Intel® Hyper-Threading Technology

Technical User's Guide

January 2003

Contents

Overview of Intel® Hyper-Threading Technology

Intel has extended and enhanced the microprocessor architecture over several generations to improve performance. But applications typically only make use of about one-third of processors' execution resources at any one time. To improve usage of execution resources, Intel introduced Hyper-Threading Technology, which enables better processor utilization and more efficient utilization of resources.

Included in this technical user's guide are:

- An overview of Hyper-Threading Technology and a description of how it increases the performance of the operating system and application software written to run on Intel® architecture-based processors.
- A discussion of how to maximize performance of an application using Hyper-Threading Technology.
- In-depth information on how to identify and lessen the impact of performance bottlenecks in applications using Hyper-Threading Technology.
- An overview of the resources available for programmers developing applications using Hyper-Threading Technology, including a comprehensive glossary.

To understand Hyper-Threading Technology and its role in application performance, it is first necessary to get a handle on some of the underlying multitasking, multithreading and multiprocessing concepts. The easiest way to keep the concepts straight is to review their evolution.

The Evolution of System and Processor Architectures

Originally, personal computing meant a desktop system with one processor that ran one program at a time. When a user ran a program, the operating system loaded it into memory and the processor was entirely devoted to its execution until the program completed. If the operating system needed the processor, it issued an interrupt. In response to the interrupt, the program would save its current state, suspend operations and surrender control to the operating system. MS-DOS,* for example, was a single-threaded operating system intended to run on a single processor – the simplest of all configurations.

While other ways of handling interrupts were introduced later on, including some that would allow two programs to run simultaneously – the result of these creative work-arounds early on was greatly reduced stability and reliability. This is because the software was trying to force the operating system to do something it was not designed to handle – switch between two running programs.

Single Processor Systems

Efforts to improve system performance on single processor systems have traditionally focused on making the processor more capable. These approaches to processor design have focused on making it possible for the processor to process more instructions faster through higher clock speeds, instruction-level parallelism (ILP) and caches. Techniques to achieve higher clock speeds include pipelining the microarchitecture to finer granularities, which is also called super-pipelining. Higher clock frequencies can greatly improve performance by increasing the number of instructions that can be executed each second. But because there are far more instructions being executed in a super-pipelined microarchitecture, handling of events that disrupt the pipeline, such as cache misses, interrupts and branch mispredictions, is much more critical and failures more costly.

ILP refers to techniques to increase the number of instructions executed each clock cycle. For example, many super-scalar processor implementations have multiple execution units that can process instructions simultaneously. In these super-scalar implementations, several instructions can be executed each clock cycle. With simple in-order execution, however, it is not enough to simply have multiple execution units. The challenge is to find enough instructions to execute. One technique is out-of-order execution where a large window of instructions is simultaneously evaluated and sent to execution units, based on instruction dependencies rather than program order.

Accesses to system memory are slow, though faster than accessing the hard disk, but when compared to execution speeds of the processor, they are slower by orders of magnitude. One technique to reduce the delays introduced by accessing system memory (called latency) is to add fast caches close to the processor. Caches provide fast memory access to frequently accessed data or instructions. As cache speeds increase, however, so does the problem of heat dissipation and of cost. For this reason, processors often are designed with a cache hierarchy in which fast, small caches are located near and operated at access latencies close to that of the processor core. Progressively larger caches, which handle less frequently accessed data or instructions, are implemented with longer access latencies. Nonetheless, times can occur when the needed data is not in any processor cache. Handling such cache misses requires accessing system memory or the hard disk, and during these times, the processor is likely to stall while waiting for memory transactions to finish.

Most techniques for improving processor performance from one generation to the next are complex and often add significant die-size and power costs. None of these techniques operate at 100 percent efficiency thanks to limited parallelism in instruction flows. As a result, doubling the number of execution units in a processor does not double the performance of the processor. Similarly, simply doubling the clock rate does not double the performance due to the number of processor cycles lost to a slower memory subsystem.

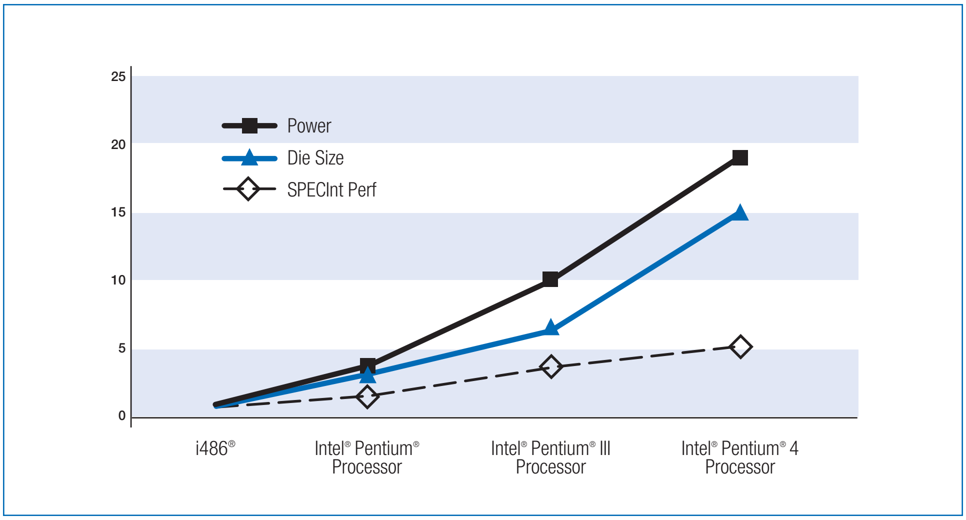


Figure 1 Single stream performance vs. cost.

Figure 1 shows the relative increase in performance and the costs, such as die size and power, over the last ten years on Intel® processors[1]. To isolate the impact of microarchitecture, this comparison assumes that the four generations of processors are on the same silicon process technology and that the speed improvements are normalized to the performance of an Intel486™ processor[2]. The Intel processor performance, due to microarchitecture advances alone, has improved integer performance five- or six-fold[3].

Multithreading

As processor capabilities have increased, so have demands on performance, which has increased pressure on processor resources with maximum efficiency. Noticing the time that processors wasted running single tasks while waiting for certain events to complete, software developers began wondering if the processor could be doing some other work at the same time.

To arrive at a solution, software architects began writing operating systems that supported running pieces of programs, called *threads*. Threads are small tasks that can run independently. Each thread gets its own time slice, so each thread represents one basic unit of processor utilization. Threads are organized into processes, which are composed of one or more threads. All threads in a process share access to the process resources.

[1] These data are approximate and are intended only to show trends, not actual performance.

[2] Although Intel processor history is used in this example, other high-performance processor manufacturers during this time period would have similar trends.

[3] These data are approximate and are intended only to show trends, not actual performance.

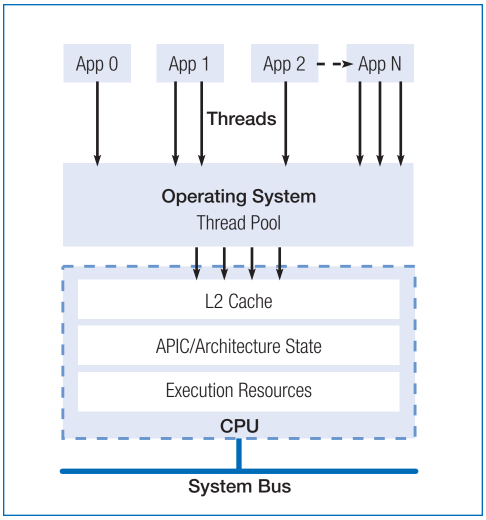


Figure 2 Multiple applications running on a single-processor system. An application can have one or more threads. For each application, a primary thread is created. The application can create more threads for secondary tasks by making calls to the operating system. Every thread is prioritized to use the processor resources. The operating system time slices the processor to run on each thread.

These multithreading operating systems made it possible for one thread to run while another was waiting for something to happen. On Intel processor-based personal computers and servers, today's operating systems, such as Microsoft Windows* 2000 and Windows* XP, all support multithreading. In fact, the operating systems themselves are multithreaded. Portions of them can run while other portions are stalled.

To benefit from multithreading, programs need to possess executable sections that can run in parallel. That is, rather than being developed as a long single sequence of instructions, programs are broken into logical operating sections. In this way, if the application performs operations that run independently of each other, those operations can be broken up into threads whose execution is scheduled and controlled by the operating system. These sections can be created to do different things, such as allowing Microsoft Word* to repaginate a document while the user is typing. Repagination occurs on one thread and handling keystrokes occurs on another. On single processor systems, these threads are executed sequentially, not concurrently. The processor switches back and forth between the keystroke thread and the repagination thread quickly enough that both processes appear to occur simultaneously. This is called functionally decomposed multithreading.

Multithreaded programs can also be written to execute the same task on parallel threads. This is called data-decomposed multithreaded, where the threads differ only in the data that is processed. For example, a scene in a graphic application could be drawn so that each thread works on half of the scene. Typically, data-decomposed applications are threaded for throughput performance while functionally decomposed applications are threaded for user responsiveness or functionality concerns.

When multithreaded programs are executing on a single processor machine, some overhead is incurred when switching context between the threads. Because switching between threads costs time, it appears that running the two threads this way is less efficient than running two threads in succession. If either thread has to wait on a system device for the user, however, the ability to have the other thread continue operating compensates very quickly for all the overhead of the switching. Since one thread in the graphic application example handles user input, frequent periods when it is just waiting certainly occur. By switching between threads, operating systems that support multi-threaded programs can improve performance and user responsiveness, even if they are running on a single processor system.

In the real world, large programs that use multithreading often run many more than two threads. Software such as database engines creates a new processing thread for every request for a record that is received. In this way, no single I/O operation prevents new requests from executing and bottlenecks can be avoided. On some servers, this approach can mean that thousands of threads are running concurrently on the same machine.

Multiprocessing

Multiprocessing systems have multiple processors running at the same time. Traditional Intel® architecture multiprocessing systems have anywhere from two to about 512 processors. Multiprocessing systems allow different threads to run on different processors. This capability considerably accelerates program performance. Now two threads can run more or less independently of each other without requiring thread switches to get at the resources of the processor. Multiprocessor operating systems are themselves multi-threaded, and the threads can use the separate processors to the best advantage.

Originally, there were two kinds of multiprocessing: asymmetrical and symmetrical. On an asymmetrical system, one or more processors were exclusively dedicated to specific tasks, such as running the operating system. The remaining processors were available for all other tasks (generally, the user applications). It quickly became apparent that this configuration was not optimal. On some machines, the operating-system processors were running at 100 percent capacity, while the user-assigned processors were doing nothing. In short order, system designers came to favor an architecture that balanced the processing load better: symmetrical multiprocessing (SMP). The "symmetry" refers to the fact that any thread – be it from the operating system or the user application – can run on any processor. In this way, the total computing load is spread evenly across all computing resources. Today, symmetrical multiprocessing systems are the norm and asymmetrical designs have nearly disappeared.

SMP systems use double the number of processors, however performance will not double. Two factors that inhibit performance from simply doubling are:

- How well the workload can be parallelized
- System overhead

Two factors govern the efficiency of interactions between threads:

- How they compete for the same resources
- How they communicate with other threads

When two threads both need access to the same resource – such as a disk drive, a record in a database that another thread is writing to, or any other system resource – one has to wait. The penalties imposed when threads have to wait for each other are so steep that minimizing this delay is a central design issue for hardware installations and the software they run. It is generally the largest factor in preventing perfect scalability of performance of multiprocessing systems, because running threads that never contend for the same resource is effectively impossible.

