PARALLELIZATION METHODOLOGY

In this section, we present an optimization and parallelization methodology, characterize different schemes and issues in parallelization, and provide some insights on how to parallelize these video analysis features on a multi-core processor.

Serial Performance Optimization

Before diving into the parallelization study, we first describe several optimization techniques to improve the application's performance. Some optimization can improve both serial and parallel performance. Following we show some widely used techniques we used in a CBVIR application optimization:

- Generic optimization techniques, like loop optimizations, etc.
- SIMD optimization to leverage the data-level parallelism (DLP) architecture features provided by the modern processor.
- Cache-conscious optimization to improve data locality. This is more pronounced for the parallel program due to a reduction of last-level cache misses as well as off-chip bandwidth demands.

Besides manual code optimization, we also extensively use Intel® performance libraries to improve performance. The libraries include the Intel® Performance Primitives (IPP) [16] and the Intel® Math Kernel Library (MKL) [17]. For example, Gabor features use the function *fftwf_execute* to execute discrete Fourier transform for Gabor filters. To achieve better performance we modify the linked library from the open sourced FFTW library to the Intel MKL. The FFTs in the MKL are highly optimized for the latest Intel dual-core and quad-core processors and can provide significant performance gains over alternative libraries for medium and large transform sizes.

Parallelism and Parallel Scheme Study

Usually, thread-level parallelism can be exploited in different ways. There are two primary decomposition methods in the design of a parallel program, i.e., data decomposition and task decomposition methods. The former divides the computations among multiple threads based on the different sections of data. The latter operates on a set of tasks that can run in parallel. Both types of parallelism can be used in the same program and no one method is always better than the other. However, in a CBVIR system, the majority of the work is conducted on 2-D images, which have abundant data parallelism at the picture-level, row-level, and even pixel-level. The selection of data parallelism is a natural choice to make use of the inherent parallelism. Further, to meet the real-time processing capability for these on-line video applications, it is important to extract the fine-grained parallelism within each image instead of exploiting the coarse-grained parallelism at the frame level.

We perform a detailed analysis on these visual features, and we restructure the data and code in order to facilitate the use of threading models. In the following section, we use several examples to demonstrate how to design proper parallel schemes for CBVIR applications.

The major work of the color correlogram consists of counting the color histograms for each pixel. The most straightforward way to do this is to partition the image into several tiles and have each tile accumulate the color data individually. To mitigate the lock contention overhead, in contrast to maintaining only a single histogram buffer, each thread is allocated a local histogram buffer to store the counting data individually. At the end of the parallel region, a reduction operation is conducted to accumulate the private data in each thread. To replicate the thread-private local buffers, we need an additional 20KB of memory per thread. In this way, we reduce the synchronization overhead and achieve better scalability performance. The pseudo code is shown in Figure 3:

```
malloc_local_histogram_array();
#pragma omp parallel
{
   #pragma omp for schedule(dynamic) nowait
   for(int y=0; y<height; y++)
      for(int x=0; x<width; x++)
            calc_correlogram(y, x);
}
merge_result_to_global_ histogram_array();
free_local_histogram_array();
```

Figure 3: The color correlogram code example

As shown in Figure 4, the parallelization of Gabor can be conducted at different granularities, such as filter level and FFT level. As we need to convolute images with multiple filters and transform them into frequency domains, the most straightforward way to do this is to perform coarse-grained-level parallelization on these independent filters. The filter-level parallelization scheme can make full use of the underlying processing capabilities with minimal effort. However, it has to prepare local buffers and construct an FFT plan for each filter. This leads to much larger memory consumption, however, and its working set cannot fit well into the last-level cache on a multi-core processor. In addition, the parallelism is also limited by the number of filters, e.g., in Gabor we only have 30 (5x6) filters: when the thread number goes beyond 16, it will incur significant load imbalance. Therefore, exploiting fine-grained parallelism within each filter is also equally important to better express the inherent parallelism. There are three kernels: convolution process, inverse FFT transform, and magnitude computing within each filter iteration. They can all be parallelized in a fine-grained fashion. As depicted in Figure 4, the parallelization of the convolution and magnitude computing kernels is straightforward. The FFT procedure is also internally parallelized by the MKL. The FFT-level parallelization holds a smaller working set by maintaining only one data copy for all the filters, which greatly improves the cache locality performance and reduces the off-chip memory bandwidth requirements. However, it suffers from fine-grained parallelization overhead and some non-parallel regions, e.g., the preparation stage in the FFT kernel. These will hurt the overall scaling performance. Therefore, we make a detailed comparison between these two parallel schemes, and we choose the one which has the best scaling performance in the final experiments.

Filter level parallelization

```
#pragma omp parallel for dynamic
for(int i=0; i<filter_number; i++)
{
   for(int k=0; k<image_size; k++)
      convolution(i,k);

   fftwf_execute(inverse_FFT_plans[i]);
   for(int k=0; k<image_size; k++)
      compute_magnitude(i,k);
}
```

FFT level parallelization

```
for(int i=0; i<filter_number; i++)
{
   #pragma omp parallel for schedule(static)
   for(int k=0; k<image_size; k++)
      convolution(k);

   // fftwf_execute is parallelized in the Intel® MKL
   fftwf_execute(inverse_FFT_plan);

   #pragma omp parallel for schedule(static)
   for(int k=0; k<image_size; k++)
      compute_magnitude(k);
}
```

Figure 4: Gabor parallelism selection

When designing a proper parallel scheme for an application, the best parallel scheme may not come from the best optimized serial algorithm. OpticalFlow (LK method [14], in OpenCV), uses a round-robin buffer to store seven rows of image data and traverses the image in a top-down manner. It has a very good data locality performance. However, the effort for parallelization is not trivial because the data in the buffer written by one thread will be used by another thread. We, therefore, need to use locks to protect this buffer. Frequent use of locks deteriorates the scaling performance. To break the data dependency among the threads, we use the original algorithm without employing an intermediate buffer for serial program acceleration. It turns out that we achieve much better scaling performance by simply performing parallelization at the row level.

To summarize, to obtain the desired parallel performance, the optimum parallelism depends on the decomposition method used and whether it resulted in the highest scalability performance, and it also depends on the data manipulation techniques and whether they efficiently improve the cache and memory utilizations. In addition, when a serial algorithm is not easy to parallelize in a straightforward way, we may resort to other ways to change the data structure and the control flow, or even modify the algorithm to make it more amenable to parallelization.

Parallel Performance Optimization

After studying the parallelism in the CBVIR applications, we further enhance their performance on multi-core processors. We use several Intel® software tools to analyze the parallel programs. For instance, we specify parallelism using OpenMP directives and compile using the Intel compilers. We use the Intel® Thread Checker [18] to test the correctness of the program, and the Intel® Thread Profiler [18] to collect parallel metrics for bottleneck identification. Furthermore, to understand the cache behavior, we use the Intel VTune™ Performance Analyzer [19] to collect different levels of cache data.

