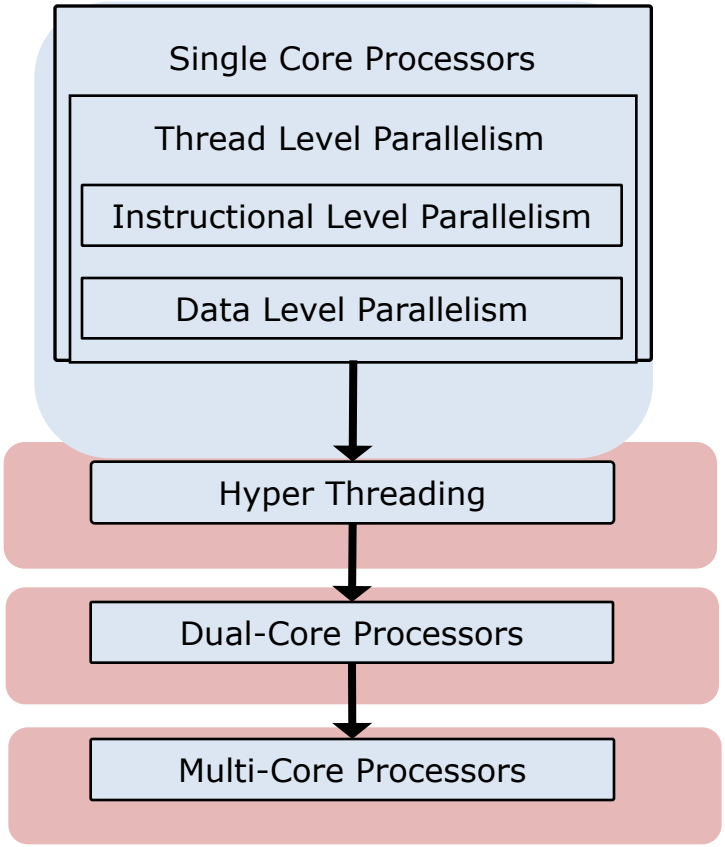
Multicore Programming Concepts and Tools

Prof. Yan Luo

For UMass Lowell 16.480/552

Acknowledgement: Slides are adopted from Intel Training Materials

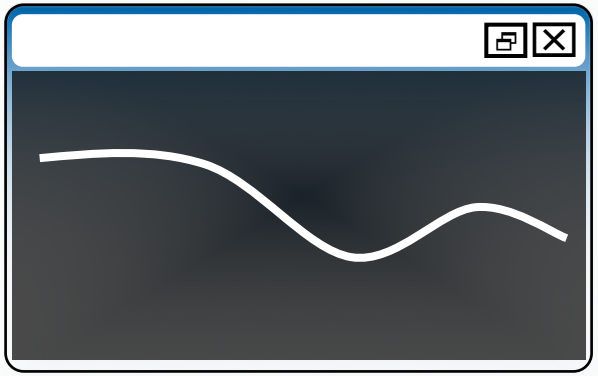
Evolution of Multi-Core Technology



Evolution of Multi-Core Technology

What is a Thread?

- **A thread:**
  - Is a single sequential flow of control within a program.
  - Is a sequence of instructions that is executed.



Single Thread Running Within a Program

Multiple Threads Running Within a Program

Why Use Threads?

Benefits of using threads are:

- Increased performance
- Better resource utilization
- Efficient data sharing

Risks of using threads are:

- Data races
- Deadlocks
- Code complexity
- Portability issues
- Testing and debugging difficulty

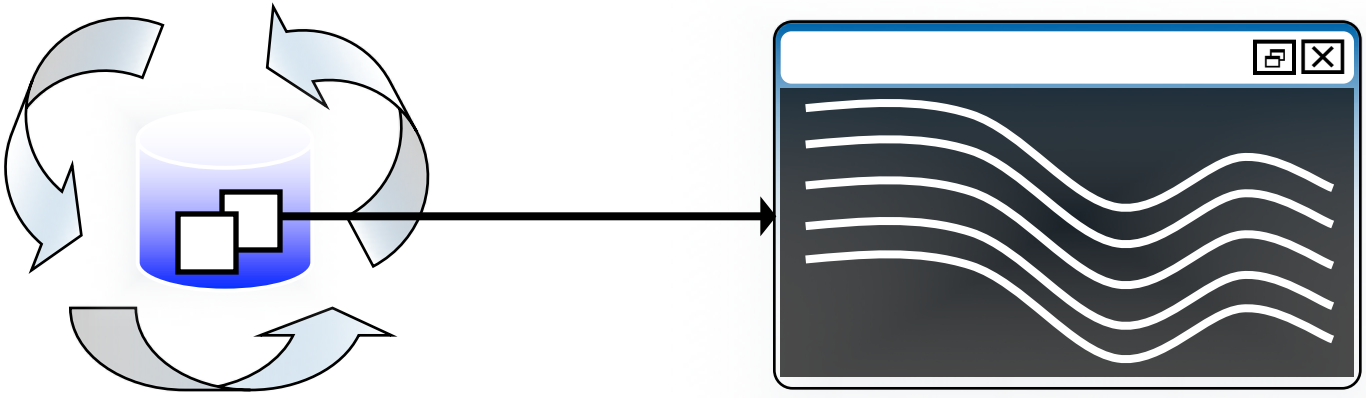
What is a Process?

Modern operating systems load programs as processes. To the operating system, processes have two roles:

- Resource holder
- Execution (thread) unit

Definition:

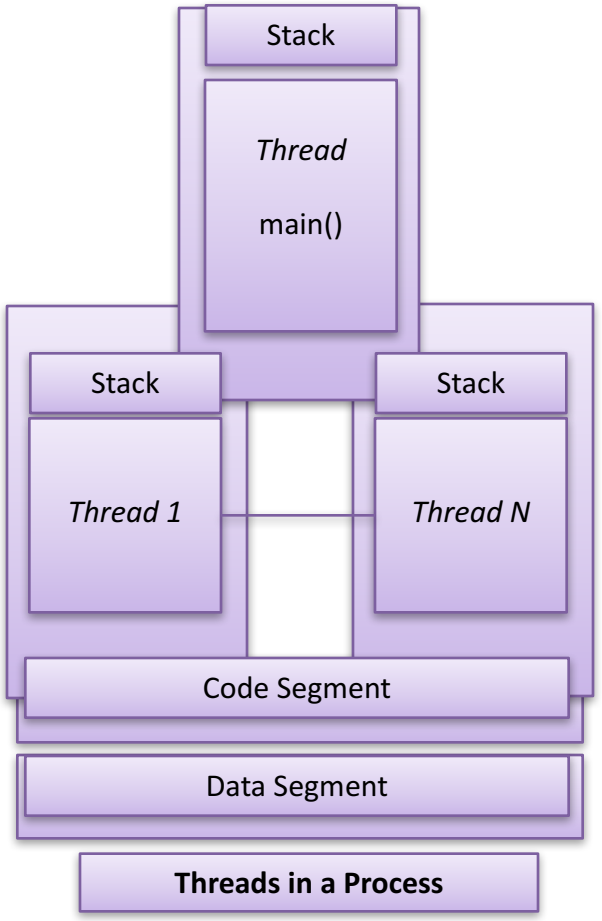
- A *process* has the main thread that initializes the process and begins executing the instructions.



Processes and Threads

Relationship of threads with a process:

- A process has the main thread that initializes the process and begins executing the instructions.
- Any thread can create other threads within the process.
- Each thread gets its own stack.
- All threads within the process share code and data segments.



Processes Vs Threads

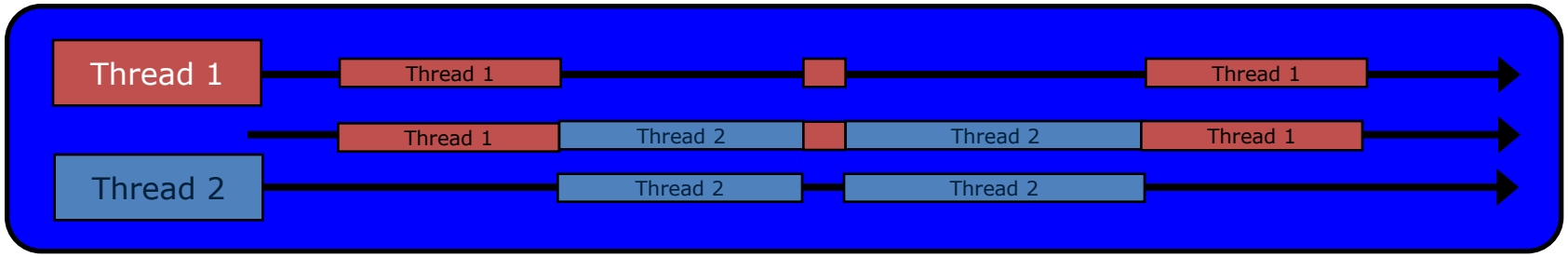
The following table lists the differences between threads and processes:

Category	Process	Thread
Address space	A process has its own address space, which the operating system protects.	A thread shares the process address space with other threads.
Interaction	A process interacts with other processes within operating system primitives and through shared locations in the operating system kernel.	A thread interacts with the other threads of the program by using primitives of the concurrent program language or library within the shared memory of the process.
Context switching	Context switching is heavy, due to the requirement that the entire process state must be preserved.	Context switching is light, because only the current register state needs to be saved.

Concurrency

Definition:

- Concurrency occurs when two or more threads are *in progress* simultaneously.
- Concurrent threads can execute on a *single* processor.

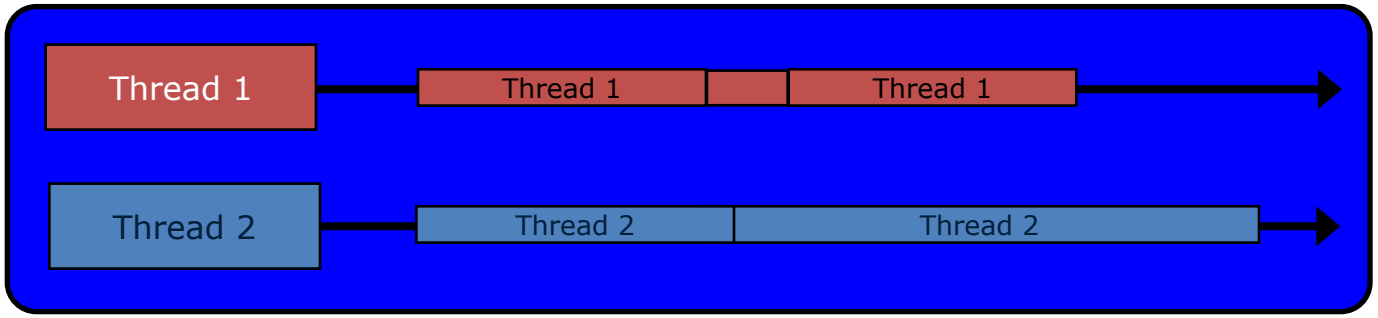


Concurrency in Multithreaded Applications

Parallelism

Definition:

- Parallelism occurs when multiple threads *execute* simultaneously.
- Parallel threads can execute on *multiple* processors.