A second factor is thread synchronization. When a program is designed in threads, many occasions arise where the threads need to interact, and the interaction points require delicate handling. For example, if one thread is preparing data for another thread to process, delays can occur when the first thread does not have data ready when the processing thread needs it. Another example occurs when two threads need to share a common area of memory. If both threads can write to the same area in memory, then the thread that wrote first has to either check to make sure that what it wrote has not been overwritten, or it must lock out other threads until it has finished using the data. This synchronization and inter-thread management does not benefit from having more available processing resources.

System overhead is the thread management done by the operating system or application. With more processors running, the operating system has to coordinate more. As a result, each new processor adds incrementally to the system-management work of the operating system. This means that each new processor contributes less and less to the overall system performance.

Multiprocessor Systems

Today's server applications consist of multiple threads or processes that can be executed in parallel. Online transaction processing and Web services have an abundance of software threads that can be executed simultaneously for faster performance. Even desktop applications are becoming increasingly parallel. Intel architects have implemented thread-level parallelism (TLP) to improve performance relative to transistor count and power consumption.

In both the high-end and mid-range server markets, multiprocessors have been commonly used to get more performance from the system. By adding more processors, applications potentially get substantial performance improvement by executing multiple threads on multiple processors at the same time. These threads might be from the same application, from different applications running simultaneously, from operating-system services, or from operating-system threads doing background maintenance. Multiprocessor systems have been used for many years, and programmers are familiar with the techniques to exploit multiprocessors for higher performance levels.

In recent years, a number of other techniques to further exploit TLP have been discussed, such as:

- Chip multiprocessing
- Time-slice multithreading
- Switch-on-event multithreading
- Simultaneous multithreading

Hyper-Threading Technology brings the simultaneous multithreading approach to the Intel architecture.

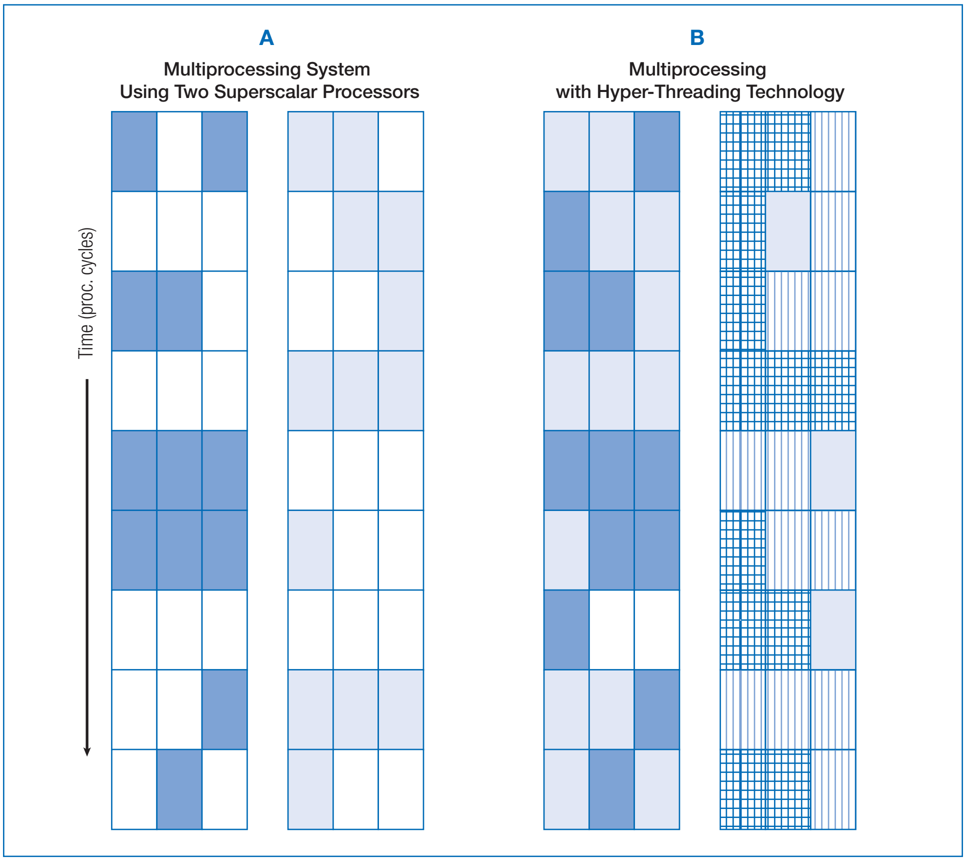

A
Multiprocessing System
Using Two Superscalar Processors
B
Multiprocessing
with Hyper-Threading Technology
Time (proc. cycles)

Figure 3 Simultaneous Multithreading.

A. Traditional multiprocessing with two physical processors. One processor is executing the blue thread, and the other is executing the light blue thread. The peak execution bandwidth is six instructions per cycle, three on each processor. The system may operate at less than peak bandwidth, as indicated by the large number of idle (white) execution units.

B. Hyper-Threading Technology on a multiprocessing system. This configuration shows a multiprocessor system with two processors featuring Hyper-Threading Technology. One processor is simultaneously executing the dark and light blue threads, while the other executes the patterned threads. Such a system operates closer to peak bandwidth, as indicated by the small number of idle (white) execution units.

Multitasking versus multithreading

Multitasking is the operating system's ability to run several programs simultaneously on a single processor by allocating the time slices of the processor to each program. For instance, if there are *n* tasks to perform, the operating system will divide up the time between the *n* tasks.

Multithreading facilitates work to be done in parallel. Multithreaded programming is implementing software to perform two or more tasks in parallel within the same application. Multithreading is spawning multiple threads to perform each task. If thread 1 is busy waiting for I/O to complete, thread 2 uses the processor during this time and then switches back to thread 1 to complete.

Hyper-Threading Technology

To keep up with today's demand for increasingly higher processor performance, traditional approaches to processor design have to be re-examined. Microarchitecture techniques used to improve processor performance in the past, such as super-pipelining, branch prediction, super-scalar execution, out-of-order execution and caches, have allowed microprocessors to become more complex, provide more transistors and consume more power. These processors operate faster, but speed alone does not always improve processor performance. As an example, consider code that produces cache misses frequently. A higher frequency processor will only miss the cache faster. Increasing processor frequency alone does not do anything to improve processor-utilization rates.

What is needed is an approach that allows the processor resources to be used in a highly efficient way. Hyper-Threading Technology is designed to increase the ability to use a processor efficiently. Hyper-Threading Technology boosts performance by allowing multiple threads of software applications to run on a single processor at one time, sharing the same core processor resources.

Hyper-Threading Technology is a form of simultaneous multithreading technology (SMT) introduced by Intel. Architecturally, a processor with Hyper-Threading Technology consists of two logical processors, each of which has its own processor architectural state[4]. After power-up and initialization, each logical processor can be individually halted, interrupted or directed to execute a specified thread, independently from the other logical processor on the chip. Unlike a traditional dual processor (DP) configuration that uses two separate physical processors (such as two Intel® Xeon™ processors), the logical processors in a processor with Hyper-Threading Technology share the execution resources of the processor core. These resources include the execution engine, the caches, the system-bus interface and the firmware.

[4] The architectural state that is duplicated for each logical processor consists of the processor data registers, segment registers, control registers, debug registers, and most of the model specific registers (MSRs). Each logical processor also has its own advanced programmable interrupt controller (APIC).

In Figure 4, the left-hand configuration represents a traditional multiprocessor system with two discrete physical processors. Each processor has its own set of processor-execution resources and its own single architectural state. The right-hand configuration represents an Intel Xeon processor family-based multiprocessor system where each processor features Hyper-Threading Technology. As you can see, the architectural state for each processor is duplicated, but each still has one set of execution resources. When scheduling threads, the operating system treats the two separate architectural states as two separate "logical" processors.

Multiprocessor-capable software applications can run unmodified with twice as many logical processors to use. Each logical processor can respond to interrupts independently. The first logical processor can track one software thread while the second logical processor tracks another software thread simultaneously. Because the two threads share one set of execution resources, the second thread can use resources that would be idle if only one thread were executing. The result is an increased utilization of the execution resources within each physical processor package.

Hyper-Threading Technology represents a new approach to improving the instruction throughput of processors that are targeted for servers, high-performance workstations and desktops. It also provides a view into the future of microprocessor design where the performance of a processor when executing a specific type of application or the space and power requirements of a physical processor within a server may be as important as its raw processing speed.

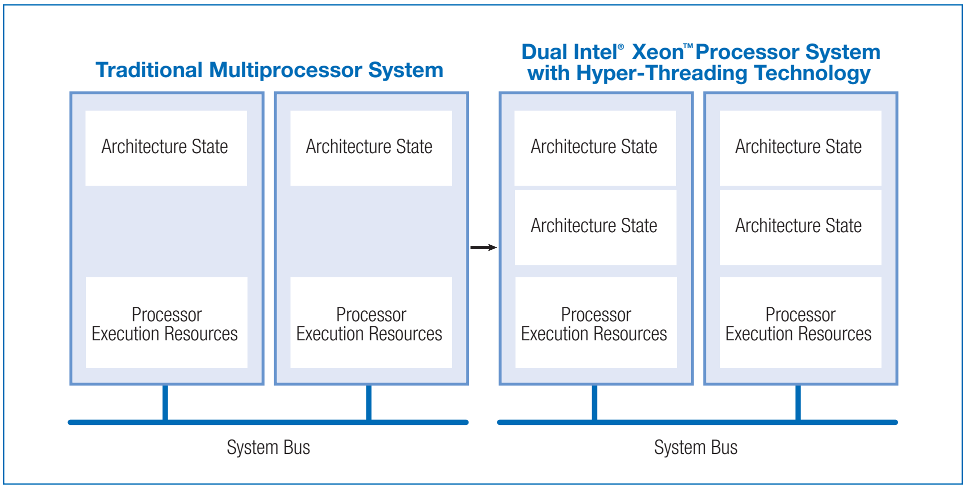
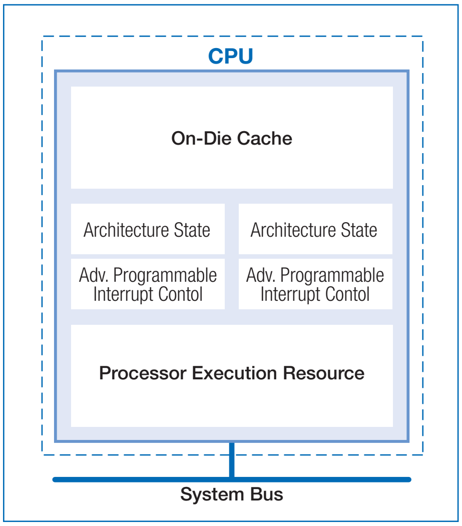

Traditional Multiprocessor System
Dual Intel® Xeon™ Processor System with Hyper-Threading Technology
Architecture State
Architecture State
Architecture State
Architecture State
Architecture State
Architecture State
Processor Execution Resources
Processor Execution Resources
Processor Execution Resources
Processor Execution Resources
System Bus
System Bus

Figure 4 Hyper-Threading Technology enables a single physical processor to execute two separate code streams (or threads) concurrently. A multiprocessor system with Hyper-Threading Technology duplicates the architectural state on each physical processor, providing two "logical" processors per physical processor. It is achieved by duplicating the architectural state on each processor, while sharing one set of processor-execution resources. The architectural state tracks the flow of a program or thread, and the execution resources are the units on the processor that do the work.