In parallelizing the CBVIR applications, we identify the parallel bottlenecks and classify them into three categories:

- **Load imbalance**. Load imbalance in a parallel section is a function of the variability of the size of the tasks and the number of tasks. For moderate multi-core processors, it is essential to keep all the cores busy by load balancing the tasks and minimizing overhead. If one core spends more time than the other cores working, the unbalanced load becomes a limiting factor for performance. In CBVIR, we use several techniques to improve the load balance performance, e.g., in MRSAR, a 2-dimension loop is merged into one dimension to enlarge the independent tasks. For almost all the workloads, we use a dynamic scheduling policy to minimize the load imbalance. Particularly, in SIFT, we manually use a “guided” scheduling policy, and the task size is chosen depending on the tasks within each parallelization loop. Since the tasks vary greatly in each iteration when the image in SIFT is downscaled, a guided scheduling policy helps to balance the size of tasks and scheduling overhead.
- **Synchronization overhead**. Often threads are not totally independent, which forces the program to add synchronization to guarantee the execution order of the threads. The frequent synchronization calls and the associated waiting operations will degrade the scaling performance on multi-core processors. Generally the synchronization is present in the form of critical section, lock, and barrier in the OpenMP implementation. In CBVIR, we largely eliminate the locks by carefully selecting the proper parallelism, e.g., we design a lock-free mechanism in the color correlogram to reduce the synchronization overhead. The shared histogram buffer is replicated into several private data copies. Each thread operates on each local data copy non-exclusively to avoid the mutual access of the shared histogram buffer. In addition, we make careful use of buffer manipulation for each thread, since frequent memory allocation/deallocation operations will cause severe lock contentions in the heap, and these requests are essentially run in serial in a parallel region.
- **Cache efficiency and memory bandwidth**. Good cache efficiency becomes even more important when using multi-core processors, since the maximum bus bandwidth remains unchanged. All cores collectively share a fixed-memory bandwidth; thus, it is important to design algorithms that are cache-conscious and can efficiently utilize the multi-core processing capability. In CBVIR, we designed the parallel programs with the cache performance in mind. We choose the most favorable granularity in terms of cache performance, where fine-grained threads are more cache-friendly than the coarse-grained ones, because more often they can make full use of data sharing instead of replicating buffers for each thread. In MRSAR, sometimes the data access patterns for a 2-D matrix is in column major rather than in row major. This breaks the spatial data locality when accessing the elements in the next row: the data are no longer contiguous. We manually transpose the 2-D matrix and selectively traverse the data according to the data access pattern. Furthermore, we use cache blocking techniques to improve the temporal data locality. We segment the whole data set into several tiles. This subset of data which can fit in cache is operated on all at once before moving on to the next set. Since the block of data can be processed several times before moving on to the next block,

this can significantly improve the cache locality performance.

Besides these general parallel-performance-limiting factors, we also investigate a few more issues specific to multi-core processing.

False sharing is a common pitfall in shared memory parallel processing. It occurs when two or more cores/processors are updating different bytes of memory that happen to be located on the same cache line. Since multiple cores cannot cache the same line of memory at the same time, when one thread writes to this cache line, the same cache line referenced by the other thread is invalidated. Any new references to data in this cache line by the second thread results in a cache miss and potentially huge memory latencies. Therefore, it is important to make sure that the memory references by the individual threads are to different non-shared cache lines. We manually resolve false sharing issues in two kernels of CBVIR, i.e., Shape Context and SIFT, by padding each thread's data element to ensure that elements owned by different threads all lie on separate cache lines. False sharing problems can also be identified during the tuning stage using the VTune Performance Analyzer, either through looking at some specific performance counters or identifying unexpected sharp increases in last-level cache misses.

Another specific performance-tuning technique is using a thread affinity mechanism to improve the cache performance. This minimizes the thread migration and context switches among cores. It also improves the data locality performance and mitigates the impact of maintaining the cache coherency among the cores/processors. Since multi-core processors are likely to have a non-uniform cache architecture (NUCA), the communication latency between different cores varies depending on its memory hierarchy. We use different thread scheduling policies to address this problem. When we find that a group of threads has high data sharing behavior, we can schedule these threads to the same cluster to utilize the shared cache for data transfer. (A cluster is a collection of closely coupled cores, e.g., two cores sharing the same L2 cache in an Intel® Core™2 Quad processor is termed a cluster.) On the other hand, for applications with high bandwidth demands, we prefer to schedule the threads on different clusters to utilize aggregated bandwidth. For instance, in SIFT, after the row-based parallelization, the image chunk assigned to one thread/core will be used by the other threads. Significant coherence traffic occurs when the image data do not reside in cores sharing the same last-level cache. Therefore, thread scheduling in the same cluster will mitigate the data transfer between loosely coupled cores that do not reside in the same cluster.

PERFORMANCE ANALYSIS ON MULTI-CORE SYSTEMS

In this section we first show twelve typical visual feature extraction workloads, which are accelerated by serial optimization. Then we parallelize six of the most compute-intensive workloads with the methodology introduced in the previous section. We evaluate the performance of these workloads on an 8-core system, which is a dual-socket, quad-core machine, with two Intel Core 2 Quad processors running at 2.33GHz. Each socket has four cores, and each core is equipped with a 32KB L1 data cache and a 32KB L1 instruction cache. The two cores on one chip share a 4MB L2 unified cache. The maximum FSB bandwidth is 21GB/s.

For the workloads studied in this work, we carefully choose the data sets to represent realistic scenarios. All the experiments are based on the TRECVID 2005 [20] developing data sets. The 141st and 142nd video sequences are chosen to evaluate the performance, which consists of around one hour of MPEG-1 (352x240 in resolution) videos and 791 key frames. The evaluations are directly performed on the extracted key frames.

Serial Performance Improvement

As shown in Figure 5, more than half of the workloads are formerly slower than real-time, i.e., 30 frames per second (FPS), in the serial performance on an 8-core system. After a series of optimizations, these kernels achieved an average of 3.3x speedup, about 60% of which came from using the Intel highly optimized libraries and the SIMD optimization. Even so, five workloads, Correlogram, MRSAR, Gabor, SIFT, and OpticalFlow, are still slower than real time. To harness the power provided by a multi-core system through exploiting thread-level parallelism, we further parallelized these workloads and analyzed their performance on an 8-core system. In addition, to make our work more comprehensive, we also included a representative shape descriptor, Shape Context, in the parallelization study.

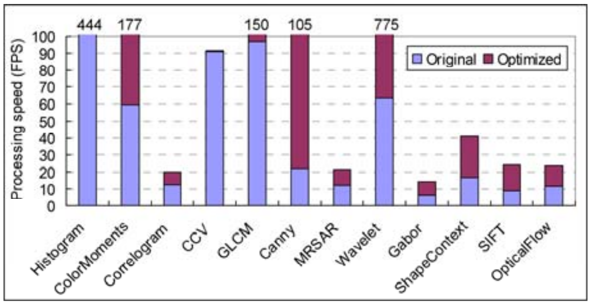


Figure 5: Serial processing speed (FPS) of CBVIR workloads on an 8-core system

Performance Scalability Analysis

These six workloads scale very well as the number of threads increases, as shown in Figure 6. Four of them exhibit almost linear speedups and two achieve quite respectable speedups. That is, CBVIR workloads can efficiently use the computational power provided by multi-core processors.

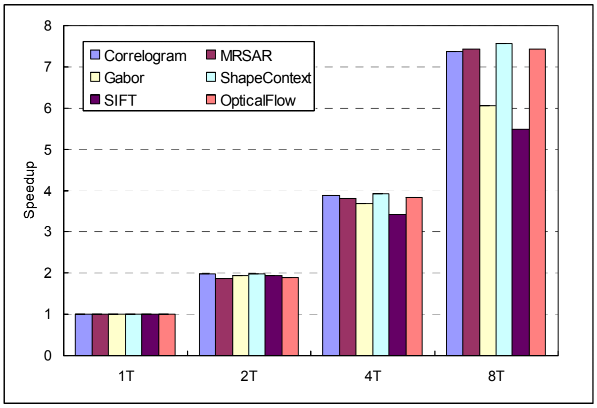


Figure 6: Scalability of parallel CBVIR workloads on an 8-core system

To fully understand the scaling limiting factors on an 8-core system, we characterize the parallel performance from the high-level general parallel overheads, e.g., synchronization penalties, load imbalance, and sequential regions, to the detailed memory hierarchy behavior, e.g., cache miss rates and FSB bandwidth.

We profile them with the Intel Thread Profiler to see their general parallel limiting factors. From Figure 7, we can see that the parallel region dominates in the execution time breakdown, which suggests these CBVIR workloads expose good parallel performance metrics. However, some workloads, especially SIFT, suffer a lot from load imbalance when the number of threads increases to four and eight, which leads to the poor speedup of SIFT. If we assume the parallel region can scale perfectly, Gabor and SIFT should achieve theoretical speedups of 7.6 and 6.2, respectively, on eight cores. The theoretical speedups are much higher than the practical results shown in Figure 6. Therefore, we believe the scalability of our workloads is also limited by some other factors.