Parallelism in Multithreaded Applications

Introduction to Design Concepts

The best time for threading while developing an application is during the design phase.

The following are the design concepts:

- Threading for functionality
- Threading for performance
    - Threading for turnaround
    - Threading for throughput
- Decomposing the work
    - Task decomposition
    - Data decomposition

Threading for Functionality

You can assign different threads to different functions done by the application.

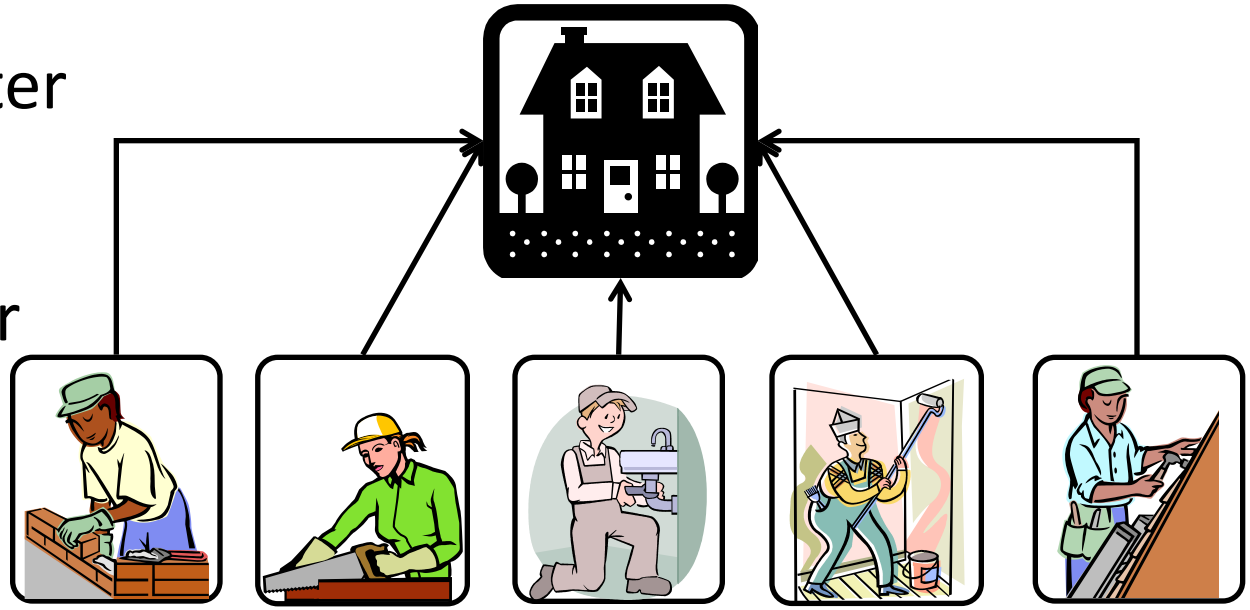
With threading for functionality:

- Chances of function overlapping are rare.
- It is easier to control the execution of concurrent functions within an application.
- Dependencies could persist between functions even without direct interference between computations.

Threading for Functionality – Example

Different people involved in *building a house* are:

- Bricklayer
- Carpenter
- Roofer
- Plumber
- Painter



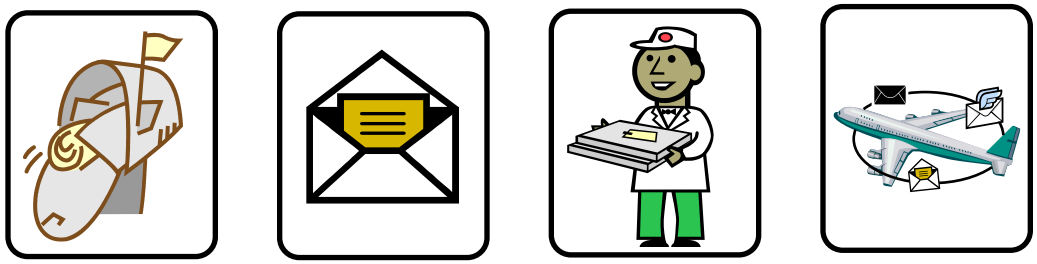
Example: Building a House

Threading for Performance

You can thread a serial code in an application to increase the performance of computations.

Example: Different tasks involved in a *postal service* are:

- Post office branches
- Mail sorters
- Delivery people
- Long distance transporters



Example: Postal Service

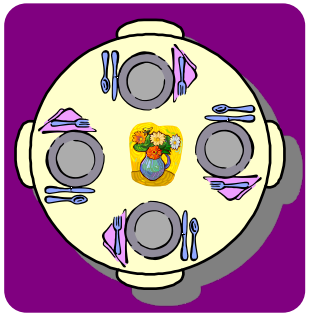
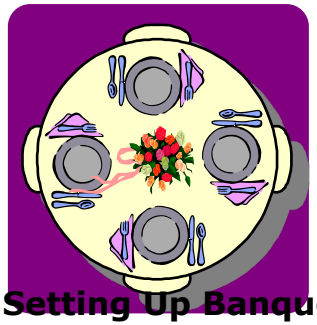
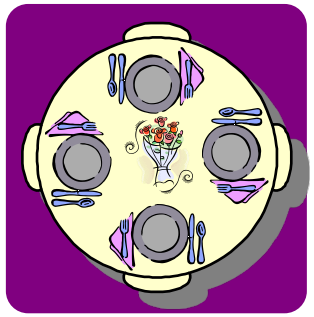
Threading for Turnaround

- *Threading for Turnaround* refers to completing a single job in the smallest amount of time possible.
- **Example: Different tasks involved in *setting up a dinner table* are:**
  - One waiter organizes the plates.
  - One waiter folds and places the napkins.
  - One waiter decorates the flowers and candles.
  - One waiter places the utensils, such as spoons, knives, and forks.
  - One waiter places the glasses.

Example: Setting Up Dinner Table

Threading for Throughput

- *Throughput* refers to accomplishing the most tasks in a fixed amount of time.
- **Example: Different tasks involved in *setting up banquet tables* are:**
  - Multiple waiters required.
  - Each waiter performs one specific task for all the tables.
  - Specialized waiters for placing the napkins.
  - Specialized waiters for decorating the flowers and candles.
  - Specialized waiters for placing the utensils, such as spoons, knives, and forks.
  - Specialized waiters for placing the glasses.

Example: Setting Up Banquet Tables

Decomposing the Work

- Logical chunking or breaking down of the programs into individual tasks and identifying the dependencies between them is known as *decomposition*.

The following table lists the decomposition methods, the respective design strategy, and implementation areas:

Decomposition	Design	Comments
Task	Different activities assigned to different threads.	Common in applications with several independent functions.
Data	Multiple threads performing the same operation but on different blocks of data.	Common in audio, processing, imaging, and scientific programming.

Task Decomposition

Decomposing a program based on the functions that it performs is called *task or functional decomposition*.

Key points to remember about task decomposition are:

- In multithreaded applications, you can divide the computation based on the natural set of independent tasks.
- Functional decomposition refers to mapping independent functions to threads that execute asynchronously.
- Parallel computing requires certain modifications to individual functions to preserve dependencies and to avoid race conditions.

Task Decomposition – Example

Consider a situation in which you want to weed and mow your lawn. You have two gardeners. You can assign the task to the gardeners based on the type of activity.

Example: Weeding and Mowing a Lawn

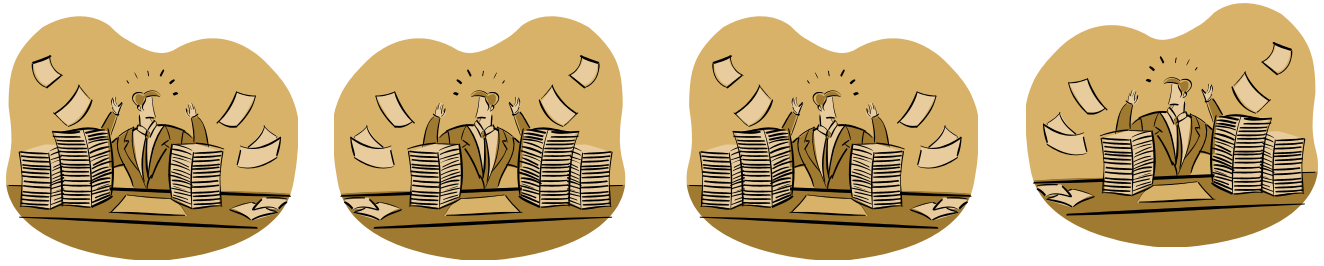
Data Decomposition

Dividing large data sets whose elements can be computed independently and associating the needed computation among threads is known as *data decomposition*.

Key points to remember about task decomposition are:

- The same independent operation is applied repeatedly to different data.
- Computation-intensive tasks with a large degree of independence like computation-intensive loops in applications are good candidates for data decomposition.

Example: The job is to *grade a large stacks of answer sheets* for a test.



Example: Grading the Answer Sheets

Which Decomposition Method to Use?

Task decomposition and data decomposition can often be applied to the same problem, depending on the number of available resources and the size of the task.

Examples:

- **Data decomposition:** In the mural painting example, you can divide the wall into two halves and assign each of the artists one half and all the colors needed to complete the assigned area.

- **Task decomposition:** In the final exam grading problem, if the graders take a single key and grade only those exams that correspond to that key, it would be considered task decomposition. Alternatively, if there are different types of questions in the exam, such as multiple choice, true/false, and essay, the job of grading could be divided based on the tasks to specialists in each of those question types.

Introduction to Correctness Concepts

The usage of threads helps you enhance the performance by allowing you to run two or more concurrent activities.

Correctness Concepts:

- Critical Region
- Mutual exclusion
- Synchronization

Race Conditions

Race conditions:

- Are the most common errors in concurrent programs.
- Occur because the programmer assumes a particular order of execution but does not guarantee that order through synchronization.

A Data Race:

- Refers to a storage conflict situation.
- Occurs when two or more threads simultaneously access the same memory location while at least one thread is updating that location.