Hyper-Threading Technology is feasible for platforms ranging from mobile processors to servers. Its introduction into market segments other than servers is gated only by the availability and prevalence of threaded applications and workloads in these markets.

Although existing operating systems and application codes will run correctly on a processor with Hyper-Threading Technology, some relatively simple code practices are recommended to get the optimum benefit from Hyper-Threading Technology.

Hyper-Threading Technology does not deliver multiprocessor scaling. Typically, applications make use of about 35 percent of the internal processor execution resources. The idea behind Hyper-Threading Technology is to enable better processor usage and to achieve about 50 percent utilization of resources.


CPU
On-Die Cache
Architecture State
Architecture State
Adv. Programmable Interrupt Contol
Adv. Programmable Interrupt Contol
Processor Execution Resource
System Bus

A processor with Hyper-Threading Technology may provide a performance gain of 30 percent when executing multi-threaded operating system and application code over that of a comparable Intel architecture processor without Hyper-Threading Technology. When placed in a multiprocessor-based system, the increase in computing power generally scales linearly as the number of physical processors in a system is increased; although as in any multiprocessor system, the scalability of performance is highly dependent on the nature of the application.

Figure 5 Architecture of processor with Hyper-Threading Technology.

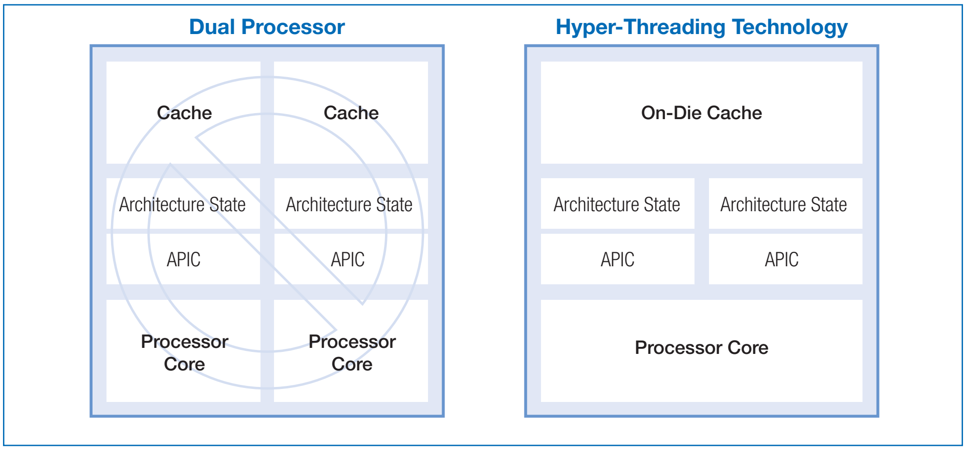


Figure 6 Two logical processors do not provide the same level of performance as a dual processor-based system.

Each logical processor

- Has its own architecture state
- Executes its own code stream concurrently
- Can be interrupted and halted independently

The two logical processors share the same

- Execution engine and the caches
- Firmware and system bus interface

Virtually all contemporary operating systems (including Microsoft Windows and Linux*) divide their workload up into processes and threads that can be independently scheduled and dispatched to run on a processor. The same division of workload can be found in many high-performance applications such as database engines, scientific computation programs, engineering-workstation tools and multimedia programs. To gain access to increased processing power, most contemporary operating systems and applications are also designed to execute in DP or multiprocessor (MP) environments, where, through the use of SMP processes, threads can be dispatched to run on a pool of processors.

Hyper-Threading Technology uses the process- and thread-level parallelism found in contemporary operating systems and high-performance applications by implementing two logical processors on a single chip. This configuration allows a thread to be executed on each logical processor. Instructions from both threads are simultaneously dispatched for execution by the processor core. The processor core executes these two threads concurrently, using out-of-order instruction scheduling to keep as many of its execution units as possible busy during each clock cycle.

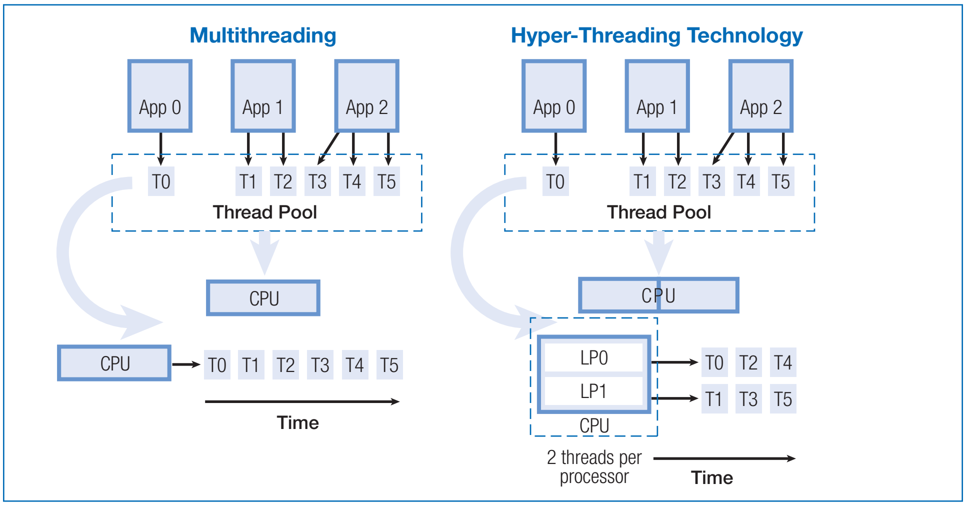


Figure 7 The time taken to process *n* threads on a single processor is significantly more than a single-processor system with Hyper-Threading Technology enabled. This is because with Hyper-Threading Technology enabled, there are two logical processors for one physical processor processing two threads concurrently.

Keys to Hyper-Threading Technology Performance

Understand and Have Clear Performance Expectations

All applications have bottlenecks – places where the application slows down, either waiting for something to happen before it can proceed, or processing commands extraneous to the task at hand. Eliminating bottlenecks in the application is the goal of performance tuning. Removing all bottlenecks in an application is usually impossible, but minimizing bottlenecks often provides significant performance benefits.

Bottlenecks can be discovered using profiling tools such as the VTune™ Performance Analyzer. A profiling tool keeps track of where an application spends its time during execution, giving a snapshot of portions of code that take a lot of time to execute.

Bottleneck	The application is
File and network I/O	Waiting to read or write to the network or disk
Processor	Waiting for the processor to become available
Memory	Busy allocating or swapping memory
Exceptions	Busy processing exceptions
Synchronization	Waiting for a shared resource to become available
Database	Waiting for a response or processing the results from a database query

Table 1 Common application bottlenecks.

Understand Hyper-Threading Technology Processor Resources

Each logical processor maintains a complete set of the architecture state. The architecture state consists of registers including the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine-state registers. From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors to store the architecture state is an extremely small fraction of the total. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic and buses.

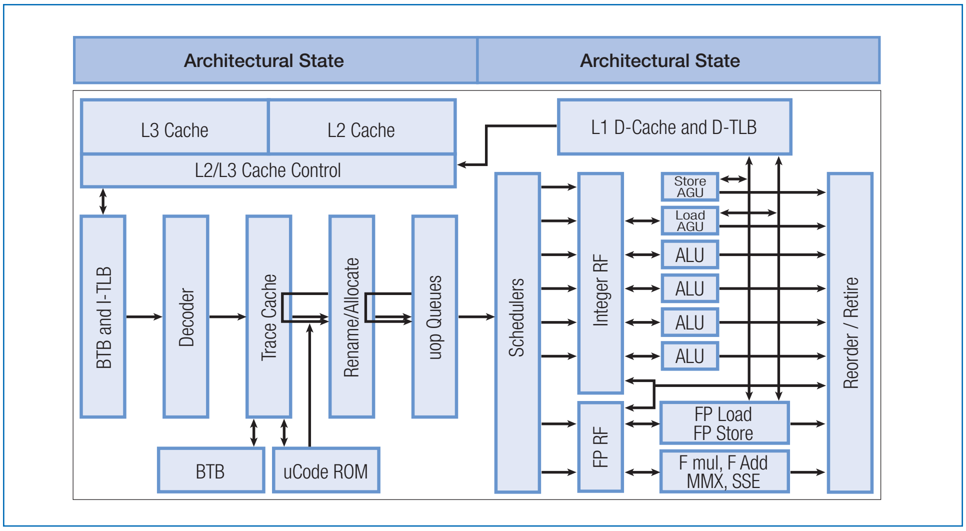


Figure 8 Hyper-Threading Technology resources.

Maximize Parallel Activity

When an application employs multithreading to exploit task-level parallelism in a workload, the control flow of the multithreaded software can be divided into two parts: parallel tasks and sequential tasks.

Amdahl's law describes an application's performance gain to the degree of parallelism in the control flow. It is a useful guide for selecting the code modules, functions or instruction sequences that are likely to realize the most gains from transforming sequential tasks and control flows into parallel code to take advantage of MP systems and Hyper-Threading Technology.

Figure 9 illustrates how performance gains can be realized for any workload according to Amdahl's law. The bar in Figure 9 represents an individual task unit or the collective workload of an entire application. In general, the speed-up of running multiple threads on MP systems with *N* physical processors (not logical processors), over single-threaded execution, can be expressed as

$$\textit{Relative Response} = \frac{\text{Tsequential}}{\text{Tparallel}} = \left(\frac{1}{1-P} + \frac{P}{N} + O\right)$$

where *P* is the fraction of workload that can be parallelized, and *O* represents the overhead of multithreading and may vary between different operating systems. The performance gain is the inverse of the relative response, in this case.

Tsequential **=** *the total time to complete work (1-P)+P*

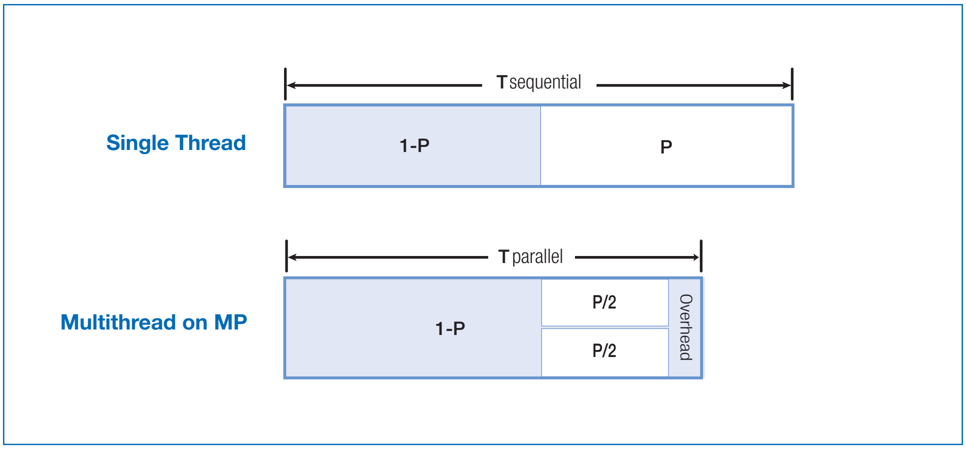


Figure 9 Amdahl's Law and MP speed-up.

When optimizing application performance in a multithreaded environment, control-flow parallelism is likely to have the largest impact on performance scaling with respect to the number of physical processors and to the number of logical processors per physical processor.