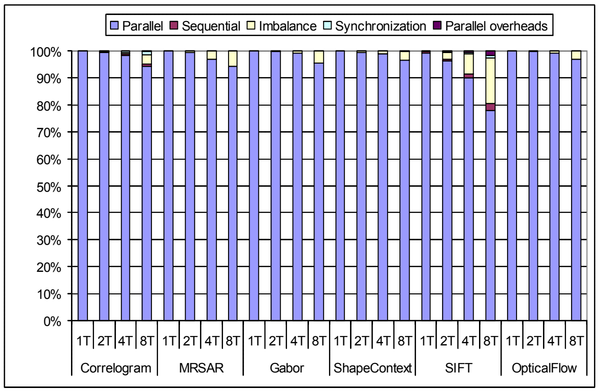


Figure 7: Execution time breakdown

Besides the general scalability performance factors, the memory subsystem also plays an important role in identifying the scaling performance bottlenecks. For further assurance, we get the memory-hierarchy micro-architectural statistics with the Intel VTune Performance Analyzer as shown in Figure 8. The figure shows that L1 cache miss rates vary little with the number of threads, while for some workloads L2 cache performance varies a lot when scaling the thread count. The L2 cache misses for most workloads is reduced when the number of threads increases to four or eight, because the system offers a larger size L2 cache from 4M to 8M and 16M. Since SIFT has a hierarchical parallel decomposition method, the downscale image has to be broadcast to all the private L2 caches after one iteration, thereby incurring significant cache coherency misses when we scale to four and eight cores.

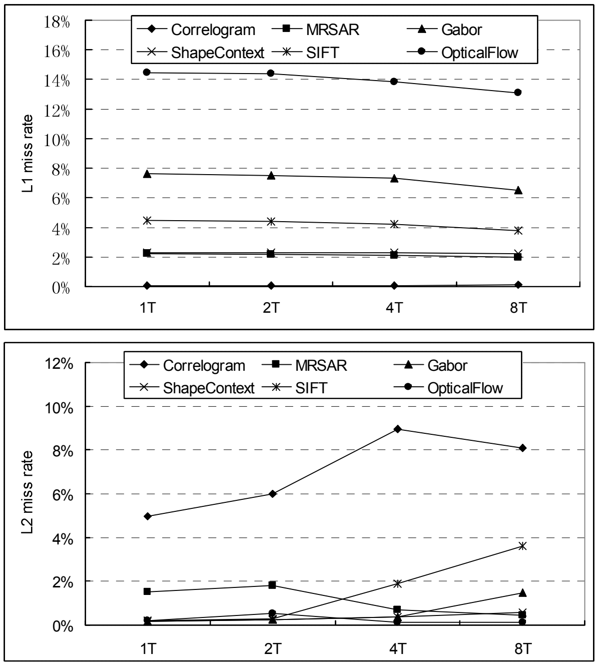


Figure 8: L1/L2 cache miss rates

Generally speaking, memory bandwidth is a key factor that may potentially limit the speedup on multi-core systems. Figure 9 shows how the average FSB bandwidth utilization varies with the number of threads. The bandwidth usages of all workloads are far below the saturated FSB bandwidth capacity supported by the system. This seems to indicate bus bandwidth does not limit the scalability of our workloads on an 8-core system. However, the scalability is limited by the instantaneous bandwidth usage for some workloads, such as Gabor. We perform interval sampling of the memory subsystem behavior over time. Figure 10 shows a representative phase of the bandwidth usage over time for this workload on eight cores. Several modules in this workload have higher bandwidth requirements than the saturated bandwidth provided by the system.

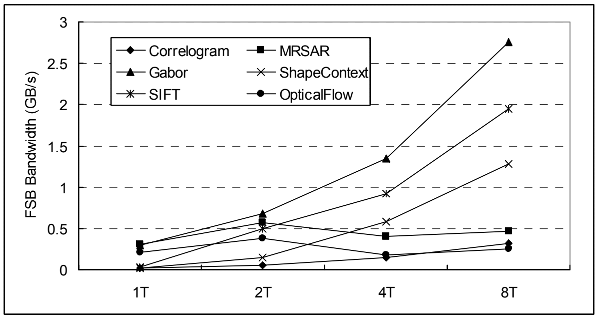


Figure 9: Average FSB bandwidth utilization vs. number of threads

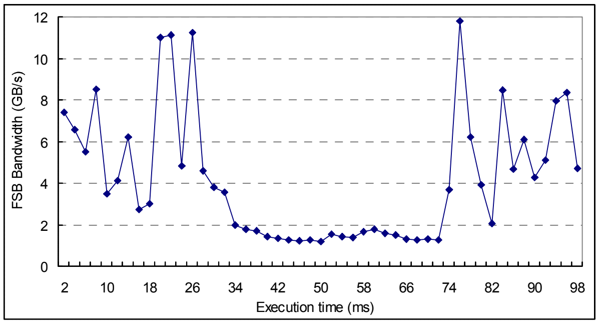


Figure 10: Bandwidth usage over time for eight-threaded Gabor workload

In addition to studying the memory sub-system performance, we also use different thread-scheduling mechanisms to further improve their performance on a multi-core system. As mentioned earlier, there are three scheduling policies: “clustered,” “non-clustered” and “os.” The “clustered” policy tries as much as possible to schedule all the threads to the closely-coupled cores; e.g., it schedules two threads to two cores residing in one chip. In contrast, the “non-clustered” policy tries to schedule the threads to the loosely coupled cores; e.g., it schedules two threads to two cores on two chips instead of one chip. The “os” is the default scheduling policy of the operating system, and it is non-aware of the hardware architecture.

Our results show that some workloads are sensitive to the scheduling policy. Figure 11 shows the scaling performance of Gabor and SIFT using different scheduling policies on an 8-core system. Gabor has better performance with the “non-clustered” policy, while SIFT has better performance with the “clustered” policy. This is because Gabor has a higher bandwidth requirement as shown in Figure 9. The “non-clustered” policy can make full use of the available L2 cache capacity and bandwidth, resulting in better cache performance as depicted in Figure 12. SIFT has better performance with the “clustered” policy because the data can reside in the same L2 cache all the while between several consecutive parallel regions. Otherwise, the data generated by one thread have to be transferred to another core that does not reside in the same L2 cache, yielding significant cache coherency traffic and slowing down the program. As shown in Figure 12, the “clustered” policy in SIFT has far fewer L2 cache misses and a lower FSB bandwidth utilization compared to the “non-clustered” policy. Hence, all the experimental results in the previous sections are obtained by choosing the best policy for each individual workload.

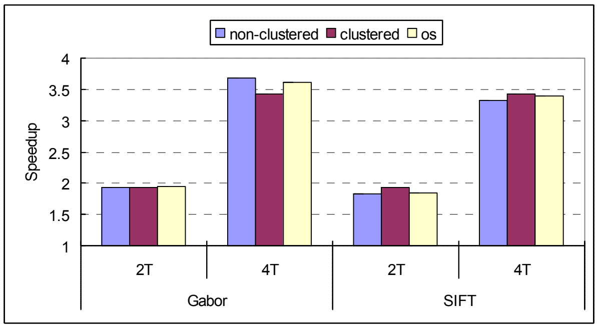


Figure 11: Effects of thread scheduling for two feature extraction workloads on an 8-core system

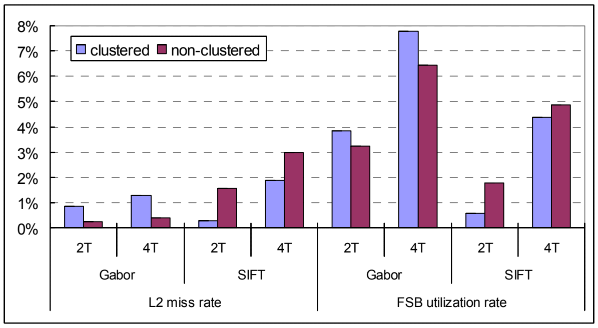


Figure 12: Effects of thread scheduling on L2 miss rate and FSB utilization rate for two feature extraction workloads on an 8-core system

CONCLUSION

CBVIR is becoming one of the best solutions to retrieve useful information from today's massive amount of video data. To accelerate CBVIR on multi-core systems, we optimize and parallelize a set of representative visual feature extraction workloads in CBVIR. We analyze their scalability and memory performance on an 8-core system and draw several conclusions.