Result in two possible conflicts:

- Read/Write conflicts
- Write/Write conflicts

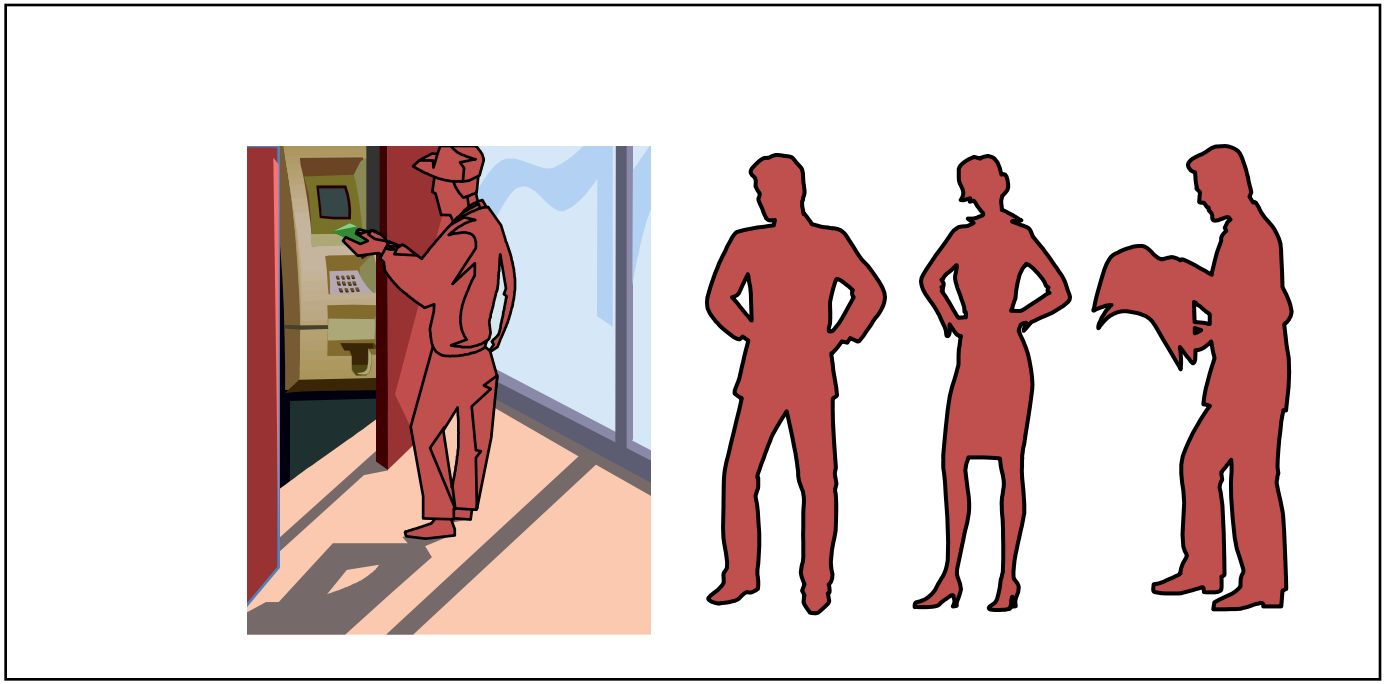
Critical Region

Critical Regions:

- Are portions of code that access (read and write) shared variables.
- Must be protected to ensure data integrity when multiple threads attempt to access shared resources.

Mutual Exclusion

Mutual exclusion refers to the program logic used to ensure single-thread access to a critical region.



Example: ATM Machine

Synchronization

Synchronization:

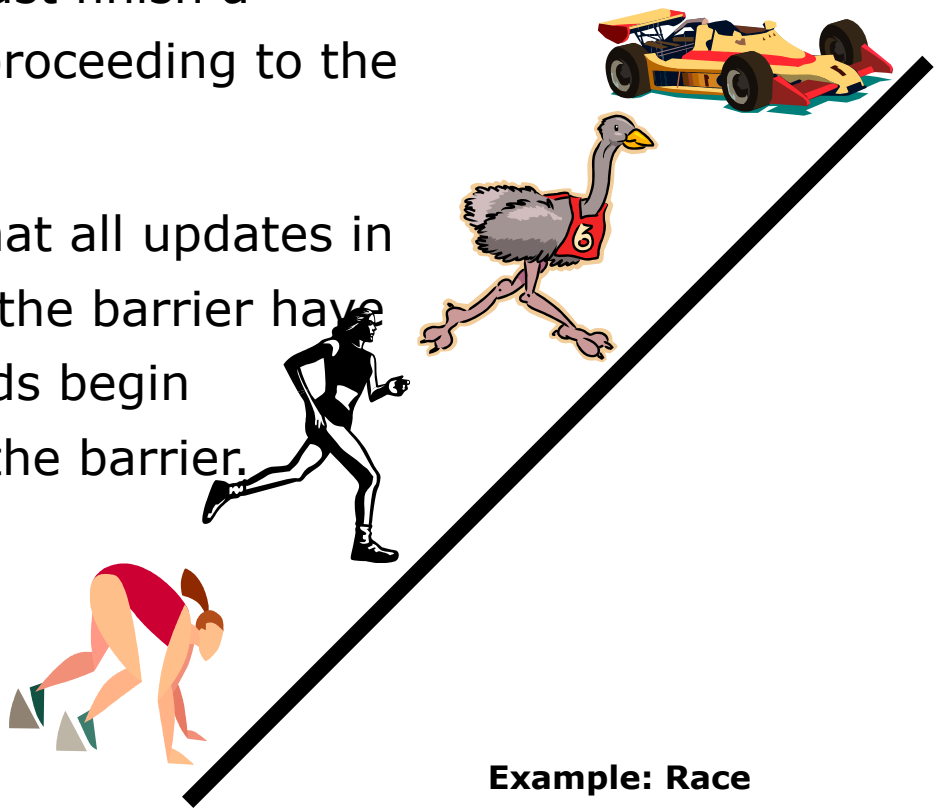
- Is the process of implementing mutual exclusion to coordinate access to shared resources in multithreaded applications is known as *synchronization*.
- Controls the relative order of thread execution and resolve any conflicts among threads that might produce unwanted behavior.

Example: Library

Barrier Synchronization

Barrier Synchronization:

- Is used when all threads must finish a portion of the code before proceeding to the next section of code.
- Is usually done to ensure that all updates in the section of code prior to the barrier have completed before the threads begin execution of the code past the barrier.



Example: Race

Deadlocks

Deadlock:

- Occurs when a thread waits for a condition that never occurs.
- Most commonly results from the competition between threads for system resources held by other thread.

The four necessary conditions for a deadlock are:

- Mutual exclusion condition
- Hold and wait condition
- No preemption condition
- Circular wait condition

Example: Traffic Jam

Livelock

Livelock refers to a situation when:

- A thread does not progress on assigned computations, but the thread is not blocked or waiting.
- Threads try to overcome an obstacle presented by another thread that is doing the same thing.

Example: Robin Hood and Little John

Introduction to Performance Concepts

Performance Concepts:

- Simple Speedup
- Computing Speedup
- Efficiency
- Granularity
- Load Balance

Simple Speedup

Speedup measures the time required for a parallel program execute versus the time the best serial code requires to accomplish the same task.

Speedup = Serial Time / Parallel Time

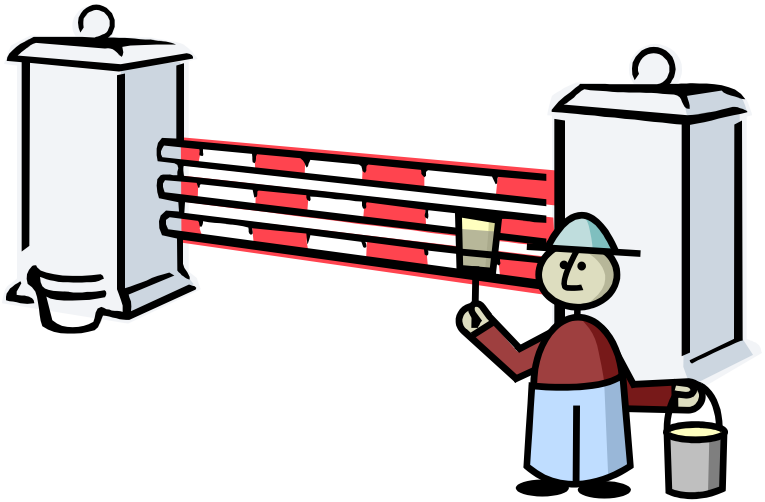
According to Amdahl's law, speedup is a function of the fraction of a program that is parallel and by how much that fraction is accelerated.

Speedup = 1 / [*S*+(1-*S*)/*n* +*H(n)*]

Computing Speedup – Example

Painting a picket fence requires:

- 30 minutes of preparation (serial).
- One minute to paint a single picket.
- 30 minutes to clean up (serial).



Example: Painting a Picket Fence

Computing Speedup

Consider how speedup is computed for different numbers of painters:

Number of Painters	Time	Speedup
1	30 + 300 + 30 = 360	1.0X
2	30 + 150 + 30 = 210	1.7X
10	30 + 30 + 30 = 90	4.0X
100	30 + 3 + 30 = 63	5.7X
Infinite	30 + 0 + 30 = 60	6.0X

Parallel Efficiency

Parallel Efficiency:

- Is a measure of how efficiently processor resources are used during parallel computations.
- Is equal to (*Speedup / Number of Threads) * 100%.*

Consider how efficiency is computed with different numbers of painters:

Number of Painters	Time	Speedup	Efficiency
1	360	1.0X	100%
2	30 + 150 + 30 = 210	1.7X	85%
10	30 + 30 + 30 = 90	4.0X	40%
100	30 + 3 + 30 = 63	5.7X	5.7%
Infinite	30 + 0 + 30 = 60	6.0X	very low

Granularity

Definition:

- An approximation of the ratio of *computation* to *synchronization*.

The two types of granularity are:

- **Coarse-grained:** Concurrent calculations that have a large amount of computation between synchronization operations are known as *coarse-grained*.
- **Fine-grained:** Cases where there is very little computation between synchronization events are known as *fine-grained*.