If the control flow of a multithreaded application contains a workload in which only 50 percent can be executed in parallel, the maximum performance gain using two physical processors is only 33 percent over the gain using a single processor. Four parallel processors could deliver no more than a 60 percent speed-up over a single processor. Thus, it is critical to maximize the portion of control flow that can take advantage of parallelism. Improper implementation of thread synchronization can significantly increase the proportion of serial control flow and further reduce the application's performance scaling.

In addition to maximizing the parallelism of control flows, multithreaded applications should ensure each thread has good frequency scaling. In the first implementation of the Hyper-Threading Technology execution environment by Intel, one common cause of poor performance scaling includes excessive cache misses. Excessive cache misses can occur due to:

- Aliased stack accesses by different threads in the same process
- Thread contention resulting in cache-line evictions
- False sharing of cache lines between different processors

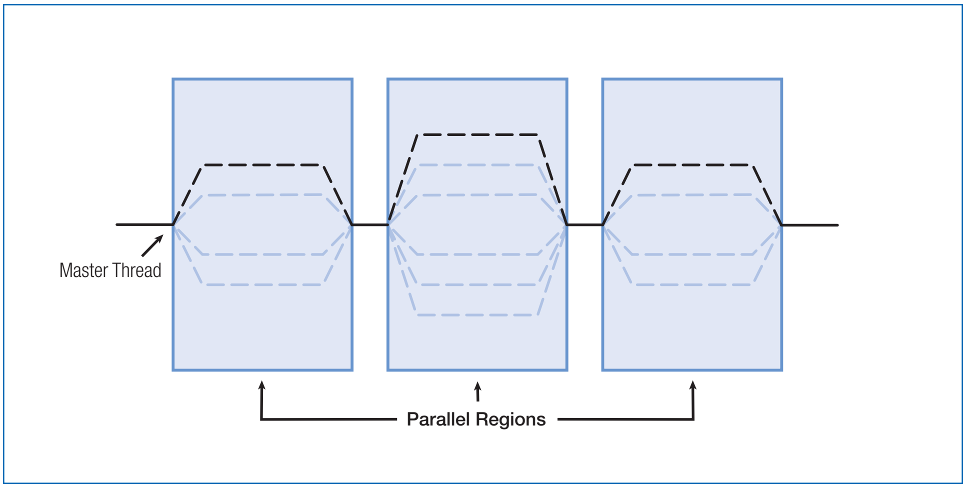


Figure 10 Fork-join control-flow paradigm. The master thread spawns additional threads as needed. This process allows parallelism to be added to a program incrementally.

Code-tuning techniques to address each of these situations and other important areas are described later in this chapter. Some examples are:

- Create a thread pool and re-use threads.
  - For data-domain decomposition, use as many threads as processors.
- Minimize operating system calls.
  - May cause implicit synchronization.
  - May cause premature context switch.
  - Copy thread invariant results.
- Avoid data sharing between threads.
  - Cache invariant results locally to the thread.
  - Avoid false cache-line sharing.
- Hyper-Threading Technology shares or splits physical processor resources, such as memory bandwidth, cache, execution units, and so on.
  - Improve use of under-utilized resources for performance gains.
  - Try not to over-use shared resources. This can diminish concurrency.

Best Practices for Optimizing Multitasking Performance

In addition to the information in this manual, refer to the application note *Threading Methodology: Principles and Practices*.

Identifying Hyper-Threading Technology Performance Bottlenecks in an Application

The following sections describe important practices, tools, coding rules and recommendations that will aid in optimizing application performance on Intel processors.

Logical vs. Physical Processors

Programmers need to know which logical processors share the same physical processor for the purposes of load balancing and application licensing strategy. The following sections tell how to:

- Detect a Hyper-Threading Technology-enabled processor.
- Identify the number of logical processors per physical processor package.
- Associate logical processors with the individual physical processors.

Note that all physical processors present on the platform must support the same number of logical processors.

The `cpuid` instruction is used to perform these tasks. It is not necessary to make a separate call to the `cpuid` instruction for each task.

Validate "Genuine Intel® Processor" with Hyper-Threading Technology

The presence of Hyper-Threading Technology in 32-bit Intel architecture processors can be detected by reading the `cpuid` feature flag bit 28 (in the edx register). A return value of 1 in bit 28 and greater than one logical processor per package indicates that Hyper-Threading Technology is present in the processor. The application must also check how many logical processors are provided under the operating system by making the appropriate operating system calls. See the application notes *Intel Processor Identification and the CPUID Instruction* and *Detecting Support for Hyper-Threading Technology Enabled Processors* for more information.

Query number of logical processors

The `cpuid` instruction is used to determine the number of logical processors in a single processor system and to determine the mapping of logical processors in a multiprocessor system. For further information about CPU counting, http://www.intel.com/cd/ids/developer/asmo-na/eng/20417.htm

Associate logical to physical processors using APIC IDs

Each logical processor has a unique APIC identification (ID). The APIC ID is initially assigned by the hardware at system reset and can be reprogrammed later by the BIOS or the operating system. The `cpuid` instruction also provides the initial APIC ID for a logical processor prior to any changes by the BIOS or operating system.

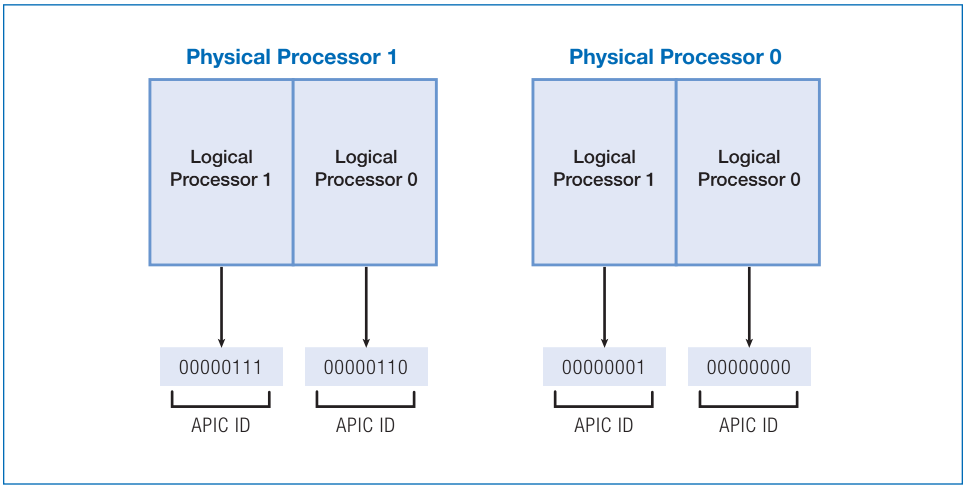
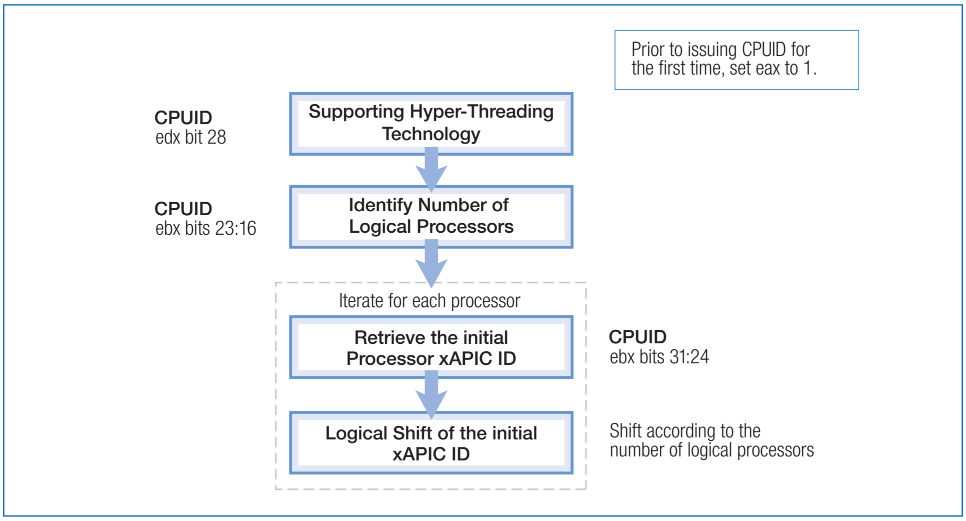


Figure 11 This is an example of the APIC ID numbers using a Hyper-Threading Technology-capable version of the Intel® Xeon™ processor MP in a dual-processor system configuration. The lowest order bit of the APIC ID distinguishes between the two logical processors. Keep in mind that these numbers are the initial values of the APIC ID at power-on reset, which can be subsequently changed by software.



Prior to issuing CPUID for the first time, set eax to 1.
CPUID edx bit 28
Supporting Hyper-Threading Technology
CPUID ebx bits 23:16
Identify Number of Logical Processors
Iterate for each processor
Retrieve the initial Processor xAPIC ID
CPUID ebx bits 31:24
Logical Shift of the initial xAPIC ID
Shift according to the number of logical processors

Figure 12 Processor-affinity algorithm.

The initial APIC ID is composed of the physical processor's ID and the logical processor's ID within the physical processor.

- The least significant bits of the APIC ID are used to identify the logical processor within a given physical processor. The number of logical processors per physical processor package determines the number of least significant bits needed.
- The most significant bits identify the physical processor ID.

Note that APIC ID numbers are not necessarily consecutive numbers starting from 0.

In addition to non-consecutive initial APIC ID numbers, the operating-system processor ID numbers are also not guaranteed to be consecutive in value.

Initial APIC ID helps software sort out the relationship between logical processors and physical processors.

Gather Initial Performance Data

Before attempting to optimize the behavior of an application, gather performance information regarding the application across platforms and configurations. What results from determining how the application currently performs is a baseline that can be used for comparison as the application's performance is tuned. The performance measurements to take depend on the application you are testing. The suggested minimum set is response time and transactions per unit of time. When gathering performance data, it's important to make sure that the same amount of work is done in the application with all the results. Recording the processor and memory utilization on the server during the tests is useful in predicting the application's scalability.

In general, an application spends 80 percent of its time executing 20 percent of the code. You need to identify and isolate that 20 percent to make changes that will impact performance. You can use the VTune Performance Analyzer to find the sections of code that occupy most of the computation time.

Implement a Single Change and Benchmark Process

As you begin optimizing an application's performance, implement changes one at a time and, after making each change, retest it using the VTune Performance Analyzer to verify that it actually improved performance. If multiple changes are made at once, it is difficult to determine which changes improved performance and which didn't. Always verify that the performance improvement didn't introduce any other problems into the application.

Continue this step until the bottlenecks have been addressed. Performance tuning is an incremental, iterative process. Plan to continue measuring and profiling the application until performance requirements are met.

General Performance Issues

Before attempting to optimize an application for Hyper-Threading Technology, use available resources to optimize for an Intel® Pentium® 4 processor-based system or Intel Xeon processor-based system including branch prediction, memory access, floating point performance, instruction selection, instruction scheduling and vectorization. For details and examples, see the *Intel® Pentium® 4 and Intel® Xeon™ Processor Optimization Guide*, **http://developer.intel.com/design/pentium4/manuals**

Identify Performance Discrepancies

To identify performance discrepancies and possible areas for optimizations, examine these program characteristics:

- Examine the number of instructions retired. Instructions retired provide a measure of how much work is being done to multithread an application if the amount of work is held constant across the multithreaded and single-threaded binary executable.
- Identify single-threaded modules and functions.
- Identify multithreaded modules and functions. By examining both single-threaded and multithreaded modules and functions and their relative performance, potential performance improvement areas can be identified.
- Identify performance discrepancies between single-threaded and multithreading overhead.