Firstly, we choose appropriate parallel schemes for the applications in CBVIR. Exploring different levels of parallelism and choosing the most favorable are necessary to enable optimal performance on multi-core systems. Secondly, we incrementally optimize the parallel performance by mitigating the parallel performance limiting factors, e.g., load imbalance removal, designing cache-friendly data structures, using different thread-scheduling policies, etc. Thirdly, we find most of the CBVIR applications have very good scaling performance. The main scalability limiting factors for SIFT and Gabor are load imbalance and the amount of available system bandwidth. Finally, the CBVIR system is significantly accelerated on multi-core systems and offers enhanced performance to satisfy user requirements.

ACKNOWLEDGMENTS

We acknowledge the encouragement and help that we received from Dr. Bob Liang, Director of the Applications Research Lab. Our thanks go to Prof. Jianming Li and Prof. Bo Zhang from Tsinghua University for collaborating with us on the TRECVID evaluation and for providing some original source code for our analysis. We also acknowledge Qi Zhang for his contribution to this work during his internship with Intel. We also thank the reviewers for their valuable comments.

REFERENCES

[1] Michael L., Nicu, S., Chabane, D., and Jain, R., "Content-based Multimedia Information Retrieval: State of the Art and Challenges," *ACM Trans. on Multimedia Computing, Communications, and Applications*, 2006, pp. 1–19.

[2] *MPEG-7 Overview*, ISO/IEC/JTC1/SC29/WG11, N6828, 2004.

[3] Zhang, Q., Chen, Y., Li, J., and Zhang, Y., "Parallelization and performance analysis of video feature extractions on multi-core based systems," in *Proceedings of the 36th International Conference on Parallel Processing*, 2007.

[4] Yoshitaka, A. and Ichikawa, T., "A survey on content-based retrieval for multimedia databases," *IEEE Trans. On Knowledge and Data Engineering*, 11(1), 1999, pp. 81–93.

[5] Smeaton, A. F., Over, P. and Kraaij, W., "Evaluation campaigns and TRECVid," in *Proceedings of the 8th ACM International Workshop on Multimedia Information Retrieval* (MIR'06), pp. 321–330, 2006.

[6] Ma, W-Y, and Zhang, H-J, "Benchmarking of image features for content-based retrieval," *IEEE Conference on Signals, Systems & Computers*, 1998.

[7] Manjunath, B. S., Ohm, J.-R., Vasudevan, V., and Yamada, A., "Color and texture descriptors," *IEEE Trans on Circuits and Systems for Video Technology*, 11(6), pp. 703–715, 2001.

[8] Huang, J., Kumar, S. R., Mitra, M., Zhu, W.J., and Zabih, R., "Spatial Color Indexing and Applications," *International Journal of Computer Vision*, 35(3), pp. 245–268, 1999.

[9] Lee, T. S., "Image representation using 2D Gabor wavelets, *IEEE Trans. PAMI*, 18(10), pp. 959–971, 1996.

[10] Manjunath, B. S., and Ma, W-Y, "Texture features for browsing and retrieval of image data," *IEEE Trans. PAMI*, 18 (8), pp. 837–842, 1996.

[11] Belongie, S., Malik, J., and Puzicha, J., "Shape Matching and Object Recognition Using Shape Contexts," *IEEE Trans. PAMI*, 24(4), pp. 509–522, 2002.

[12] Mikolajczyk, K. and Schmid, C., "A performance evaluation of local descriptors," *IEEE Trans. PAMI*, 27(10), pp. 1615–1630, 2005.

[13] Lowe, D. G., "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, 60(2), pp. 91–110, 2004.

[14] Lucas B. and Kanade T., "An iterative image registration technique with an application to stereo vision," in *Proceedings of Imaging understanding workshop*, 1981.

[15] Mao, J. and Jain, A.K., "Texture classification and segmentation using multi-resolution simultaneous autoregressive models," *Pattern Recognition*, 25(2), pp. 173–188, 1992.

[16] "Intel® Integrated Performance Primitives," at http://www.intel.com/software/products/IPP

[17] "Intel® Math Kernel Library," at http://www.intel.com/software/products/MKL

[18] "Intel® Threading Analysis Tools," at http://www.intel.com/software/products/Threading

[19] "Intel VTune™ Performance Analyzer," at http://www.intel.com/software/products/VTune

[20] "NIST, TREC Video Retrieval Evaluation," at http://www-nlpir.nist.gov/projects/trecvid/

AUTHORS' BIOGRAPHIES

Yurong Chen is a researcher at the Microprocessor Technology Lab, Beijing. Currently, he conducts research on parallel processing of emerging applications, scalable workloads, benchmarking, and performance analysis for next-generation microprocessors/platforms. He joined Intel in 2004. Before that he did two years' postdoctoral research on large-scale scientific computing in the Institute of Software, Chinese Academy of Sciences. He received his Ph.D. degree from Tsinghua University in 2002. His e-mail is yurong.chen at intel.com.

Eric Li is a researcher in the Microprocessor Technology Lab, Beijing. Currently, he is working on media-mining technology development and performance analysis on multi-core architecture. Prior to this, he was involved in several projects related to bioinformatics, multimedia, and parallel computing. He received his M.S. degree from Tsinghua University in 2002 and joined Intel that same year. His e-mail is eric.q.li at intel.com.

Jianguo Li is a researcher in the Microprocessor Technology Lab, Beijing. Currently, he works on multimedia mining and parallel algorithm design and implementation. He has been involved in several projects related to sports video analysis and content-based media mining. He received his Ph.D. degree from Tsinghua University in June 2006 and joined Intel after graduation. His e-mail is jianguo.li at intel.com.

Yimin Zhang is a researcher in the Microprocessor Technology Lab, Beijing. He leads a team of researchers working on various statistical computing techniques and their scalability analysis, recently focusing on media mining, data mining, etc. He joined Intel in 2000. At Intel, he has been involved in several projects related to natural language processing and speech recognition, especially focusing on Chinese-named entity extraction and DBN-based speech recognition. He received his B.A. degree from Fudan University in 1993, his M.S. degree from Shanghai Maritime University in 1996, and his Ph.D. degree from Shanghai Jiao Tong University in 1999, all in Computer Science. His e-mail is yimin.zhang at intel.com.

Process Scheduling Challenges in the Era of Multi-core Processors

Suresh Siddha, Software Solutions Group, Intel Corporation
Venkatesh Pallipadi, Software Solutions Group, Intel Corporation
Asit Mallick, Software Solutions Group, Intel Corporation

Index words: multi-core, chip multi-processors, process scheduling, power management

ABSTRACT

In this era of computing, each processor package has multiple execution cores. Each of these execution cores is perceived as a discrete logical processor by the software. Any operating system that is optimized for Symmetric Multi Processing (SMP) and that scales well with the increase in processor count can instantaneously benefit from these multiple execution cores.

Design innovations in multi-core processor architectures bring new optimization opportunities and challenges for the system software. Addressing these challenges will further enhance system performance. The process (task) scheduler, in particular, one of the critical components of system software, is garnering great interest.

In this paper, we look at how the different multi-core topologies and the associated processor power management technologies bring new optimization opportunities to the process scheduler. We look into different scheduling mechanisms and the associated tradeoffs. Using the Linux* Operating System as an example, we also look into how some of these scheduling mechanisms are currently implemented.