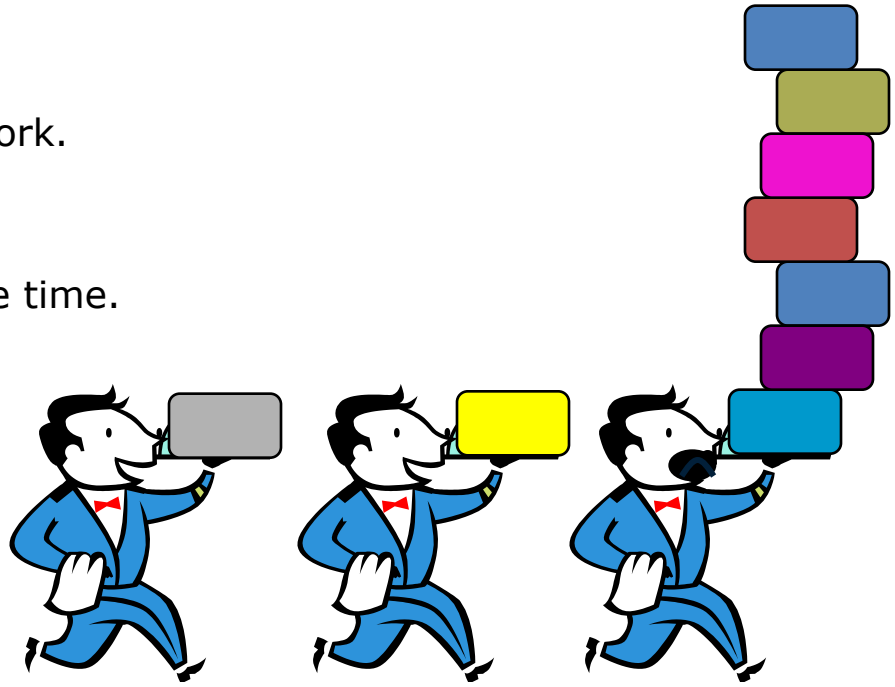
Example: Field and Farmers

Load Balance

Load balancing refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.

Most effective distribution is such that:

- Threads perform equal amounts of work.
- Threads that finish first sit idle.
- Threads finish work close to the same time.



Example: Cleaning Banquet Tables

Summary

- A *thread* is a discrete sequence of related instructions that is executed independently. It is a single sequential flow of control within a program.
- The *benefits* of using threads are increased performance, better resource utilization, and efficient data sharing.
- The *risks* of using threads are data races, deadlocks, code complexity, portability issues, and testing and debugging difficulty.
- Every process has at least one thread, which is the main thread that initializes the process and begins executing the initial instructions.
- All threads within a process share code and data segments.
- Concurrent threads can execute on a single processor. Parallelism requires multiple processors.

Summary (Continued)

- *Turnaround* refers to completing a single a task in the smallest amount of time possible, whereas accomplishing the most tasks in a fixed amount of time refers to *throughput*.
- Decomposing a program based on the number and type of functions that it performs is called *functional decomposition*.
- Dividing large data sets whose elements can be computed independently, and associating the required computation among threads is known as *data decomposition* in multithreaded applications.
- Applications that scale with the number of independent functions are probably best suited to functional decomposition while applications that scale with the amount of independent data are probably best suited to data decomposition.
- *Race conditions* occur because the programmer assumes a particular order of execution but does not use synchronization to guarantee that order.

Summary (Continued)

- Storage conflicts can occur when multiple threads attempt to simultaneously update the same memory location or variable.
- *Critical regions* are parts of threaded code that access (read or write) shared data resources. To ensure data integrity when multiple threads attempt to access shared resources, critical regions must be protected so that only one thread executes within them at a time.
- *Mutual exclusion* refers to the program logic used to ensure single-thread access to a critical region.
- *Barrier synchronization* is used when all threads must have completed a portion of the code before proceeding to the next section of code.
- *Deadlock* refers to a situation when a thread waits for an event that never occur. This is usually the result of two threads requiring access to resources held by the other thread.

Summary (Continued)

- *Livelock* refers to a situation when threads are not making progress on assigned computations, but are not idle waiting for an event.
- S*peedup* is the metric that characterizes how much faster the parallel computation executes relative to the best serial code.
- *Parallel Efficiency* is a measure of how busy the threads are during parallel computations.
- *Granularity* is defined as the ratio of computation to synchronization.
- *Load balancing* refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.

Intel® Compiler and Intel® VTune™ Performance Analyzer

Introduction to Intel Tools - Foundation

Two of the tools that Intel offers to make threading easier and more comprehensive are:

- **Intel® Compiler:**
  - Accelerates application performance.
  - Supports multithreading capabilities in multi-core and multiprocessor systems.

- **Intel® VTune™ Performance Analyzer:**
  - Identifies sections where an application can be threaded.
  - Monitors performance of the application and the computer.
  - Helps you tune your application.

Intel® Compilers

- **Intel compilers:**
  - Supports IA-32, Intel® Core processor, and Intel® Itanium® architectures.
  - Can be used with Microsoft Visual C++, GNU C++, and Compaq Visual Fortran.
  - Offer advanced technology to optimize applications.
  - Provide support to threaded applications.
  - Are compatible with industry processes and standards such as the C++ application binary interface.
  - Allow you to compile applications for both 32-bit and 64-bit environments.
  - Are compatible with Intel® multi-core processors.

Optimization Switches

The Intel® compiler supports certain switches that enable optimization as shown in the table:

Windows	Linux	Mac	Description
/Od	-O0	-O0	Disables optimizations
/Zi	-g	-g	Creates symbols and provides symbol information
/01	-01	-01	Optimizes for size of binary code (server code)
/02	-02	-02	Optimizes for speed (default)
/QaxP	-axP	-axP	Optimizes for Intel processors with SSE3 capabilities
/03 with /QaxP	-03 with -axP	-03 with -axP	Optimizes for data cache—loopy and floating-point code

Profile-Guided Optimizations

Definition:

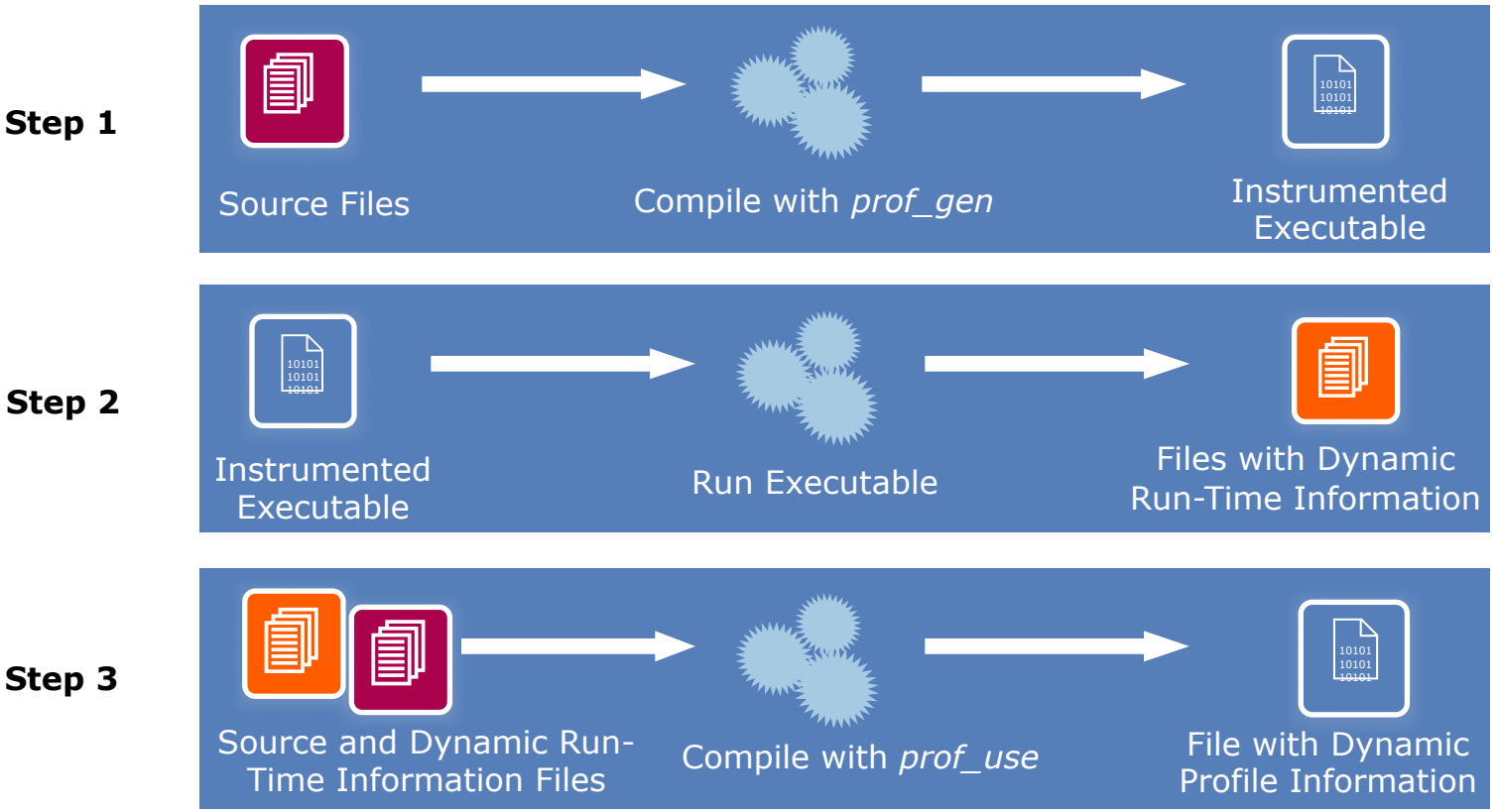
- *Profile-guided optimization* (PGO) is used for analyzing application workload at run time.

PGO has the following features:

- Detects and provides opportunities to optimize the performance of the application.
- Can be used for optimizing function ordering, reorganizing code layout for improved instruction cache memory utilization, and accurate branch predictions.
- Helps in ordering the sequence of cases in a C/C++ switch statement.
- PGO is a family of switches.

PGO Process

PGO is a three-stage process, which involves dynamic analysis.



PGO Process

PGO Process (Continued)

PGO performs the following optimizations:

- Basic block ordering
- Better register allocation
- Better decision of functions to inline
- Function ordering
- Switch-statement optimization
- Better vectorization decisions

PGO uses the *prof_gen* switch for optimization

- Additional functionality id provided by *prof_genx*

***prof_genx* enables use of two Intel tools:**

- Code Coverage (codecov)
- Test Prioritization (tselect)

Inter-Procedural Optimizations

Definition:

- *Inter-Procedural Optimization* (IPO) performs a static analysis of the application at link time along with the details of the variables and functions seen during a typical link step.