Dealing with Multithreading Code Pitfalls

This section summarizes the optimization guideline for tuning multithreaded applications. The optimization guideline covers five specific areas (arranged in order of importance):

- Thread synchronization
- Bus optimization
- Memory optimization
- Front-end optimization
- Execution-resource optimization

The key practices associated with each area are listed in this section. The guidelines for each area are discussed in greater detail in separate sections following this one. Most of the coding recommendations improve performance scaling with the number of physical processors and scaling due to Hyper-Threading Technology. Techniques that apply to only one or the other are specifically noted.

Key practices of thread synchronization

Key practices for minimizing the cost of thread synchronization are summarized below:

- Insert the `pause` instruction in fast spin loops and keep the number of loop repetitions to a minimum to improve overall system performance.
- Replace a spin lock that may be acquired by multiple threads with pipelined locks so that no more than two threads have write accesses to one lock. If only one thread needs to write to a variable shared by two threads, there is no need to acquire a lock.
- Use a thread-blocking API in a long idle loop to free up the processor.
- Prevent false sharing of per-thread data between two threads.
- Place each synchronization variable alone, separated by a cache line (128 bytes for Intel Pentium 4 processors).
- Always regression test an application with the number of threads limited to one, so that its performance can be assessed in situations where multiple threads are not available or do not work.

Key practices of system-bus optimization

Managing bus traffic can significantly impact the overall performance of multi-threaded software and MP systems. Key practices of system-bus optimization for achieving high data throughput and quick response are:

- Improve data and code locality to conserve bus-command bandwidth.
- Avoid excessive use of software prefetch instructions and allow the automatic hardware prefetcher to work. Excessive use of software prefetches can significantly and unnecessarily increase bus utilization if used inappropriately.
- Consider using overlapping multiple back-to-back memory reads to improve effective cache-miss latencies.
- Use full write transactions to achieve higher data throughput.

Key practices of memory optimization

Key practices for optimizing memory operations are summarized below:

- Use cache blocking to improve locality of data access. Target one-quarter to one-half of the cache size when targeting Intel architecture 32-bit processors with Hyper-Threading Technology.
- Minimize the sharing of data between threads that execute on different physical processors sharing a common bus.
- Minimize data-access patterns that are offset by multiples of 64KB in each thread.
- Adjust the private stack of each thread in an application so the spacing between these stacks is not offset by multiples of 64KB or 1MB to prevent unnecessary cache-line evictions, when targeting Intel architecture 32-bit processors with Hyper-Threading Technology.

- When targeting Intel architecture 32-bit processors with Hyper-Threading Technology, add a per-instance stack offset when two instances of the same application are executing in lock steps to avoid memory accesses that are offset by multiples of 64KB or 1MB.
- Evenly balance workloads between processors, physical or logical. Load imbalance occurs when one or more processors sit idle waiting for other processors to finish. Load imbalance issues can be as simple as one thread completing its allocated work before the others. Resolving imbalance issues typically requires splitting the work into smaller units that can be more evenly distributed across available resources.

Key practices of front-end optimization

Key practices for front-end optimization are:

- Avoid excessive loop unrolling to ensure the trace cache is operating efficiently.
- Optimize code size to improve locality of trace cache and increase delivered trace length.

Key practices of execution-resource optimization

Each physical processor has dedicated execution resources, and the logical processors in each physical processor that supports Hyper-Threading Technology share on-chip execution resources. Key practices for execution-resource optimization include:

- Optimize each thread to achieve optimal frequency scaling first.
- Optimize multithreaded applications to achieve optimal scaling with respect to the number of physical processors.
- To ensure compatibility with future processor implementations, do not hard code the value of cache sizes or cache lines into an application; instead, always query the processor to determine the sizes of the shared cache resources.
- Use on-chip execution resources cooperatively if two threads are sharing the execution resources in the same physical processor package.
- For each processor with Hyper-Threading Technology, consider adding functionally uncorrelated threads to increase the hardware-resource utilization of each physical processor package.

Optimization Techniques

Typically, a given application only needs to apply a few optimization techniques in selected areas to combine multiple scaling factors (frequency, number of physical processors and Hyper-Threading Technology). The following section describes some typical performance bottlenecks and provides guidelines for optimizing applications that encounter these problems.

Eliminate or reduce the impact of spin-wait loops

The frequency and duration with which a thread needs to synchronize with other threads depends on the characteristics of an application. When a synchronization loop needs a very fast response, an application may use a spin-wait loop.

A spin-wait loop is typically used when one thread needs to wait for a short amount of time for another thread to reach a point of synchronization. The basic structure of a spin-wait loop consists of a loop that compares a synchronization variable with some pre-defined value.

Spin-wait loops are used:

- For synchronization
- To avoid overhead of operating system calls or context-switches
- When relatively short durations are expected
- To have limited or no impact on other physical processors

Conversely, when a worker thread is expected to remain busy for an extended period (for example, longer than the operating-system time quanta for task switching), a different coding technique is needed to implement the synchronization between the worker threads and the control thread.

Spin-wait loops should be used sparingly, if at all, in hyper-threaded applications because:

- They create very high throughput loops.
- They consume split or shared resources without producing useful work. Spin-wait loops running on one logical processor consume the shared processor resources of the physical processor, and so cause execution of other threads to slow down.
- They can also cause memory-order conflicts.

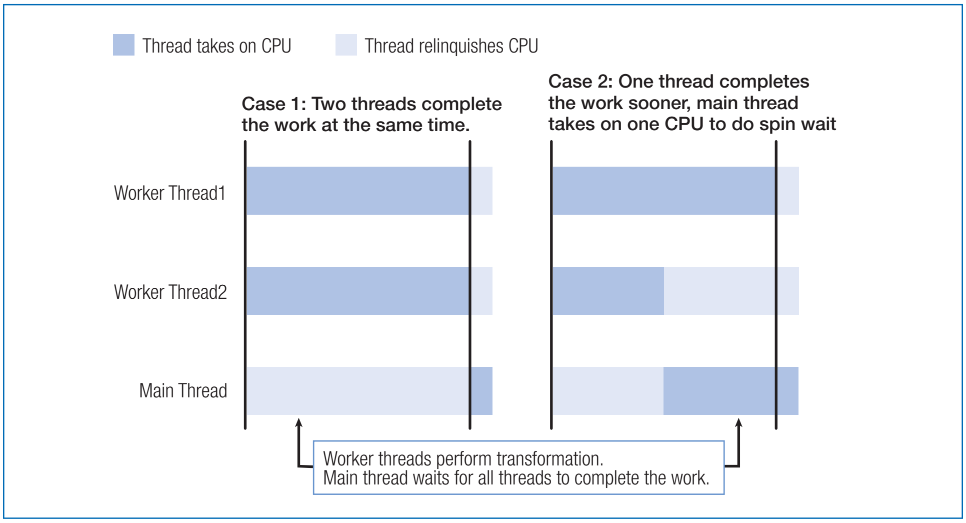

Thread takes on CPU
Thread relinquishes CPU
Case 1: Two threads complete the work at the same time.
Case 2: One thread completes the work sooner, main thread takes on one CPU to do spin wait
Worker Thread1
Worker Thread2
Main Thread
Worker threads perform transformation.
Main thread waits for all threads to complete the work.

Figure 13 Example of overhead created due to the use of spin-wait loops in a multithreaded application.

To avoid these performance problems, use Hyper-Threading Technology-aware operating-system synchronization constructs.

- For Win32 applications use the synchronization function `WaitForMultipleObjects()`. The `WaitForMultipleObjects()` processor usage of main thread reduces execution time significantly.
- Insert a new `pause` instruction if a spin wait cannot be avoided. On 32-bit Intel architecture-based processor generations earlier than the Pentium 4 processor, a `pause` instruction is treated as a `nop` instruction.

Avoiding 64K aliasing in the first level data cache

A 64-byte or greater data structure or array should be aligned so that its base address is a multiple of 64. Sorting data in decreasing size order is one heuristic for assisting with natural alignment. As long as 16-byte boundaries (and cache lines) are never crossed, natural alignment is not strictly necessary, though it is an easy way to enforce this.

```
and edi, 0xffff0000

mov ebx, 0x 10000

mov [edi], eax

mov [edi + ebx +32], eax
```

Figure 14 Two stores that alias the second store do so because it lies within a cache boundary that is modulo 64k.

Two memory references that access linear address ranges whose cache line boundaries are offset by multiples of 64K bytes. This condition affects both single-threaded and multi-threaded applications.

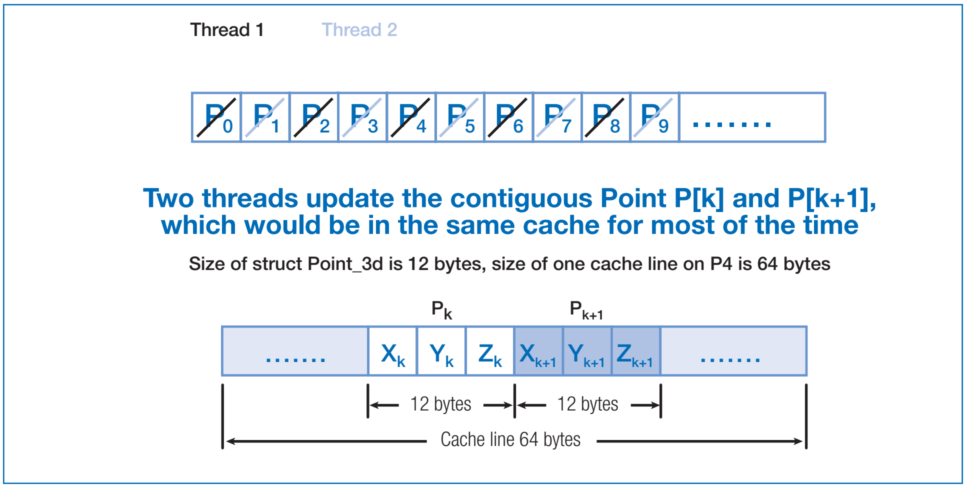

Thread 1
Thread 2
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 ....
Two threads update the contiguous Point P[k] and P[k+1], which would be in the same cache for most of the time
Size of struct Point_3d is 12 bytes, size of one cache line on P4 is 64 bytes
Pk
Pk+1
.... Xk Yk Zk Xk+1 Yk+1 Zk+1 ....
12 bytes
12 bytes
Cache line 64 bytes

Figure 15 Example showing false sharing. Thread 1 (in black) and thread 2 (in blue) divide the work by taking on the elements of one array alternately.

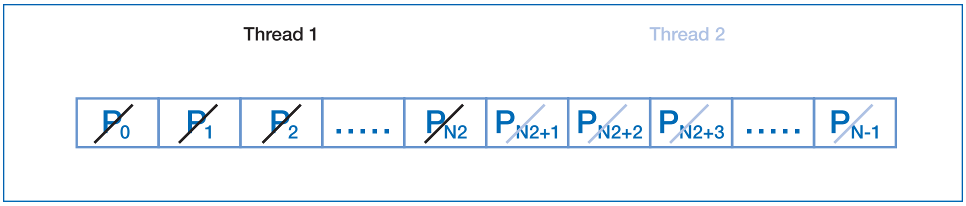


Figure 16 Two threads divide the array so that thread 1 takes the first half and thread 2 takes the second.