As the multi-core platform is evolving, some portions of the hardware and software are being reshaped to take maximum advantage of the platform resources. We close this paper with a look at where future efforts in this technology are heading.

INTRODUCTION

In multi-core processor packages, each processor package contains two or more execution cores, with each core having its own resources (registers, execution units, some or all levels of caches, etc.). Even if the applications are not multi-threaded, multi-tasking environments will benefit from multi-core processors.

Design innovations of multi-core processor architectures mainly span the area of shared resources (caches, power management, etc.) between cores, core topologies (number of cores in a package, relationship between them, etc.), and platform topology (relation between cores in different packages, etc.). These innovations bring new opportunities and challenges to the system software. To exploit optimal performance, components such as the process scheduler need to be aware of the multi-core topologies and the task characteristics.

We start with a brief look at how the traditional process scheduler works and how the earlier challenges in the Symmetric Multi Processing (SMP), Non Uniform Memory Access (NUMA), and Simultaneous Multi-Threading (SMT) environments were addressed. We look at multi-core topologies with respect to core, cache, power management, and platform topologies. In the current generation of mainstream multi-core processors, the execution cores in a given processor package are symmetric and our focus in this paper is on such processors. Asymmetric multi-core processors are beyond the scope of this paper. We examine the need for a multi-core-aware process scheduler and look into the opportunities in this area. We examine different scheduling mechanisms for multi-core platforms under different load scenarios and the associated tradeoffs. With Linux as an example, we examine how some of these scheduling mechanisms are currently implemented. Finally, we close this paper with a look at current and future research in this field.

PROCESS SCHEDULER

The process scheduler, which is a critical piece of the operating system software, manages the CPU resource allocation to tasks. The process scheduler typically strives

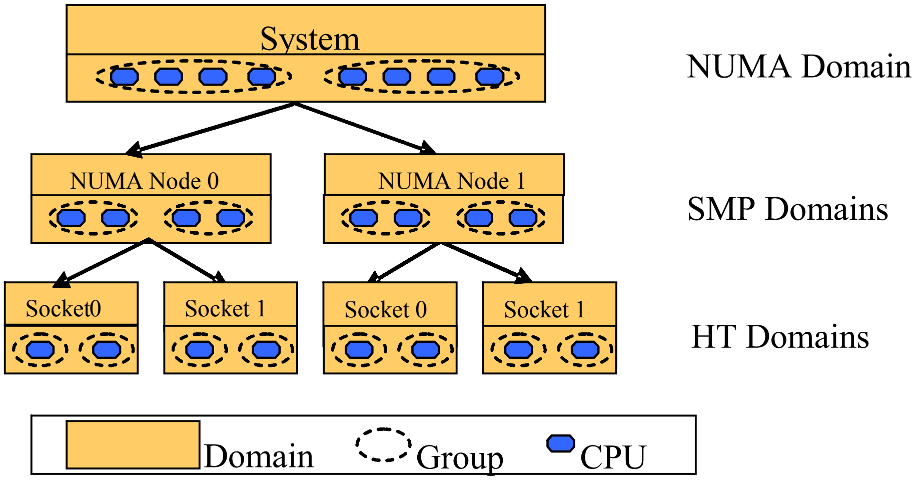


Figure 1: Process scheduling domain topology setup in the Linux kernel

for maximizing system throughput, minimizing response time, and ensuring fairness among the running tasks in the system.

Process priority determines the allotted time (time-slice) on a CPU and when to run on a CPU. In SMP, the process scheduler is also responsible for distributing the process load to different CPUs in the system.

In NUMA platforms, memory access time is not uniform across all the CPUs in the system and depends on the memory location relative to a processor. System software tries to minimize the access times, by allocating the process memory on the node that is closest to the CPU that the process is running on. As such, the cost associated with the process migration from one NUMA node to another is big. As a result, the process scheduler needs to be aware of NUMA topology. NUMA schedulers use some heuristics (such as tolerating more load imbalances between nodes and tracking the home node of each process, where the majority of process memory resides) to minimize the migrations and costs associated with the migrations.

In SMT (for example, Intel® Hyper Threading Technology), most of the core execution resources are shared by more than one logical processor. The process scheduler needs to be aware of the SMT topology and avoid situations where more than one thread sibling on one core is busy, while all the thread siblings on another core are completely idle. This will minimize the resource contention, maximize the utilization of CPU resources, and thus maximize system throughput. As the logical thread siblings are very close to each other, process migration between them is very cheap and as such, process load balancing between them can be done very often.

The process scheduler needs to consider all these topological differences while balancing process loads across different CPUs in the system. For example, the 2.6 Linux kernel process scheduler introduced a concept called scheduling domains [8] to incorporate the platform topology information into the process scheduler. The hierarchical scheduler domains are constructed dynamically depending on the CPU platform topology in the system. Each scheduler domain contains a list of scheduler groups having a common property. The load balancer runs at each domain level, and domain properties dictate the balancing that happens between the scheduling groups in that domain. On a high-end NUMA system with SMT capable processors, there are three scheduling domains, one each for SMT, SMP, and NUMA, as shown in Figure 1.

MULTI-CORE TOPOLOGIES

In most of the multi-core implementations, to make the best use of the resources and to make inter-core communication efficient, cores in a physical package share some of the resources. For example, the Intel® Core™2 Duo processor has two CPU cores sharing the Level 2 (L2) cache (Intel® Advanced Smart Cache), as shown in Figure 2. The Intel® Core™2 Quad processor has four cores in a physical package with two last-level (L2) caches. Each of the L2 caches is shared by two cores. Going forward, as more and more logic gets integrated into the processor package; more resources will be shared between the cores on the die.

If only one of the cores in the package is active, a thread running on that core gets to use all the shared resources, resulting in maximum resource utilization and peak performance for that single thread. If multiple threads or processes run on different cores of the same physical package and if they share data that fit in the cache, then the shared last-level cache between cores will minimize the data duplication. This sharing, therefore, results in more efficient inter-thread communication.

Multi-core Power Management

In typical multi-core configurations, all cores in one physical package reside in the same power domain (voltage and frequency). As a result, the processor performance state (P-state) transitions for all the cores need to happen at the same time. If one core is busy running a task at P0, this coordination will ensure that other cores in that package can't enter low-power P-states, resulting in the complete package at the highest power P0 state for optimal performance.

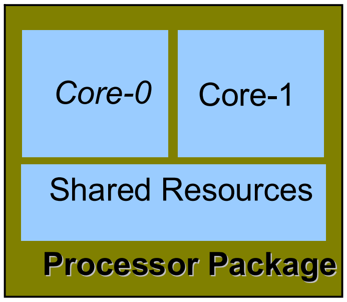


Figure 2: Dual-core package with shared resources

Since each execution core operates independently, each core block can independently enter a processor power state (C-state). For example, one core can enter lower power C1 or C2 while the other executes code in the active power state C0. The common block will always reside in the numerically lowest (highest power) C-state of all the cores. For example, if one core is in C2 and another core is in C0, the shared block will reside in C0.

Intel Dynamic Acceleration Technology

Intel® Dynamic Acceleration Technology [7], available in the current Intel Core 2 processor family, increases the performance of single-threaded applications. If one core is in deep C-state, some of the power normally available to that idle core can be applied to the active core while still staying within the thermal design power specification for the processor. This increases the speed at which a single-threaded application can be executed, thereby improving the performance of the application.

MULTI-CORE SCHEDULING

Shared resource topologies in multi-core platforms pose interesting challenges and opportunities to the system software. Shared resources between cores like shared cache hierarchy, provide good resource utilization and make inter-core communication efficient. However, heterogeneous data access patterns of memory-intensive tasks running on the cores sharing caches can lead to cache contention and sub-optimal performance. Contention and its impact on performance depend on the resources shared, the number of active tasks, and the access patterns of the individual tasks. A fair amount of CPU time allocated to each task by the process scheduler will not essentially translate into efficient and fair usage of the shared resources. The main challenge before the process scheduler is to identify and predict the resource needs of each task and schedule them in a fashion that will minimize shared resource contention, maximize shared resource utilization, and exploit the advantage of shared resources between cores. To achieve this, the process scheduler needs to be aware of multi-core, shared resource topology, resource requirements of tasks, and the inter-relationships between the tasks.