IPO has the following features:

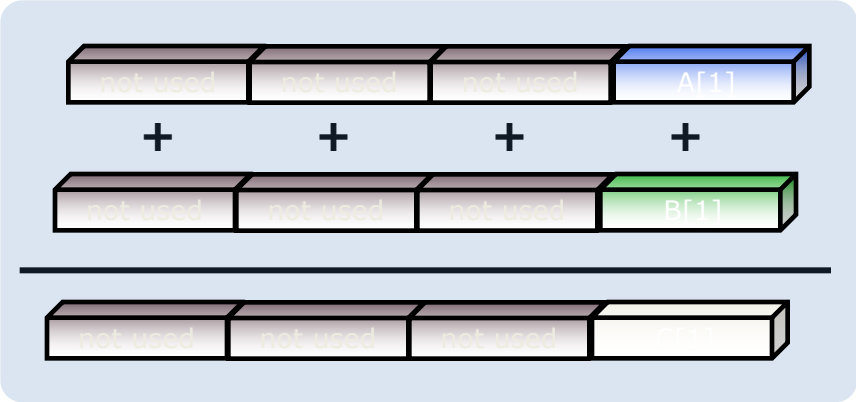
- Improves application performance in programs that contain frequently used small- or medium-sized functions.
- Is useful for programs that contain function calls within loops.
- Performs analysis for multiple files or over entire programs to detect and perform optimizations.

Compiler-Based Vectorization

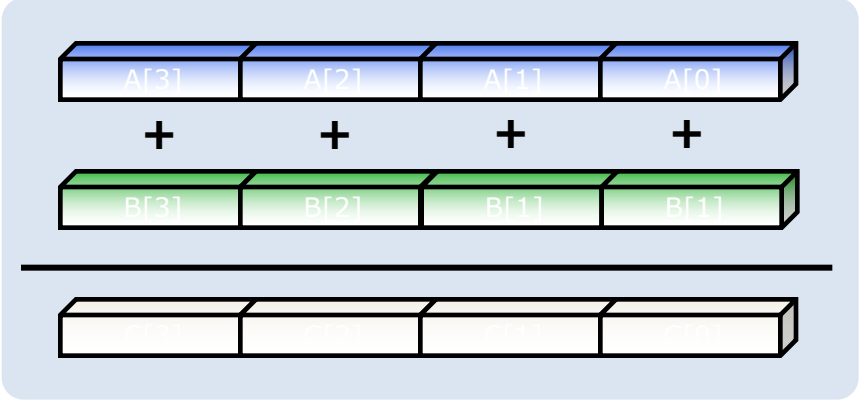
Compiler-based vectorization allows you to invoke the Streaming SIMD Extensions (SSE) capabilities of the underlying processor.

Suppose you are dealing with single precision, floating-point elements within the following scalar loop:

```
float a[64], b[64], c[64];
int i;
for (i=0; i<1000; i++)
        c[i] = b[i] + a[i];
```



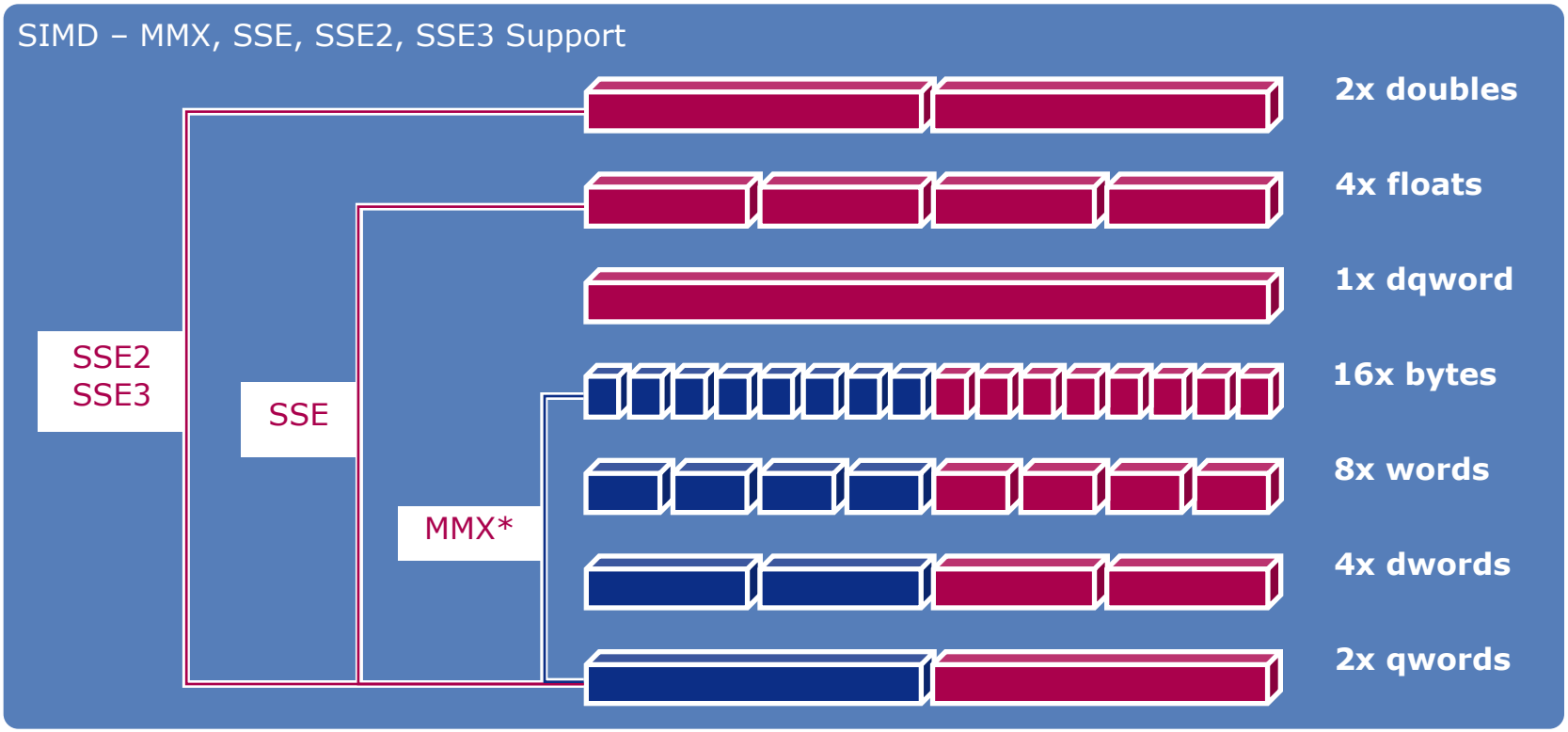
Sequence of Execution without Vectorization



Sequence of Execution with Vectorization

SSE3 Instructions

Streaming SIMD Extensions3 also known as Prescott New Instructions (PNI) is the third iteration of the SSE instruction set for the IA-32 architecture.



Supported Data Sizes for Vectorization

Compiler-Based Vectorization (Continued)

Compiler-based vectorization is processor-specific. A compiler switch is used to control the level of vectorization that instructions should incorporate into the application being compiled based on the target processor.

Two of the most recent important processor flag values that you can use with the Intel compiler are:

Processor Value	Windows	Linux	Mac
W	/QxW	-xW	Does not apply
P	/QxP	-xP	Vectorization occurs by default
	/QaxP	-axP	

Automatic Processor Dispatch – ax[?]

The automatic processor dispatch switch is used in situations where:

- The target processor may be unknown.
- The application may run on many generations of processors.
- You want to take advantage of high-level vectorization available on most of the recent processors.

The automatic processor dispatch version of the vectorization switch uses processor-specific instructions and performs the vectorization:

Linux	Windows	Description
-axp	/Qaxp	Enables scalar as well as vector addition

The */QaxP* switch involves low overhead and may result in slightly larger binaries. However, this is highly application-dependant.

Intel® VTune™ Performance Analyzer

Utility:

- Intel® VTune™ Performance Analyzer is a powerful and an easy-to-use tool.
- It collects, analyzes, and displays performance data for a wide variety of applications.

The VTune Performance Analyzer performs the following functions:

- Collects performance data from the system
- Organizes and displays the data in a variety of interactive views
- Identifies potential performance issues and can suggest improvements

Supported Environments

There are two ways to install and run the Intel® VTune™ Performance Analyzer:

- **For native data collection:** Profile applications that are running on the system with the VTune Performance Analyzer installed on it.
- **For remote data collection:** Also known as the Host/Target environment. Run profiling experiments on other systems on your subnet with VTune Performance Analyzer remote agents installed on them.

Native Performance Analysis

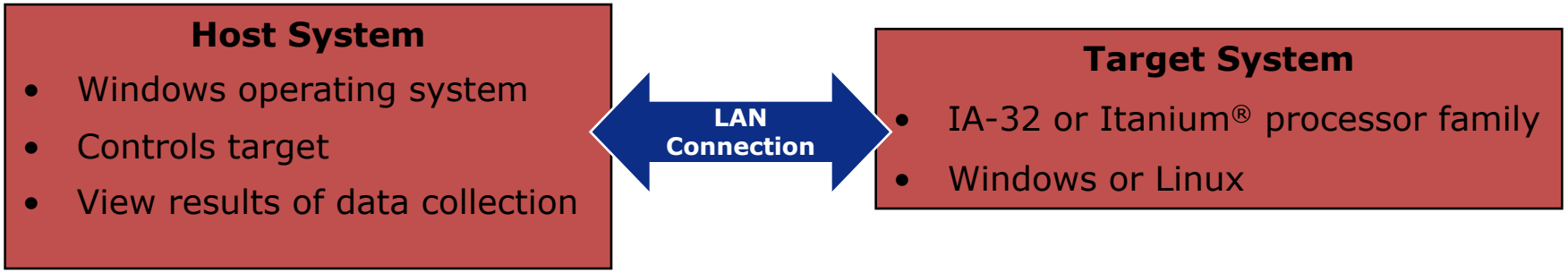
The system performing local or native data collection can belong to any of the following processor families:

- **Intel® Core and IA-32 Processors:**
  - Microsoft Windows* operating systems—GUI and command line interface (CLI)
  - Red Hat Linux*—GUI and CLI
  - SuSE Linux*— GUI and CLI

- **Itanium® Family Processors:**
  - Microsoft Windows operating systems—GUI and CLI
  - Red Hat Linux—GUI and CLI
  - SuSE Linux—GUI and CLI

Remote Data Collection: Host/Target Environment

The Intel® VTune™ Performance Analyzer is installed on a host system, and the remote agent is installed on the target system.