Some operating systems create threads with stack spaces aligned on 1MB boundaries, which is modulo 64K. Two threads executing simultaneously are likely to have many stack variables located at address ranges that meet the 64K aliasing condition. Access to any variable declared in the stack may therefore use 64K aliasing.

Solve this problem by assigning a stack offset value, which will vary each stack offset for each thread and thereby avoid 64K aliasing conflicts on the stack.

If you suspect that 64K aliasing is a problem, use the VTune Performance Analyzer to monitor 64K aliasing events. This allows you to find 64K aliasing problems associated with thread stacks and in other places in the code. Assign a stack offset value and recompile, then retest using the VTune Performance Analyzer to ensure that the problem has been addressed. To assign a stack offset value, allocate memory from the program stack for each thread. For example `void * pOff=_alloca(size);` with a different `size` for each thread will assign a stack offset value of `size` bytes. Manipulating the stack offset can assist with reducing the number 64K aliasing events.

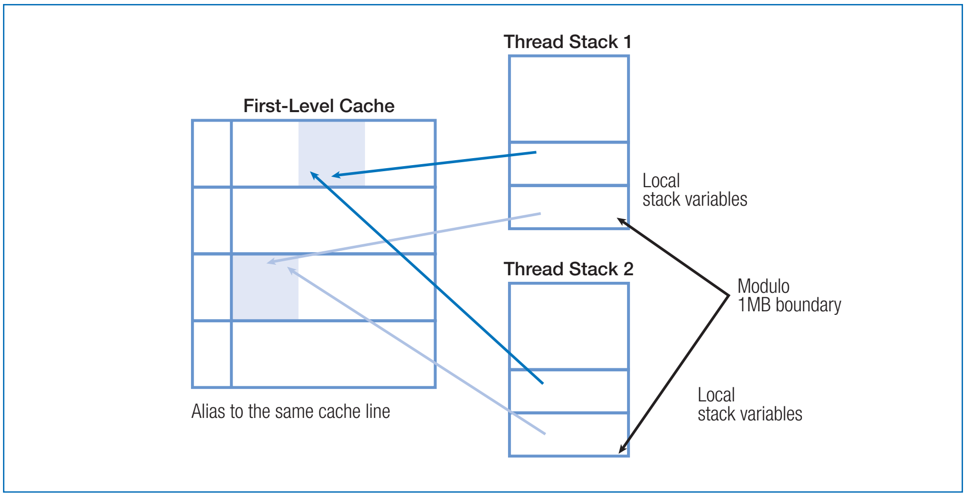


Figure 17 Effect of stack alignment on first-level cache aliasing.

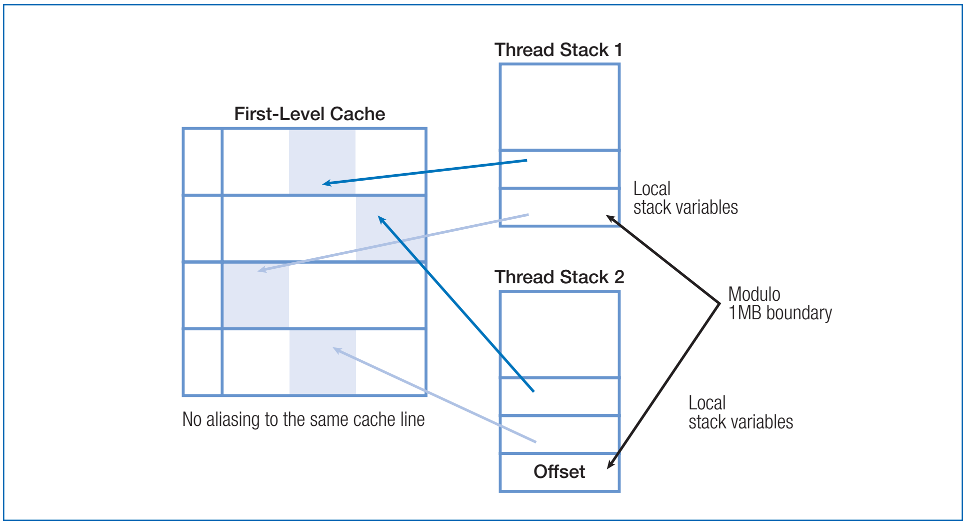


Figure 18 Adding a stack offset eliminates aliasing to the same cache line.

Balance the impact of background task priorities on physical processors

Background tasks are used for:

- Accomplishing low priority tasks
- Running when available resources allow
- Making little impact on the user experience
- Making limited or no impact on other physical processors

Background tasks can consume resources needed by higher priority tasks. To avoid this, adjust the frequency and duration of background tasks based on physical processors.

Avoid serializing events and instructions

Processor serial events are used to:

- Ensure coherency and consistency
- Handle unusual events

Serializing events cause:

- Pipelines and caches to be emptied
- Significant performance impact
- All logical processors to be affected

To optimize performance of a multithreaded application, avoid processor serializing events, such as:

- Floating-point denormals
- Memory-order conflicts
- Self-modifying code (SMC)

Optimize cache sharing

The use of caches and shared cache data is key to building high performance multi-threaded applications. The basic unit of sharing between processes is a cache line, as opposed to data variables. Processors with Hyper-Threading Technology enabled use shared cache space, and as a result, how the cache space is used has a great impact on the performance of a hyper-threaded application.

Overcoming false sharing in data cache

When two threads must share data, it is important to avoid what is commonly called false sharing. False sharing applies to data used by one thread that happens to reside on the same cache line as different data used by another thread. In some cases, one part of a cache line can be written while at the same time, a different part of the cache line is being read by another thread.

An example of false sharing is when thread-private data and a thread synchronization variable are located within the line-size boundary (64 bytes for write, 128 bytes for read). When one thread modifies the synchronization variable, the “dirty” cache line must be written out to memory and updated to each physical processor sharing the bus.

Subsequently, data is fetched into each target processor 128 bytes at a time, causing previously cached data to be evicted from its cache on each target processor.

False-sharing incurs a performance penalty, when two threads run on different physical processors or on two logical processors in the physical processor package. In the first case, the performance penalty is due to cache evictions to maintain cache coherency. In the latter case, performance penalty is due to memory-order machine-clear conditions.

When a common block of parameters is passed from a parent thread to several worker threads, it is desirable for each worker thread to create a private copy of frequently accessed data in the parameter block.

To solve false sharing problems, follow these guidelines:

- Use the VTune Performance Analyzer to monitor "machine clear caused by other thread." Avoid accessing data on the same cache line with multiple threads by partitioning data.
- Make copies of the structure for each thread to eliminate false sharing.
- Pad structures so that they are twice the size of a read cache line (128 bytes for Intel Pentium 4 processor).

Synchronization overhead greater than parallel region

When threads access shared resources, they use signals (called semaphores) to ensure that the data is not corrupted. Problems occur when the overhead associated with using a synchronization mechanism (semaphores, locks or other methods) takes a significant amount of time compared to the time it takes the parallel sections of code take to execute. When this is the case, there is no point in threading that section of the application since the synchronization mechanism takes too much time. When this problem is identified, either reduce the synchronization time or eliminate the thread.

Take advantage of write combining buffers

Intel® NetBurst™ microarchitecture supports six write combining store buffers, each buffering one cache line. Write combining provides a fast and efficient method of data transfer between the store buffers used when write instructions are executed to the first and second level caches.

When optimizing single-threaded inner-loops for the NetBurst microarchitecture, it is recommended that the inner loop perform no more than four cache-line writes per iteration. By doing so, the inner loop benefits from optimal usage of the write combining buffers. When two or more threads are used, it is best to restrict the inner loop to only two cache-line writes per inner loop iteration to obtain the benefit of the write combining buffers.

One method for taking advantage of the writing combining performance benefits is to split the inner loop into multiple loops. The example following shows a code segment that initially does not benefit from the write combining buffers if two threads are used. After the inner loop in split up into two loops, however, the benefits of using the write combining buffers can be obtained.

```
for() {

   for( I )  {

              pA[I] = data1;

              pB[I] = data2;

              pC[I] = data3;

              pD[I] = data4;

 }

}
```

Figure 19 Before optimizing.

```
for() {

   for( I )  {

              pA[I] = data1;

              pB[I] = data2;

 }

 for( I )  {

              pC[I] = data3;

              pD[I] = data4;

 }
```

Figure 20 After optimizing to take advantage of write combining buffers.

Correct load imbalance

Load imbalance occurs when one or more processors, logical or physical, sit idle waiting for others processors to finish work. Load imbalance can be the result of a number of underlying causes, but most of the time, it is as simple as one thread finishing its allocated work before the other. Load imbalance usually gets worse as the number of processors increase, because as a general rule, it becomes progressively more difficult to evenly split workloads into smaller and smaller units of work. Improving balance often means rethinking the design and redesigning the work so that it can be divided into more evenly distributed workloads.

Reducing excessive idle time generally requires recoding. Sometimes calling a different operating system function, such as `PostMessage` instead of `SendMessage`, or `EnterCriticalSection` instead of `WaitForSingleObject`, can reduce idle time sufficiently to avoid the need for more extensive recoding. Other times, redesign is the only option.

Hyper-Threading Technology Application Development Resources

Intel offers several tools that can help you optimize your application's performance.

Intel® C++ Compiler

Programming directly to a multithreading application-programming interface (API) is not the only method for creating multithreaded applications. New tools such as the Intel® C++ Compiler with OpenMP* support has become available with capabilities that make the challenge of creating multithreaded applications much easier.

Two features available in the latest Intel C++ Compilers are:

- Generating multithreaded code using OpenMP directives
- Generating multithreaded code automatically from unmodified high-level code

Use the Intel C++ Compiler following the recommendations described here wherever possible. The Intel C++ Compiler's advanced optimization features provide good performance without the need to hand-tune assembly code. The following features may enhance performance even further:

- Inlined assembly
- Intrinsics, which have a one-to-one correspondence with assembly language instructions, but allow the compiler to perform register allocation and instruction scheduling so the user does not need to do this. (Refer to the "Intel® C++ Intrinsics Reference" section of the *Intel® C++ Compiler User's Guide*.)
- C++ class libraries. (Refer to the "Intel® C++ Class Libraries for SIMD Operations Reference" section of the *Intel® C++ Compiler User's Guide*.)
- Vectorization, in conjunction with compiler directives (pragmas). (Refer to the "Compiler Vectorization Support and Guidelines" section of the *Intel® C++ Compiler User's Guide*.)

The Intel C++ Compiler can generate a single executable that uses features such as SSE2 to maximize performance on a Pentium 4 processor, but which still executes correctly on older processors without such features. (See the "Processor Dispatch Support" section in the *Intel® C++ Compiler User's Guide*.)

General compiler recommendations

Any compiler that has been extensively tuned for the Pentium 4 processor can be expected to match or outperform hand coding, in general. If particular performance problems are noted with the compiled code, however, some compilers (such as the Intel C++ and Fortran Compilers) allow the coder to insert intrinsics or inline assembly, to exert greater control over what code is generated. If inlined assembly is used, the user should verify that the code generated to integrate the inline assembly is of good quality and yields good overall performance.

Default compiler switches are generally targeted in most cases. That is, an optimization may be the default solution if it is beneficial for most programs. In the unlikely event that a performance problem is the root-cause of a poor choice on the part of the compiler, using different switches for that compiler, or compiling that module with a different compiler may help.