In the following sections, we describe some of the multi-core scheduling mechanisms; challenges in exploiting optimal performance, and power savings in the SMP platform. We analyze the impact of Intel Dynamic Acceleration Technology and processor power management technologies on these scheduling mechanisms. We also look into some of the heuristics that today's system software can exploit to minimize the shared resource contention among the cores sharing resources.

Experimental Setup

For our experiments and analysis, we primarily considered a dual-package SMP platform, with each package having two cores sharing a 4MB last-level cache. Different workloads such as SPECjbb2000, SPECjbb2005, SPECfp_rate of CPU2000, and an in-memory database search (IMDS) are considered for our analysis. These workloads are widely known except for the IMDS one. The IMDS workload is a non-standard workload simulating CPU and memory behavior of a typical database search algorithm. This workload is considered because of its high cycles per instruction (CPI) characteristic when compared to the other workloads used in our experiments. Run to run variations of these workloads are within 1%. Each of these workloads was run three times and the middle number was used for the performance comparisons.

Some of these workloads (SPECjbb2000, IMDS, for example) spawn threads sharing a process address space and some (like SPECfp_rate) spawn different processes, each having its own address space. Platform under test is run at 3GHz processor frequency unless otherwise stated and doesn't support Intel Dynamic Acceleration Technology.

Scheduling on Cores Sharing Resources vs. Not Sharing

In this workload scenario, all the considered workloads were run in a 2-task configuration. For example, the SPECjbb workload was run in two warehouse configurations (where each warehouse is represented by a user-level thread), and the SPECfp rate was run in the base configuration with two users (where each user is represented by an individual process). Similarly, the IMDS workload used two threads (belonging to the same process) to process the database queries.

Table 1: Performance difference between scheduling two tasks running on two cores using different last-level cache vs. scheduling on cores sharing same last-level cache. The higher % means that scheduling on cores with different caches is better.

Workloads	% Performance improvement with scheduling on different last-level caches when compared to scheduling on same last-level caches
SPECjbb2005	13.22
SPECjbb2000	5.19
SPECfp_rate (base2000)	16
IMDS	1

With two running threads on a dual-core, dual-package SMP platform, the main choices before the process scheduler are to schedule the two running threads on the cores in the different (Option 1) or same (Option 2) packages. Option 1 will result in maximum resource utilization, and as the other core in each package is idle, there is no shared resource contention. Option 2 will result in one busy package (with both the cores busy running the tasks), and the other package being completely idle. While this is not the best solution from the resource utilization and shared resource contention (tasks running in one package may contend for shared resources between cores) perspective, this mechanism will take advantage of the data sharing between tasks, if any.

Table 1 shows the results of different workloads with different scheduling mechanisms on a dual-core, dual-package SMP platform. As shown in the table, all the workloads benefited from distributing the load to two different packages. This indicates that the considered workloads take advantage of the increased available cache and the shared resource (primarily last-level cache) contention is playing a significant role when both the tasks run on the same package. Moreover, the contention is present whether the running tasks belong to the same process (where there is some data sharing, for example, SPECjbb) or different processes (for example, SPECfp_rate of cpu2000). Among the workloads, the IMDS workload in fact performs almost the same, irrespective of the threads running on cores sharing the same or different packages. This is primarily because the workload doesn’t exhibit good locality of memory references and as such doesn’t get affected much by sharing the last-level cache between two threads.

Last-level Cache Size Influence on Shared Resource Contentions

Hardware designers and researchers are looking into different options (like optimum size, layout of shared resources, design and management of these shared resources) and solutions for maximizing resource utilization and at the same time minimizing the resource contention. Moore's law [6] is dictating the cache size increase on Intel® platforms from generation to generation. The current x86 generation of 65nm processors features up to 4MB of L2 cache in the dual-core version and up to 8MB in the quad-core version, and the leading-edge 45nm generation [1] of x86 processors sports up to 6MB of L2 cache in the dual-core version and up to 12MB in the quad-core version. The degree of cache associativity is increasing with the increase in cache size, leading to hit rate improvement and better utilization.

Figure 3 shows the impact of the last-level cache size on the process scheduling mechanisms for the workloads we considered in Table 1. While the platforms considered for this experiment are quite different from each other (different characteristics and properties), each of the platforms under test is configured to work as dual-core, dual-package platforms. Each of these platforms has a different last-level cache size shared by two cores that reside in the processor package.

The data in Figure 3 show that for the given task load (two tasks in our experiments), as the shared resources among the cores increases, one can expect that the amount of shared resource contention will decrease accordingly. For example, SPECfp_rate of CPU2000 was performing the same whether the two tasks were running in the same package or in different packages with 16MB of last-level cache. The impact of last-level cache size is fairly negligible for the two threaded IMDS workloads. As noticed before, this is primarily because of the poor locality of memory references.

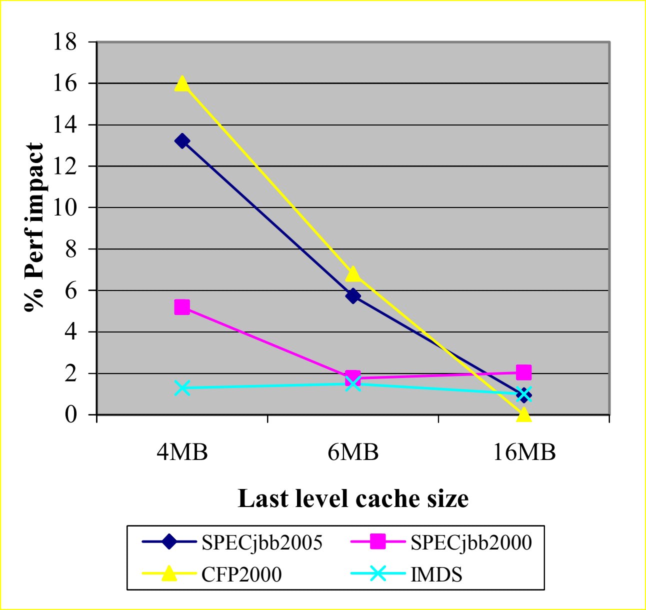


Figure 3: Impact of last-level cache size on the performance differences between scenarios doing scheduling on cores using same vs. different last-level cache. The smaller number indicates less resource contention in the scenarios when tasks run on cores sharing last-level cache.

Influence of Intel Dynamic Acceleration Technology

Intel Dynamic Acceleration Technology is currently available in Intel Core 2 Duo mobile processors. Let us look into the influence of this technology on process scheduling, if this support is available in the future for the server platforms supporting multiple processor packages.

Using the Linux kernel CPUfreq subsystem, we simulated the concept of Intel Dynamic Acceleration Technology in today's mainstream dual-package platform based on Intel Core 2 Duo processors. With the help of CPUfreq subsystem, the processor frequency can be changed to a specific value that the processor supports. Using this infrastructure, 3GHz-capable processors were run at 2.66GHz (a bin down) in the mode when the process scheduler schedules the two running tasks on two cores belonging to the same package. In the mode when the scheduler selects two different packages for running the two tasks, processors were run at 3GHz (as Intel Dynamic Acceleration Technology will enhance the speed of the active core, while one or more cores in the same package are idle). Figure 4 shows the performance numbers, which include the effects of running on different caches and at improved processor speeds as a result of dynamic acceleration.