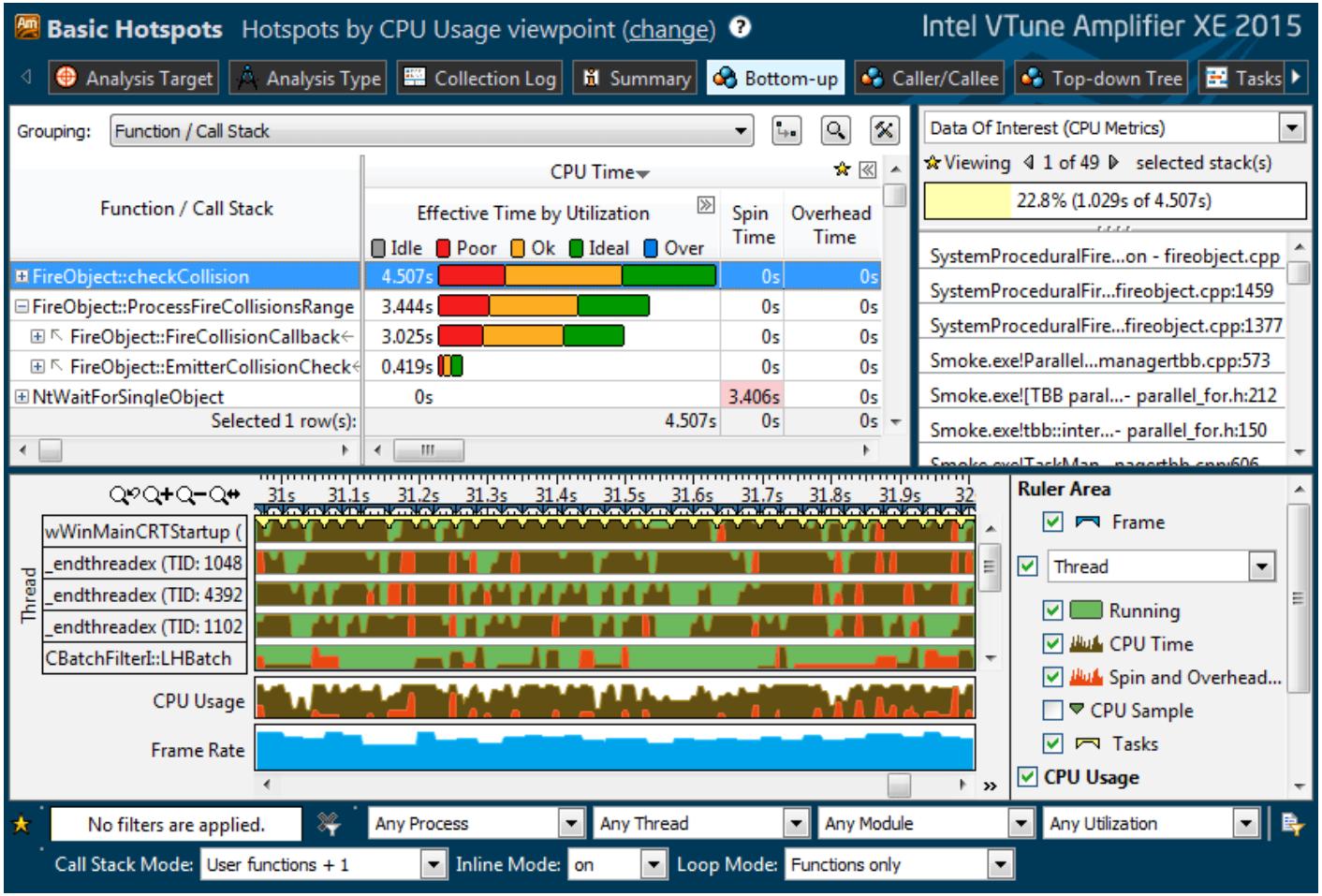


Intel® VTune™ Performance Analyzer

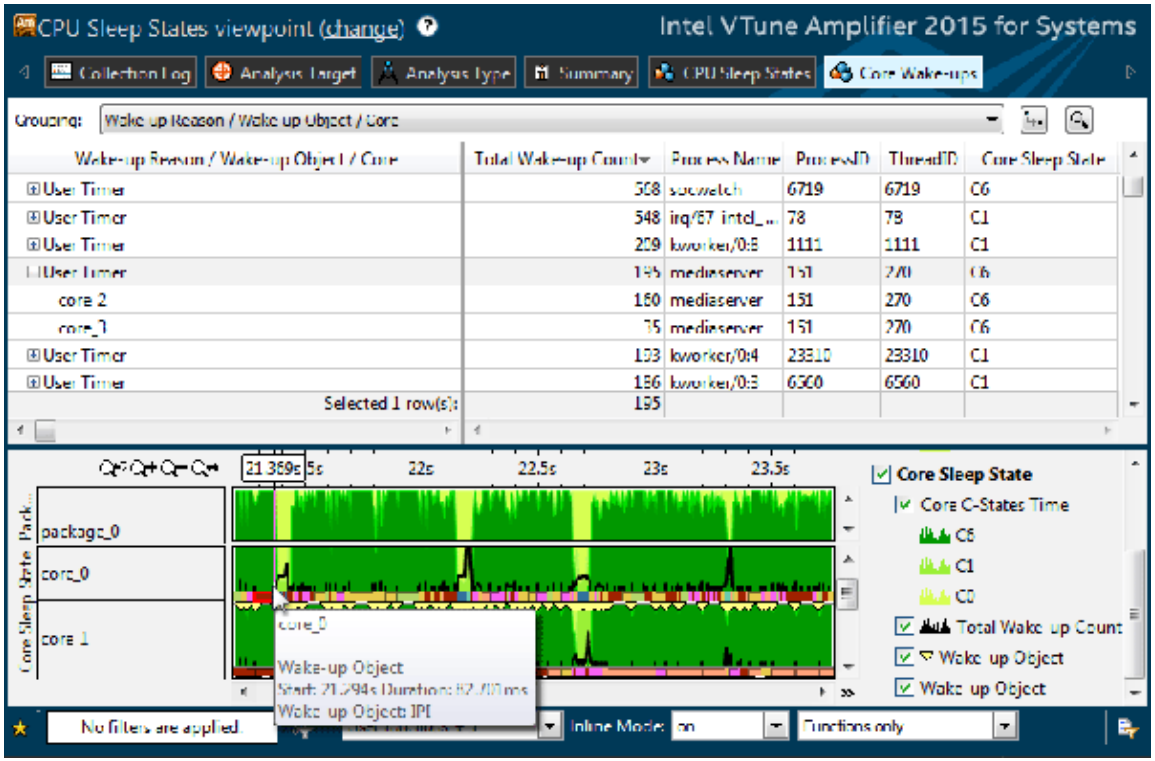
The Intel® VTune™ Performance Analyzer offers features that help in performance tuning:

- **Sampling:** Calculates the performance of an application over a period and for various processor events.
- **Call Graph:** Provides a graphical view of the flow of an application and helps you identify critical functions and timing details.
- **Counter Monitor:** Provides system-level performance information such as resource utilization during the execution of an application. This functionality works only on Windows.
- **Hotspots View:** Helps identify the area of code that consumes the maximum CPU time.
- **Tuning Assistant:** Provides tuning advice by analyzing the performance data. The tuning advice provides a guideline for the programmer to improve the performance of an application. This functionality works only on Windows.

Intel VTune Amplifier



Energy Profiling



Sampling – A Detailed Study

Sampling is the process of collecting performance data by observing the processor state at regular defined intervals.

Sampling has the following features:

- **Identify Hotspots:** A hotspot is a section of code that contains a significant amount of activity for some internal processor event, such as clockticks, cache misses, or disk reads.
- **Identify Bottlenecks:** A bottleneck is an area in code that slows down the execution of an application. To optimize code, you need to remove the bottlenecks.

Usually people confuse a hotspot and a bottleneck. A hotspot shows where to focus your attention when looking for bottlenecks. All bottlenecks are hotspots, but all hotspots need not necessarily be bottlenecks.

Demo: Find the Hotspot

Objective:

- Identify hotspots with the Intel® VTune™ Performance Analyzer.

The main idea of this activity is to identify how to use sampling to identify hotspots on sample code using the VTune Performance Analyzer.

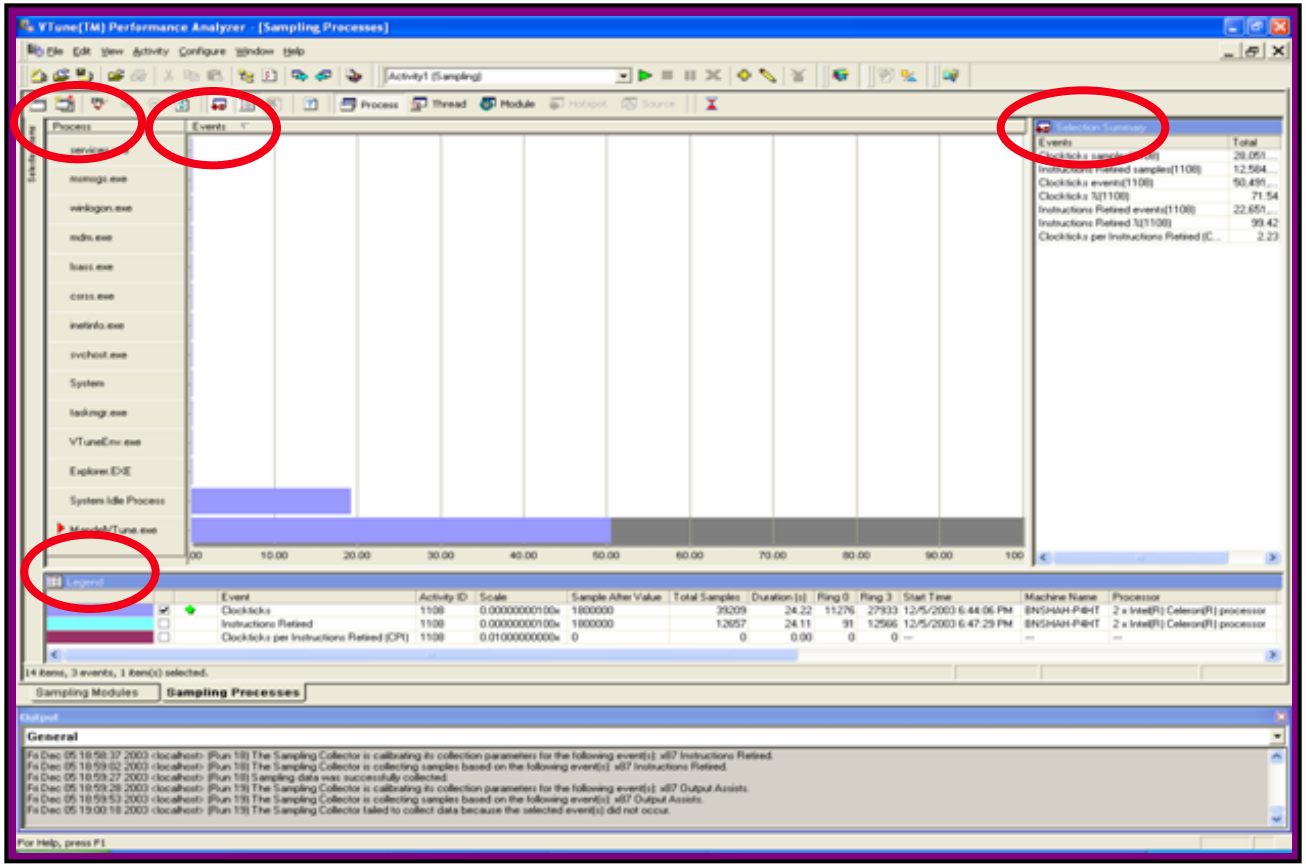
Benefits of Profiling with Sampling

Profiling with sampling has three key benefits:

- You need not modify your code.
- Sampling is system-wide. It is not restricted to your application.
- Sampling involves low overhead.
    - Its validity is always statistically relevant, but is highest when perturbation is low.