Performance of compiler-generated code may vary from one compiler vendor to another. The Intel C++ Compiler and The Intel Fortran Compiler are highly optimized for the Pentium 4 processor. You may find significant performance advantages to using one of these as your back-end compiler.

VTune™ Performance Analyzer

Where performance is of critical concern, use performance-monitoring hardware and software tools to tune your application and its interaction with the hardware. The Pentium 4 processor provides counters that monitor a large number of performance-related events affecting overall performance, branch prediction, the number and type of cache misses and average trace length. The counters also provide information that helps resolve coding pitfalls.

The VTune Performance Analyzer uses these counters to provide you with two kinds of feedback:

- An indication of a performance improvement from using a specific coding recommendation or microarchitecture feature
- Information on whether a change in the program has improved or degraded performance with respect to a particular metric

The VTune Performance Analyzer contains many features that may help in determining the thread performance issues. As discussed early in this document, performance limiting issues such as load imbalance, excessive overhead, idle time, and processor architectural issues like memory aliasing can be identified using the VTune Performance Analyzer.

See the VTune Performance Analyzer online help for instructions on how to use this tool. See the application note *Optimizing for Hyper-Threading Technology Using the VTune™ Performance Analyzer* for additional information on using this tool to optimize applications targeted for Hyper-Threaded deployment.

Automatic parallelization of code

Intel C++ Compiler 7.0 supports an option `-Qparallel`, which can automatically identify certain loop structures that contain parallelism. During program compilation, the compiler automatically attempts to decompose the parallelism into threads for parallel processing. No other intervention or effort by the programmer is needed.

Thread APIs

The OpenMP API is an easy-to-use API for writing multithreaded programs. OpenMP provides a standardized, non-proprietary, portable set of FORTRAN and C++ compiler directives supporting shared memory parallelism in applications and library routines for parallel application programmers. OpenMP supports directive-based processing, which uses special preprocessors or modified compilers to interpret the parallelism expressed in FORTRAN comments or C/C++ pragmas. This makes it easier to convert serial applications into parallel applications. The benefits of directive-based processing include:

- Original source is compiled unmodified.
- Incremental code changes are possible, which preserve the algorithms of the original code and enable rapid debugging.
- Incremental code changes help programmers maintain serial consistency. When the code is run on one processor, it gives the same result as the unmodified source code.

Most of the constructs in OpenMP are compiler directives or pragmas.

For C and C++, the pragmas take the form of:

```
#pragma omp construct [clause [clause].] construct [clause [clause].]
```

For FORTRAN, the directives take one of these forms:

- `C$OMP construct [clause [clause].] construct [clause [clause].]`
- `!$OMP construct [clause [clause].] construct [clause [clause].]`
- `*$OMP construct [clause [clause].] construct [clause [clause].]`

To incorporate OpenMP into a C/C++ application, include the `omp.h` file or for a FORTRAN application include the `omp_lib.h` file and `OMP_LIB.mod` module. In addition, the compiler flag `/Qopenmp` must be used to notify that OpenMP is used within the application.

Most OpenMP constructs apply to *structured blocks*. A structured block is a block with one point of entry at the top and one point of exit at the bottom, with the only branches allowed being `STOP` statements in FORTRAN and `exit()` in C and C++.

```
#pragma omp parallel
{
   int id = omp_get_thread_num();
more: res(id) = do_big_job(id);
   if(conv(res(id))goto more;
}
  printf("All done \n");
```

```
   if(go_now())goto more;
#pragma omp parallel
{
   int id = omp_get_thread_num();
more: res(id) = do_big_job(id);
   if(conv(res(id))goto done;
   goto more;
}
done: if(!really_done())goto more;
```

Figure 21 Structured block of code on the left; invalid code on the right.

Parallel regions

You create parallel threads using the `omp parallel` pragma.

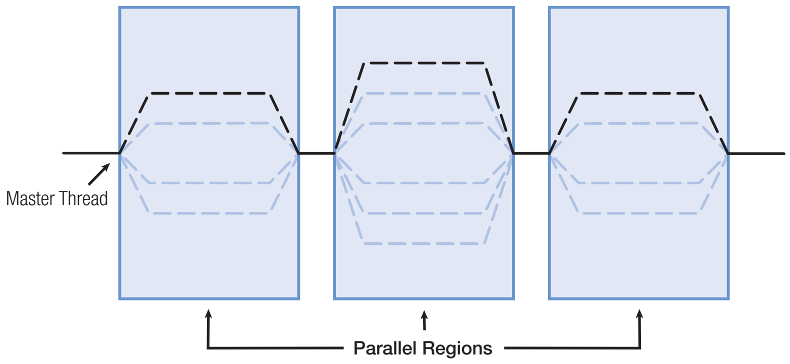


Figure 22 Parallel regions.

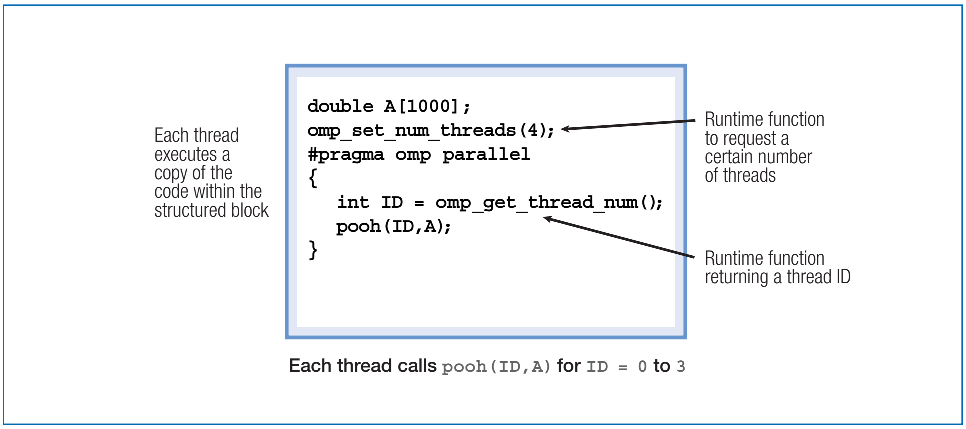
OpenMP uses a shared-memory programming model. Most variables defined prior to a parallel region are *shared*.

Global variables are shared among threads:

- In FORTRAN, these include: `COMMON` blocks, `SAVE` and `MODULE` variables
- In C, these include: File scope variables, static and heap memory (`malloc`)

But some variables are local to a thread, or *private*. These include:

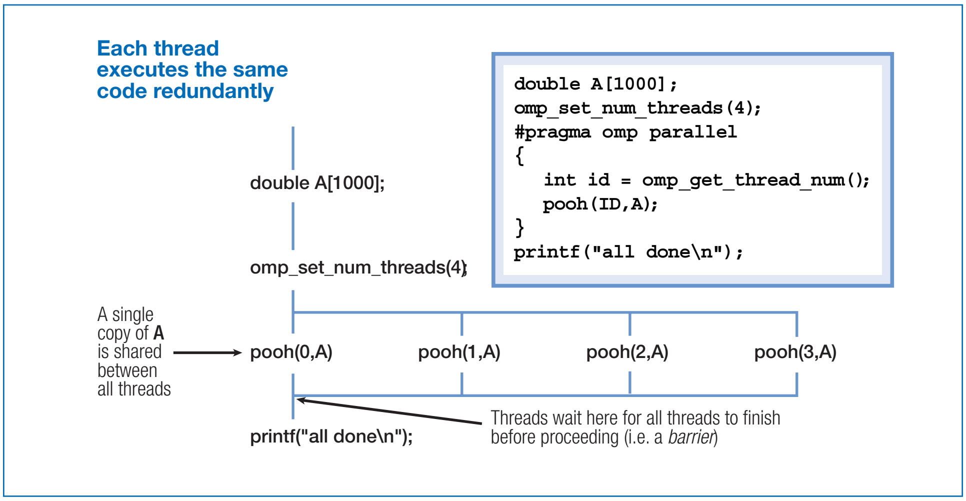
- Local variables in sub-programs called from parallel regions
- Automatic variables within a statement block



```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block
Runtime function to request a certain number of threads
Runtime function returning a thread ID
Each thread calls pooh(ID,A) for ID = 0 to 3

Figure 23 Example showing how to create a 4-thread parallel region.

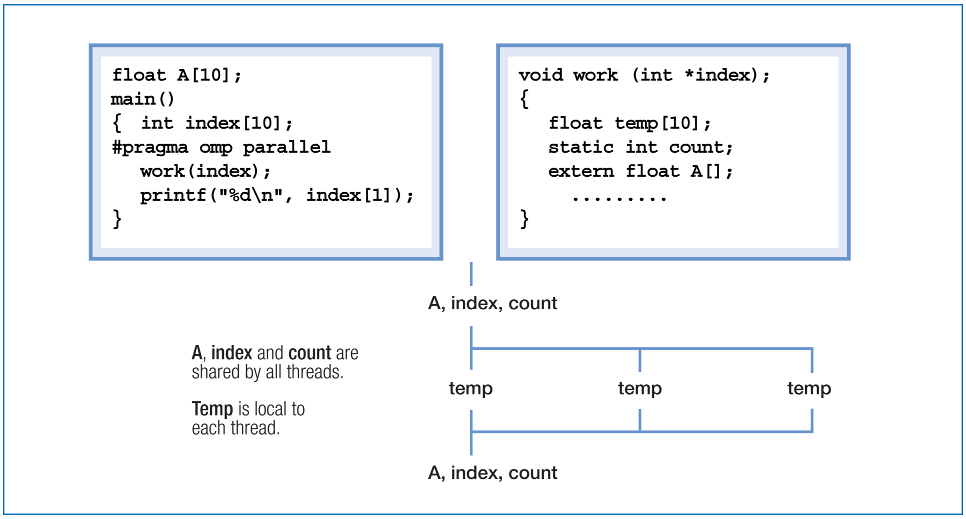


```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int id = omp_get_thread_num();
   pooh(ID,A);
}
printf("all done\n");
```

Each thread executes the same code redundantly
double A[1000];
omp_set_num_threads(4)
A single copy of A is shared between all threads
pooh(0,A)
pooh(1,A)
pooh(2,A)
pooh(3,A)
printf("all done\n");
Threads wait here for all threads to finish before proceeding (i.e. a barrier)

Figure 24 Code for a 4-thread parallel region.

Clauses on directives can declare variables to be either shared or private.



```
float A[10];
main()
{  int index[10];
#pragma omp parallel
   work(index);
   printf("%d\n", index[1]);
}
```

```
void work (int *index);
{
   float temp[10];
   static int count;
   extern float A[];
     .........
}
```

A, index, count
A, index and count are shared by all threads.
Temp is local to each thread.
temp
temp
temp
A, index, count

Figure 25 Shared data example.

Work sharing

The `for` work-sharing construct splits loop iterations among the threads in a team.

By default, there is a barrier at the end of the `omp for` construct. Use the `nowait` clause to turn off the barrier.

Using work-sharing code can simplify programming. The following examples show four sections of code. Figure 26 is the code written for sequential processing. Figure 27 shows equivalent code written for parallel processing using OpenMP, but without using work sharing.

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

Figure 26 Sequential code.

```
#pragma omp parallel

#pragma omp for

   for (i=0;i<N;i++){

       NEAT_STUFF(i);

   }
```

Figure 27 Example of the `for` work-sharing construct.

Figure 28 is showing how to break up the `for` construct into parallel regions.
Figure 29 shows the equivalent code written for parallel processing using work sharing.