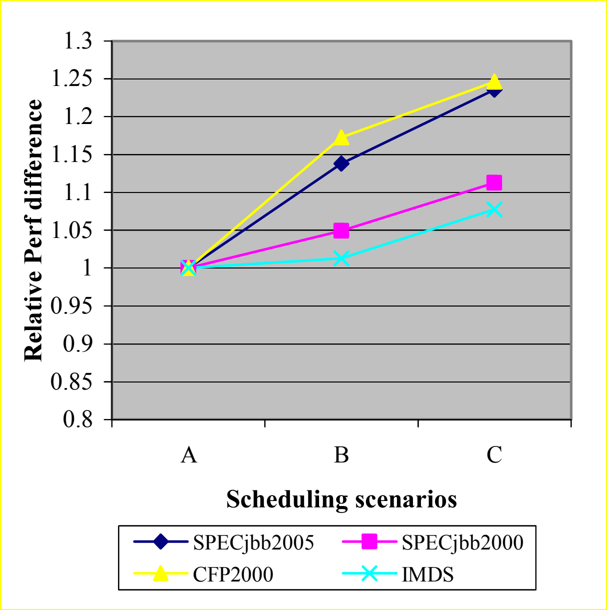


Figure 4: Performance difference between running two tasks on a) cores running at 2.66GHz, sharing last-level cache vs. b) cores running at 2.66GHz, different last-level cache vs. c) cores running at 3GHz, different last-level cache

Figure 4 shows that the dynamic acceleration favors the scheduling policy of distributing the load among the available processor packages for optimal performance. In the presence of dynamic acceleration, IMDS workloads also benefited when the two IMDS threads were run on different packages.

Scheduling for Optimal Power Savings

Consider the same dual-package experimental system that we looked at before. If we have two runnable tasks, as observed in the previous sub sections, resource contention will be minimized when these two tasks are scheduled on different packages. But, because of the P-state coordination in the current generation of multi-core platforms, we are restricting other idle cores in both packages to run at higher power P-state (voltage/frequency pair). Similarly, the shared block in both packages will reside in higher power C0 state (because of one busy core). This will result in a non-optimal solution from a power-savings perspective. Instead, if the scheduler picks the same package for both tasks, other packages with all cores being idle, will transition into the lowest-power P and C-state, resulting in more power savings. For optimal power savings, the number of physical packages carrying the load needs to be minimized. But as the cores share resources (like last-level cache) as seen in previous sections, scheduling both tasks to the same package may or may not lead to optimal behavior from a performance perspective.

Task Group Scheduling

In the workload scenario where all the shared resources and packages are busy, the main challenge for the process scheduler is to schedule the tasks in such a way that will minimize the shared resource contention and take maximum advantage of the shared resources between cores.

If all the running tasks are resource intensive, the challenge before the process scheduler is to identify the tasks that share data and schedule them on the cores sharing the last-level cache. This will help minimize the shared resource contentions and shared data duplication. This will also result in efficient data communication between the tasks that share data. The system software has some inherent knowledge about data sharing between tasks. For example, threads belonging to a process share the same address space and as such share everything (text, data, heap, etc.). Similarly, processes attached to the same shared memory segment will share the data in that segment. The process scheduler can do optimizations such as grouping threads belonging to a process or grouping processes attached to the same shared memory segment and co-schedule them in the cores sharing the package resources. To highlight the group-scheduling potential, we ran two instances of the SPECjbb workload, with each instance having two warehouses (each warehouse represented by a thread) on the dual-core, dual-package platform, considered before. Table 2 shows that grouping the threads belonging to a process onto the same package takes advantage of shared resources between cores and helps minimize the shared resource contentions.

Table 2: Performance improvement seen when threads belonging to a process scheduled to two cores residing on same package when compared to scheduling them on different packages. Workload considered is with two instances of SPECjbb in a two warehouse (two processes with two threads each) configuration.

Workloads	% Throughput improvement
SPECjbb2005	10
SPECjbb2000	7.5

Scheduling Challenges

For exploiting optimal performance, the process scheduler needs to schedule tasks in such a way that all the platform resources are used effectively. And this effective mechanism varies with workload, processor, platform topology, and system load.

Some workloads will exploit optimal benefit when the tasks are scheduled on the cores that share resources. For example, the IMDS workload performed the same, whether the tasks were run on cores sharing resources or not. For such workloads, in the presence of idle packages, by scheduling the tasks on the cores residing in a package, optimal performance and power-savings will be achieved. Similarly, workloads that share data and take maximum advantage of the shared resources between cores will achieve optimal performance when run on cores that are closer. For example, if the data shared between the tasks are modified and exchanged often or if one executing task prefetches data for the other task, optimal performance will be achieved when the tasks are scheduled closer to each other.

For some workloads, even in the presence of data sharing, distributing the load among the available idle packages will lead to optimal performance. This distribution will lead to shared data duplication in the caches of the packages carrying the load. If the shared data are mostly read-only, this data duplication still may be better than leaving the shared resources idle. In this scenario, tasks can take advantage of the increased shared resources (caches in our considered setup) that are available and can cache more shared data and/or task private data.

The presence of technologies, like dynamic acceleration, influence the process scheduling mechanisms for some workloads in the presence of idle cores and packages. As seen in Figure 4, when the load is uniformly distributed among the available packages, the resulting core speed increases, resulting from dynamic acceleration, helped achieve optimal performance. For some workloads, even in the presence of dynamic acceleration, running on the cores sharing caches may give optimal performance.

In future, as more cores are integrated into the processor package, the available shared resources will also increase accordingly. As such, the amount of shared resource contention will be minimal when few of the available cores in the package are busy (similar to what we see in Figure 3). The challenge for the process scheduler is to track the shared resource usages and the associated contentions. In such a scenario, the scheduler can minimize the processor packages carrying load, and when there is a contention for the shared resources, the scheduler can distribute the load to minimize resource contentions. To address the challenge, the process scheduler needs to track the micro-architectural information like the task's cycles per instruction (CPI) and how the task's CPI is affected by the co-running tasks in the other cores sharing resources. An individual task's CPI will also help the process scheduler in making decisions such as which tasks benefit most from the increased core speed that the dynamic acceleration technology brings in.

In a scenario where all the shared resources and packages are busy, the process scheduler needs to minimize the resource contention for exploiting optimal performance. For example, grouping CPU-intensive and memory-intensive tasks onto the cores sharing the same last-level cache will result in minimized cache contention. Task characteristics and behavior can be predicted using the micro-architectural history of a task by using performance counters. In the absence of such micro-architectural information, the system software can also use some heuristics to estimate the resource requirements. For example, one can use the number of physical pages that are accessed (using the Accessed bit in the page tables that manage virtual to physical address translation in x86 architecture) for certain intervals or use the tasks memory Resident Set Size (RSS). The process scheduler can use this information and group schedule tasks on the cores residing in a physical package with the goal of minimizing shared resource contention.

MULTI-CORE AWARE LINUX* PROCESS SCHEDULER

In this section, we consider the Linux operating system as an example and see how some of these scheduling challenges are addressed. A new scheduler domain representing multi-core characteristics has been added to the domain hierarchy of the Linux process scheduler. This scheduler domain helps identify cores sharing the same package and sharing resources, and it paves the way for the multi-core scheduler enhancements.

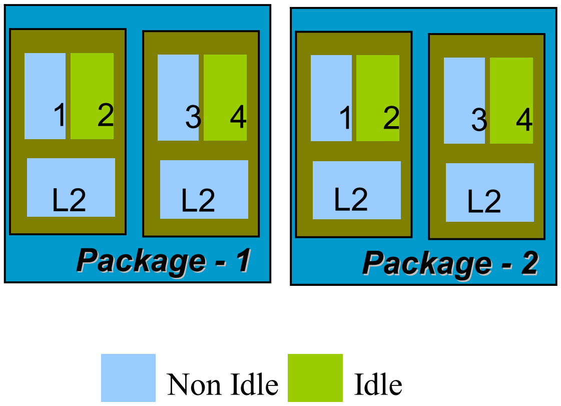


Figure 5: Scheduling mechanism showing four running tasks scheduled on four L2s on a dual package with Intel Core 2 Quad processors

By default, the current Linux kernel scheduler distributes the running tasks equally among the available last-level caches in an SMP domain. Within logical CPUs that share the last-level cache, the scheduler distributes the load equally, first among the available CPU cores and then among the available logical thread siblings. For example, consider a dual package SMP platform with Intel Core 2 quad processors with four running tasks. The multi-core-aware Linux process scheduler distributes these four running tasks among the four L2's that are available in the system as shown in Figure 5. This scheduling mechanism will lead to maximized resource utilization and minimized resource contention. And as observed in the previous sections, this will lead to optimal performance for most of the workloads. On platforms with dynamic acceleration technology, this mechanism will also result in optimal performance by making the cores run faster.