Sampling Collector

The sampling collector is responsible for collecting the sampling data.



Process View

Time-Based Sampling

Time-Based Sampling (TBS) helps to reveal the routines in which the application spends the most time. This feature is applicable for Windows* only.

When you perform an activity by using TBS, the Intel® VTune™ Performance Analyzer performs the following functions:

- Executes the application you launched.
- Interrupts the processor at the sampling interval and collects data on the current process executing.
- Continues to collect sampling data until the specified application terminates or the specified sampling duration ends.
- Analyzes the collected data, creates an activity result in the Tuning Browser window, and displays the data collected for each module.

Event-Based Sampling

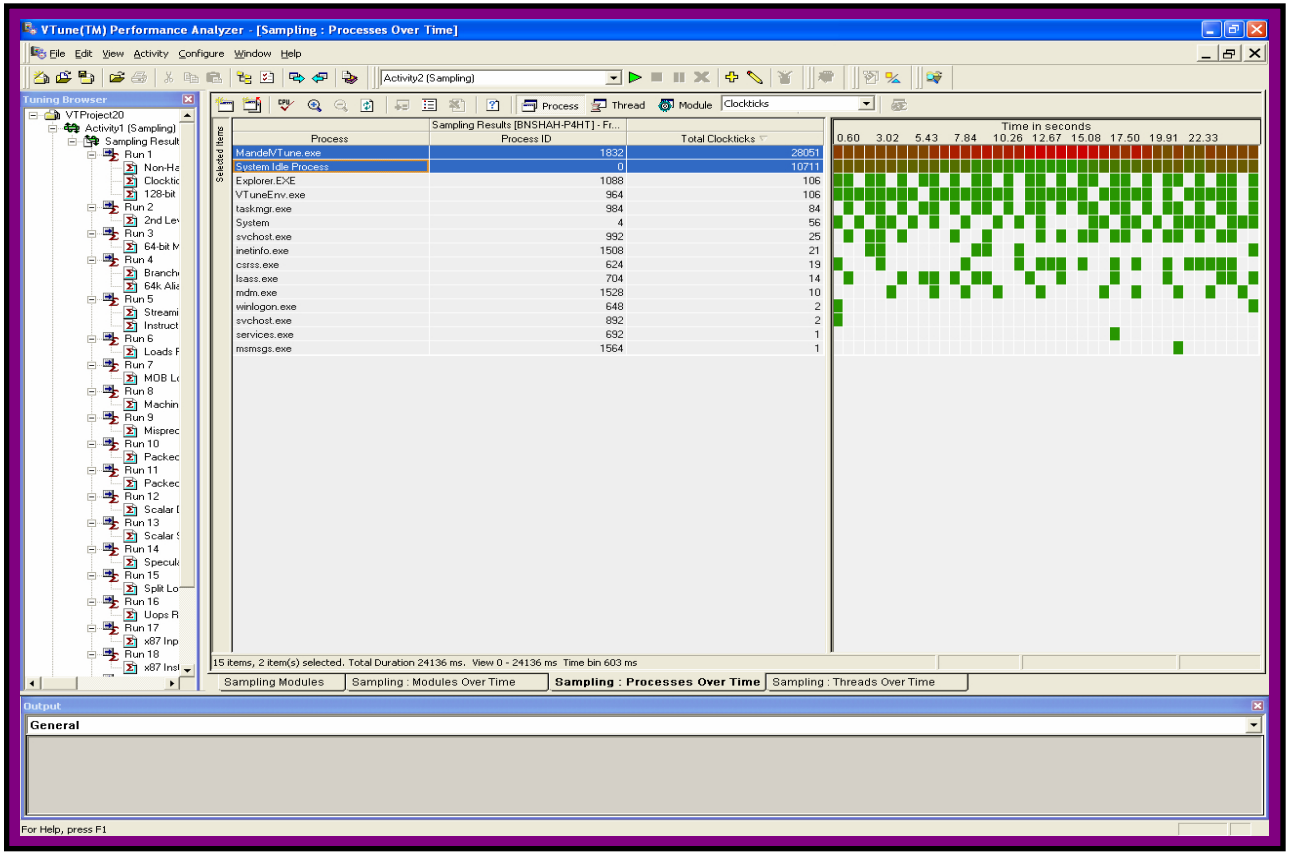
Event-Based Sampling (EBS) is triggered by a processor event counter underflow.

A few things to remember about EBS are as follows:

- EBS is performed on processor events, such as L2 cache misses, branch mispredictions, and retired floating-point instructions.
- EBS helps you determine which process, thread, module, function, or code line in the application generates the largest number of chosen processor events.
- You choose one or more processor events of interest from the Events list when configuring the sampling collector.
- EBS does not work on laptops and on non-Intel® processors.

Sampling Over Time

The Sampling Over Time functionality shows how sample distributions change with time.



Sampling Over Time View

Call Graph – A Detailed Study

The call graph collector of the Intel® VTune™ Performance Analyzer helps you obtain information about the functional flow of an application.

Call graph helps you:

- Obtain information about the number of times a function is called from a specific location.
- Obtain information about the amount of time spent in each function when executing its code.

Call Graph Profiling

Call graph performs the following functions:

- Tracks the function entry and exit points of your code at run time.
- Uses this data to determine program flow, critical functions, and call sequences.
- Requires the instrumentation of target binaries.
- Only profiles the code in Ring 3 or application level modules at run time.

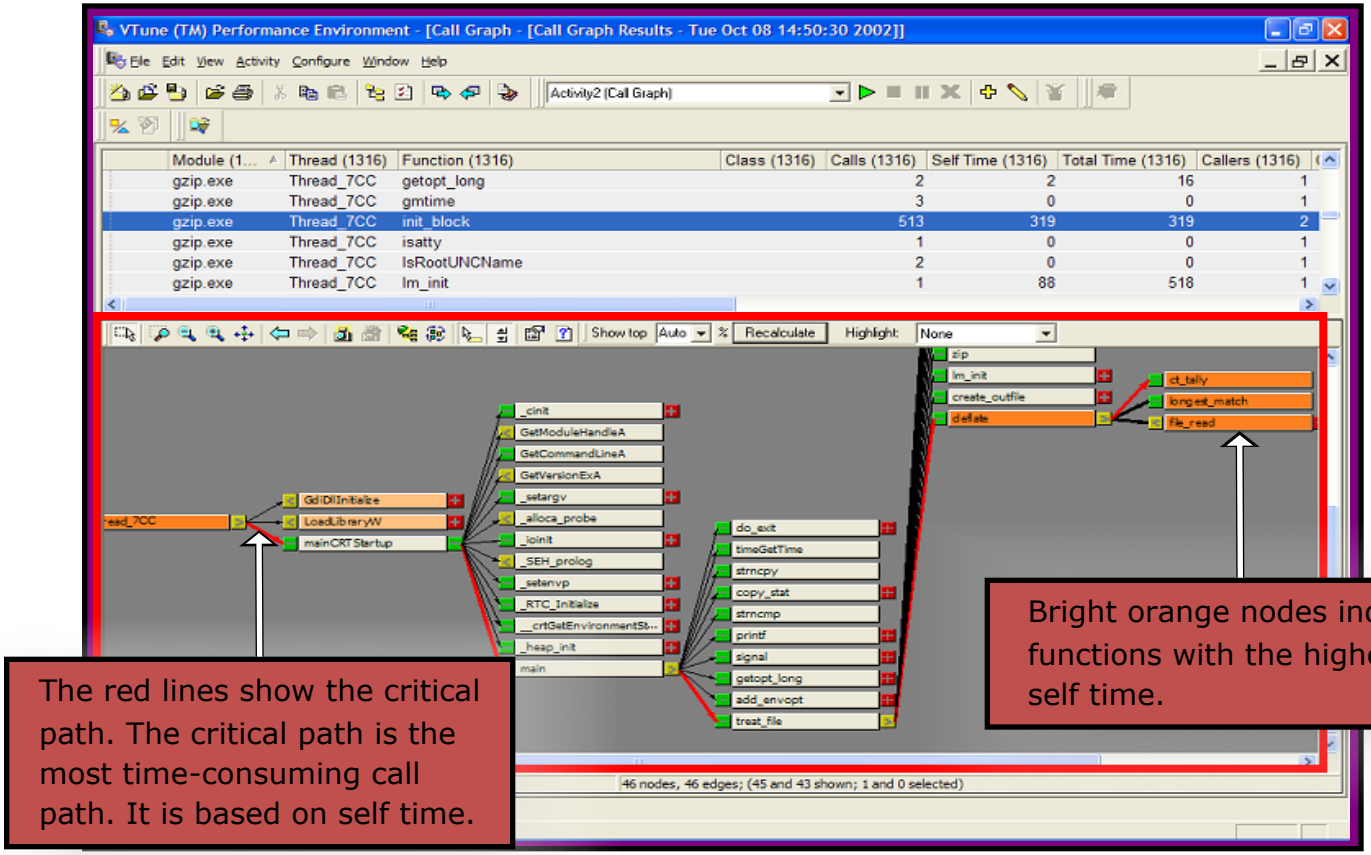
With call graph, you do not need to use a special compiler or to insert special API calls into your code. You only need generate debug information when you build the application.

What can you Profile?