```
#pragma omp parallel

{

   int id, i, Nthrds, istart, iend;

   id = omp_get_thread_num();

   Nthrds = omp_get_num_threads();

   istart = id * N / Nthrds;

   iend = (id+1) * N / Nthrds;

   for(i=istart;i<iend;i++) { a[i] = a[i] + b[i];}

}
```

Figure 28 OpenMP parallel region.

```
#pragma omp parallel

#pragma omp for

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

Figure 29 OpenMP parallel region and a work-sharing `for` construct.

Reduction

Loop iterations must be independent for work sharing to be valid.
Consider the very common pattern shown in the following example:

```
for (i+0; i< 10000; i++){

   ZZ = func(i);

   res = res + ZZ;

}
```

Figure 30 Loop-carried dependency example. The variable `res` creates a loop-carried dependency, and therefore, parallelization does not work.

The reduction clause lets you parallelize loops with dependencies similar to the one shown in the previous example using the `pragma omp for` reduction.

```
#pragma omp for reduction(+:res)

   for (i+0; i< 10000; i++){

       ZZ = func(i);

       res = res + ZZ;

   }
```

Figure 31 Example showing use of the reduction clause to create parallel loops.

Each thread gets a *private* copy of `res`. Accumulation is in the private copy. When the loop is complete, the private copies are combined into a single shared copy.

References

For more information on the Intel architecture, specific techniques and processor-architecture terminology referenced in this manual, see the following resources:

- *Multithreaded Programming in a Microsoft Win32* Environment*:
  HTML format:
  http://www.intel.com/cd/ids/developer/asmo-na/eng/20439.htm

  PDF format:
  http://www.intel.com/cd/ids/developer/asmo-na/eng/17878.htm

- *Introduction to Hyper-Threading Technology*:
  http://www.intel.com/technology/hyperthread/download/25000802.pdf
- *Multithreaded Programming for Next Generation MultiProcessing Technology*:
  http://www.intel.com/technology/hyperthread/multi_nexgen/
- *Intel® Multiprocessor Specification*:
  ftp://download.intel.com/design/pentium/datashts/24201606.pdf
- *Intel® Pentium® 4 and Intel® Xeon™ Processor Optimization Reference Manual*:
  ftp://download.intel.com/design/Pentium4/manuals/24896606.pdf
- *Intel® C++ Compiler User's Guide*
- *Intel® Fortran Compiler User's Guide*
- *VTune™ Performance Analyzer Online Help*
- *Intel® Processor Identification with the Processor ID Instruction*, document number 241618
- *Intel® Architecture Software Developer's Manual:*
  - Volume 1: *Basic Architecture*, document number 245470
  - Volume 2: *Instruction Set Reference Manual*, document number 245471
  - Volume 3: *System Programmer's Guide*, document number 245472

Also, refer to the following Application Notes:

- *Adjusting Initial Thread Stack Address to Improve Performance on Intel® Xeon™ Processor MP*
- *Hyper-Threading Technology-Enabled Processors*
- *Detecting Support for Hyper-Threading Technology-Enabled Processors*
- *Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon™ Processor MP*
- *Optimizing for Hyper-Threaded Technology Using the VTune™ Performance Analyzer*
- *Threading Methodology: Principles and Practices*

In addition, refer to the Hyper-Threading Technology publications in the following Web sites:

- http://developer.intel.com/technology/hyperthread
- http://developer.intel.com/technology/itj/2002/volume06issue01/art01_hyper/p01_abstract.htm

Glossary

APIC
An abbreviation for *advanced programmable interrupt controller*. The APIC handles interrupts sent to a specified logical processor.

Application
Refers to the primary code being characterized; a self-contained program that performs a specific function directly for the user.

Chip multiprocessing
One of these techniques is chip multiprocessing (CMP), where two or more processors are put on a single die. Each processor can have multiple sets of full execution and architectural resources. The processors may or may not share a large on-chip cache. CMP is largely orthogonal to conventional multiprocessor systems, as you can have multiple CMP processors in a multiprocessor configuration. A CMP chip is significantly larger, however, than the size of a single-core chip and therefore more expensive to manufacture; moreover, it does not begin to address the die size and power considerations..

Clockticks
Refers to the basic unit of time recognized by a processor. It can also be used to indicate the time required by the processor to execute an instruction.

Conditional variables
A simple supplement to mutexes to synchronize time access.

Context switching
Refers to a situation when the current software thread transitions out of a processor and another thread transitions into the processor.

Critical sections
These are blocks of codes that can be executed by only one thread at a time.

Decomposition
- Domain decomposition:
  Different threads for different data.
- Functional decomposition:
  Different threads for different tasks.

DP
An abbreviation for dual processor, meaning two physical processors working together in a system.

DP scaling
A quantity derived by dividing a DP system performance by a UP system performance.

DTLB misses
Refers to the number of retired `load` or `store` instructions that experienced data translation lookaside buffer (DTLB) misses.

Events
A signal used for synchronization methods to denote when a resource is active or ready.

False sharing
Refers to different processors working on different data within the same cache line.

Front end
The front end of the pipeline is responsible for delivering instructions to the later pipe stages.

FSB data ready
Counts the number of front-side bus clocks that the bus is transmitting, including full reads|writes and partial reads|writes and implicit writebacks.

H-T
An abbreviation for Hyper-Threading Technology, the multithreading design technique that allows an operating system to view a single physical processor as if it were two logical processors.

Hyper-Threading Technology
Hyper-Threading Technology makes a single physical processor appear as multiple logical processors. Through simultaneous multithreading, Hyper-Threading Technology allows a single processor to manage data as if it were two processors by handling data instructions in parallel rather than one at a time.

Hyper-Threading Technology effectiveness
A quantity derived by describing the effectiveness of Hyper-Threading Technology while taking the scalability of the workload into account.

Hyper-Threading Technology scaling
A quantity derived by dividing a Hyper-Threading Technology-enabled system performance by the performance number obtained on the Hyper-Threading Technology-disabled system.

Instruction-level parallelism (ILP)
Refers to techniques to increase the number of instructions executed each clock cycle. ILP causes the overlap of the execution of independent instructions.

ITLB misses
Instruction translation lookaside buffer (ITLB) misses are triggered by a TC miss. The ITLB receives a request from the TC to deliver new instructions and translates the 32-bit linear address to a 32-bit memory physical address before the cache lookup is performed.

Logical processor
Hyper-Threading Technology makes a single physical processor appear to the operating system as multiple logical processors. Each logical processor retains a copy of the architecture state, while sharing a single set of physical execution resources.

Memory-order machine clears
Refers to the number of times the entire pipeline of the machine is cleared due to memory ordering.

Message passing interface (MPI)
Refers to a library of routines that can be used to create parallel programs.

MP
An abbreviation for multiprocessor, meaning four or more physical processors working together in a system.

MP scaling
A quantity derived by dividing the MP system performance by the UP system performance. MP scaling has to be quantified by providing the number of processors.

Multiprocessing
Refers to a state where the system is running parallel tasks using two or more physical processors.

Multitasking
Refers to multiple applications running concurrently or simultaneously in one computer. The number of programs that can be effectively multitasked depends on the type of multitasking performed (preemptive versus cooperative), processor speed and memory and disk capacity.

Multithreading
Refers to a processing state that allows multiple streams of execution to take place concurrently within the same program, each stream processing a different transaction or message.

Mutexes
Refers to simple lock primitives that can be used to control access to shared resources.

Non-scalable workload
Specifically refers to an application workload, because the performance of an application workload does not increase on a DP or MP system compared to a UP system.

OpenMP
A particular threading model where the programmer introduces parallelism or threading by using directives or pragmas. This API makes it easier to create multithreaded programs in FORTRAN, C and C++.

Physical processor
Refers to the actual processor die that when Hyper-Threading Technology is added, includes two logical processors. With first implementation of Hyper-Threading Technology, there are two logical processors per physical processor.

Process
An instance of running a program with the attendant states needed to keep it running.

Scalable workload
Refers to a performance increase on a DP or MP system compared to a UP system. The workload scalability, however, is determined by the relative performance difference between a DP/MP system and a UP system.

Semaphores
Refers to a counting primitive that allows access to shared data between threads.

Simultaneous multithreading
Finally, there is simultaneous multithreading, where multiple threads can execute on a single processor without switching. The threads execute simultaneously and make much better use of the resources. This approach makes the most effective use of processor resources: it maximizes the performance versus transistor count and power consumption.

Spin-locks
Blocking method used to ensure data integrity when multiple processors access it.

Switch-on-event multithreading
Switch-on-event multithreading would switch threads on long latency events such as cache misses. This approach can work well for server applications that have large numbers of cache misses and where the two threads are executing similar tasks. Neither the time-slice nor the switch-on-event multithreading techniques, however, achieve optimal overlap of many sources of inefficient resource usage, such as branch mispredictions, instruction dependencies and so on.

Symmetric multiprocessors (SMP)
Processors that have a symmetric relationship to a main memory and a uniform access time from each processor.

TC deliver mode
Execution trace cache (TC) counts the number of cycles that the trace cache delivers instructions from the trace cache, as opposed to decoding and building traces.

TC misses
TC misses count the times an instruction needed wasn't available in the trace cache.

Threading
Refers to a method of writing a program that divides it into multiple tasks. Each task may handle a different function (I/O, GUI, etc.). A particularly time-intensive function can be divided so that two or more threads cooperate to complete the overall task.

Thread-level parallelism (TLP)
Refers to a state whereby multiple threads or processes can be executed in parallel, such as the sharing of code and address space during the execution of a process.

Time-slice multithreading

Another approach to exploiting TLP is to allow a single processor to execute multiple threads by switching between them. Time-slice multithreading is where the processor switches between software threads after a fixed time period. Time-slice multithreading can result in wasted execution slots but can effectively minimize the effects of long latencies to memory[5].

µops retired

Counts the number of µops (also known as micro-operations) retired. Each instruction is made up of one or more micro-operations. This number count doesn't include false micro-operations, however, because they are typically in a mis-predicted branch path and are not retired.

UP

An abbreviation for the term uni-processor, meaning one physical processor existing in a system.

VTA

VTune Performance Analyzer threading analysis. A series of tools or VTune Performance Analyzer views that will help in analyzing threaded applications running on typical UP, DP or MP systems with or without Hyper-Threading Technology.

Workload

Refers to the constant work being done by an application – the output must be repeatable.

[5] Microsoft Windows implements time slices as quanta.

THIS DOCUMENT AND RELATED MATERIALS AND INFORMATION ARE PROVIDED "AS IS" WITH NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE. INTEL ASSUMES NO RESPONSIBILITY FOR ANY ERRORS CONTAINED IN THIS DOCUMENT AND HAS NO LIABILITIES OR OBLIGATIONS FOR ANY DAMAGES ARISING FROM OR IN CONNECTION WITH THE USE OF THIS DOCUMENT. Intel® products are not intended for use in medical, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's web site at http://www.intel.com

Intel, the Intel logo, Pentium, Intel Xeon, i486, Intel486, VTune and Intel NetBurst are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2003, Intel Corporation

Intel, the Intel logo, Pentium, Intel Xeon, i486, Intel486, VTune and Intel NetBurst are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
*Other names and brands may be claimed as the property of others.
Copyright © 2003 Intel Corporation. All rights reserved. 0103/CS/HB/PDF C14287-002