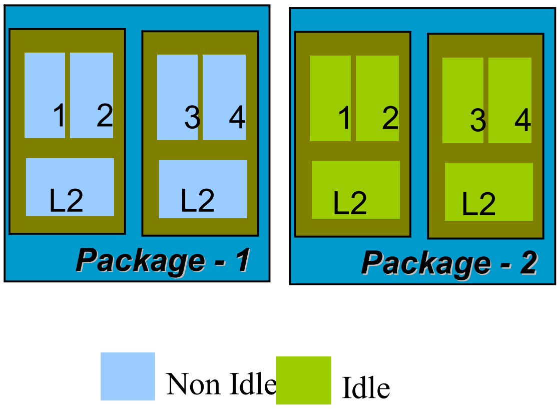


Figure 6: Scheduling mechanism showing four running tasks scheduled on four cores in one single package on a dual package with Intel Core 2 Quad processors

For optimal power savings or for workloads that benefit most by running the tasks on the cores sharing resources, the Linux kernel provides a tunable that can be set by an administrator. When this tunable is set, the process scheduler will try to minimize the packages in an SMP domain that carry the load. It will first try to load all the logical threads and cores in the package before distributing the load to another package. This policy is referred to as a power-savings policy. For example, consider the same four-task scenario on a dual package SMP platform with Intel Core 2 Quad processors. With the power-savings tunable set, all the four tasks will be run on the four cores residing in a single package as shown in Figure 6. Minimizing the number of packages that are active will lead to optimal power savings. As seen before, in the absence of dynamic acceleration support, this scheduling mechanism will not have any impact on performance for workloads such as the IMDS workload considered in Table 1.

Scheduling policy decisions are left to the administrator in the hope that the target workloads will be analyzed offline and the tunable will be set based on optimal performance and/or power-savings requirements. By default, the process scheduler takes a non-aggressive approach when distributing the load among the available shared resources.

RESEARCH WORK

Quite a bit of recent research in the process scheduler area is to do with trying to address multi-core scheduling challenges. For example, Micro Architectural Scheduling Assist [2] talks about tracking the shared resource usage with performance-monitoring counters and using this information for effective distribution of shared resource load. Another body of work in this area is the cache-fair algorithm [4] that tries to address the application performance variability that depends on the other co-scheduled threads in the same multi-core package. This algorithm uses an analytical model to estimate the L2 cache miss rate a thread would have if the cache were shared equally among all the threads, i.e., the fair miss rate. The algorithm then adjusts the thread's share of CPU cycles in proportion to its deviation from its fair miss rate. This algorithm showed a reduction of the effect of the schedule-dependent miss rate variability on the thread's runtime. The L2-conscious scheduling algorithm [5] separates all runnable threads into groups, such that the combined working set of each group fits in the cache. By scheduling a group at a time and making sure that the working set of each scheduled group fits in the cache, this algorithm reduces the cache miss ratios.

While the research shows promising results, it is far from being implementation ready and from inclusion in commercial operating systems. The main challenges of these algorithms include the dependency of the performance-monitoring counters (which are not designed primarily for process scheduling and which vary from processor generation to processor generation), the different algorithm phases (data collection phase and usage phase), applicability of mathematical models to wide heterogeneous workloads, and above all, incorporating this knowledge into the traditional process scheduler that works across wide multi-core topologies and platforms. One of the current focus areas is to turn this research into reality.

Most of the software algorithms exploit the differences in the individual task characteristics and their resource usages. Scenarios such as those in which all the tasks in the system have similar characteristics and resource requirements cannot be addressed by software alone with the current generation of multi-core hardware. CQoS [3] presents a new cache management framework for improving shared cache efficiency and improving system performance. It proposes options for priority classification, priority assignment, and priority enforcement to heterogeneous memory access streams. Hardware solutions like these help maximize resource utilization and minimize the impact on performance in the presence of shared resource contention.

As more logic gets integrated into the processor die, future work in this area will focus on the increasing shared resources between cores on the die and their interactions with the system software; the process scheduler in particular. In the area of multi-core processor power management, one of the areas that is making rapid progress is the reduction of idle processor power. In future platforms, as the power consumed by idle cores decreases and becomes independent of the busy cores in the packages, scheduling mechanisms for power savings need to be revisited.

CONCLUSION

In this paper, we showed that optimal performance can be exploited by making the process scheduler aware of the multi-core topologies and the task characteristics. Multi-core scheduling mechanisms and challenges are analyzed in an SMP environment. We looked at the impact of Intel Dynamic Acceleration Technology on these workload scenarios. Some of the group-scheduling heuristics that can enhance optimal performance are presented. We looked at how some of these multi-core scheduling mechanisms are implemented in the Linux operating system.

In future, one can expect the process scheduler to be micro-architectural aware for exploiting optimal performance. Similarly, one can expect that the research proposals and solutions in this area will drive future hardware and platform designs that will minimize the effects of shared resource contention and also assist the software in making and enforcing the right decisions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We thank David Levinthal for his mentoring support.

REFERENCES

[1] "Introducing the 45nm Next-Generation Intel® Core™ Microarchitecture," at http://www.intel.com/technology/architecture-silicon/intel64/45nm-core2_whitepaper.pdf

[2] Nakajima, Jun and Pallipadi, Venkatesh, "Enhancements for Hyper-Threading Technology in the Operating System–Seeking the Optimal Scheduling," *Workshop on Industrial Experiences with Systems Software*, Boston, MA, Dec. 2002.

[3] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," *18th International*

Conference on Supercomputing (ICS), San Malo, France, June 2004.

[4] Alexandra Fedorova, Margo Seltzer, Christopher Small and Daniel Nussbaum, "Performance Of Multithreaded Chip Multiprocessors and Implications For OS Design," in *Proceedings of the USENIX 2005 Annual Technical Conference*, Anaheim, CA, April 2005.

[5] Alexandra Fedorova, "Improving Performance Isolation on Chip Multiprocessors via an OS Scheduler," at http://www.eecs.harvard.edu/~fedorova/papers/fairsched.pdf

[6] "Moore's Law," at http://www.intel.com/technology/mooreslaw/index.htm

[7] "Intel® Centrino® Duo Processor Technology," at http://www.intel.com/products/centrino/duo/description.htm

[8] "Scheduling Domains," at http://lwn.net/Articles/80911/

AUTHORS' BIOGRAPHIES

Suresh Siddha is a Senior Staff Software Engineer in Intel's Open Source Technology Center. Suresh joined Intel in 2001 and works on enabling various Intel processor and platform features in the Linux kernel. His current focus is on multi-core technologies, the process scheduler, and system software scalability. His e-mail address is suresh.b.siddha at intel.com.

Venkatesh Pallipadi is a Senior Staff Software Engineer in Intel's Open Source Technology Center. He joined Intel in 2001 and works on enabling various core features in the Linux kernel across all Intel architectures. His current focus is on processor and platform power management. Prior to joining Intel, Venkatesh received his ME degree in Computer Science and Engineering from the Indian Institute of Science in Bangalore, India. His e-mail is venkatesh.pallipadi at intel.com.

Asit Mallick is a Senior Principal Engineer leading the system software architecture in the Intel Open Source Technology Center. He joined Intel in 1992 and has worked on the development and porting of numerous operating systems to Intel architecture. Prior to joining Intel, he worked in Wipro Infotech, India on the development of networking software. Asit earned his Masters degree in Engineering from the Indian Institute of Science, India. His e-mail address is asit.k.mallick at intel.com.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

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