Call graph can profile different software applications:

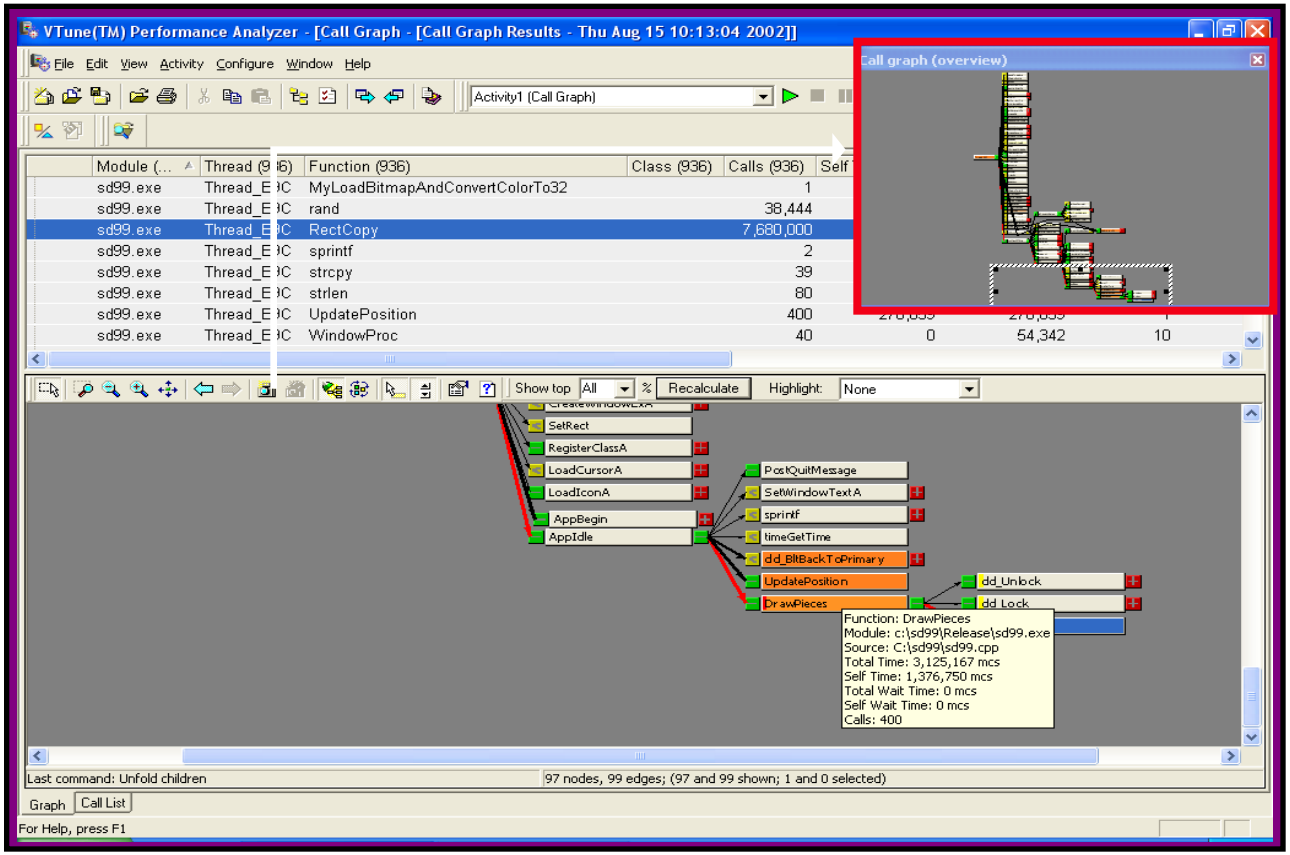
- Win32* applications
- Stand-alone Win32 DLLs
- Stand-alone COM+* DLLs
- Java* applications
- .NET* applications
- ASP.NET* applications
- Linux32* applications
- Linux64* applications

Call Graph View



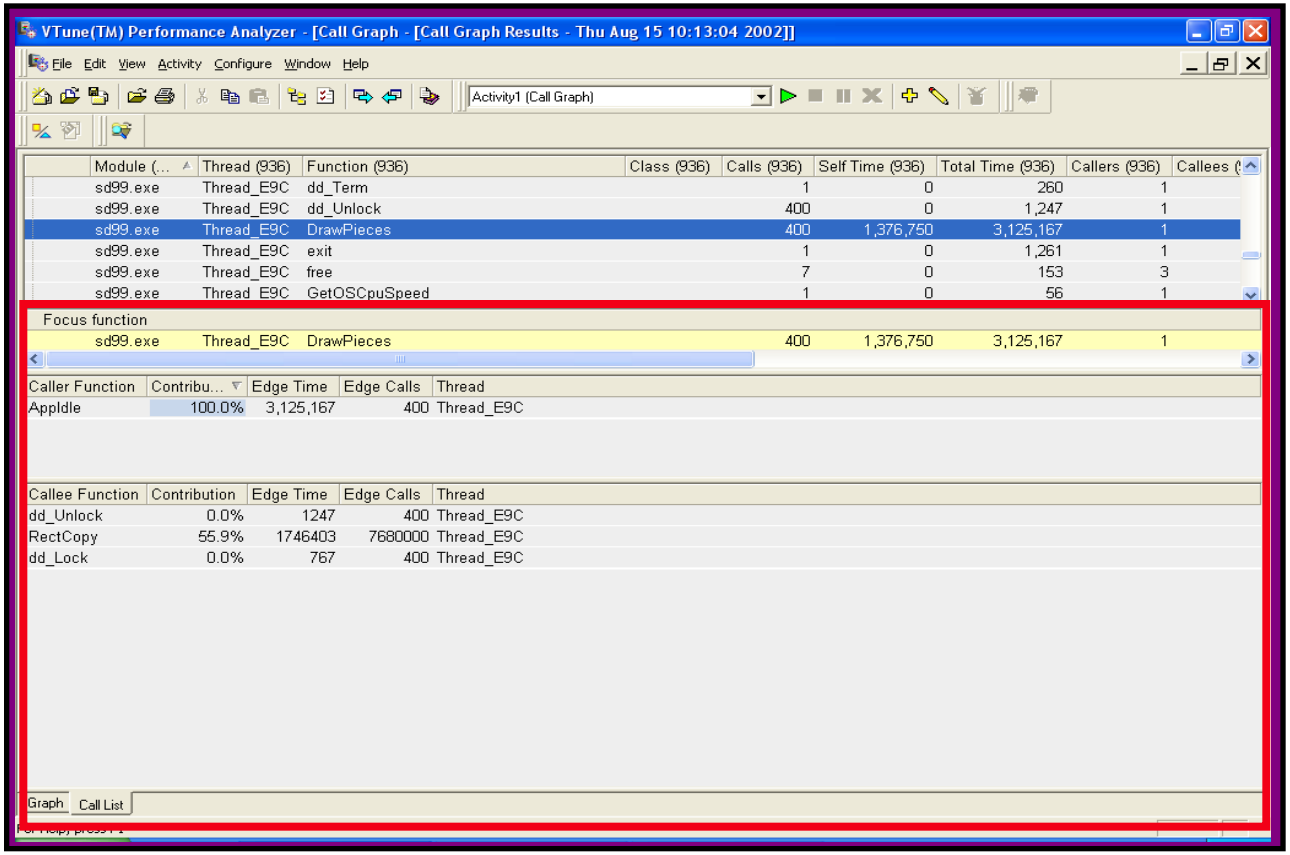
Call Graph View

Call Graph Navigation View



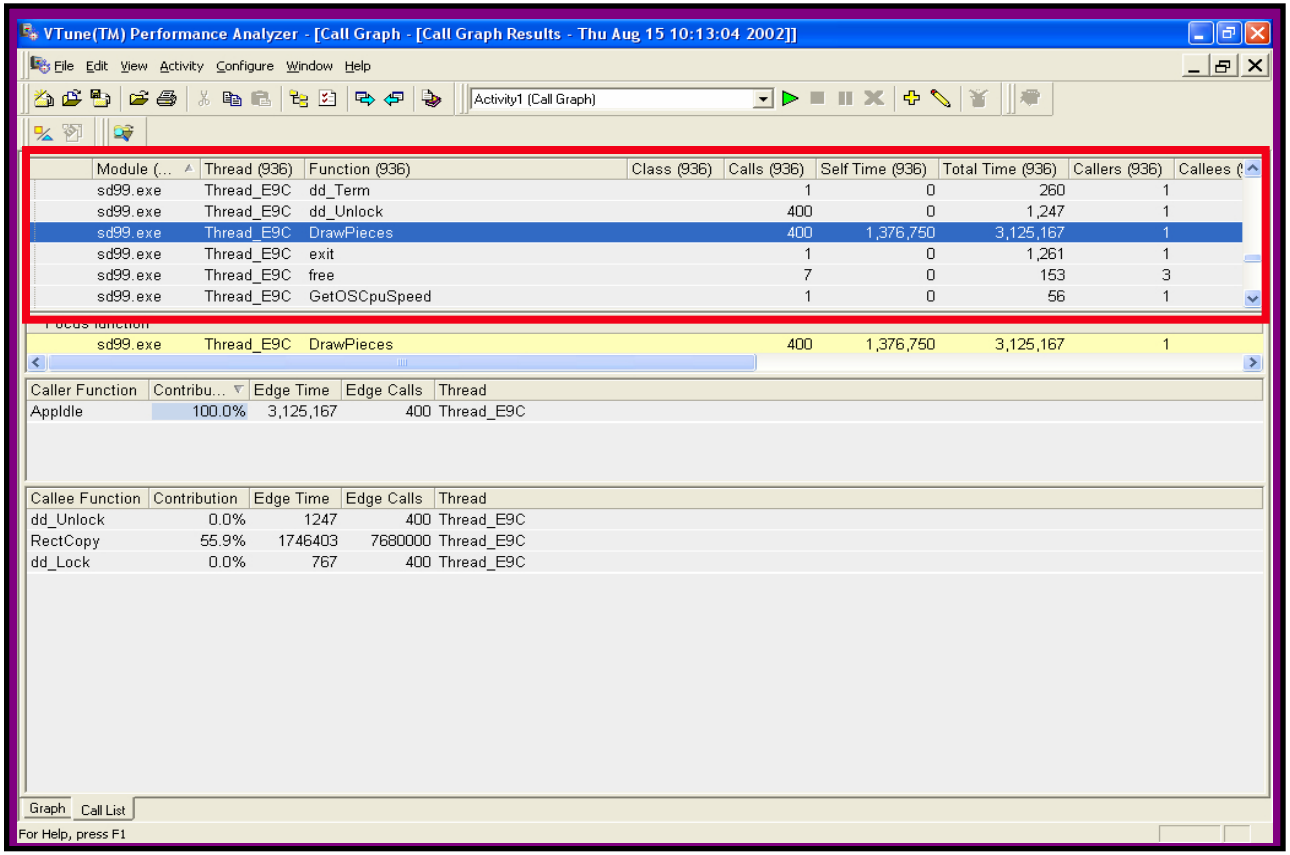
Call Graph Navigation View

Call Graph Call List View



Call Graph Call List View

Call Graph Function Summary View



Call Graph Function Summary View

Call Graph Metrics

The Call Graph Function Summary View displays various call graph metrics:

Performance Metric	Description
Total time	Time measured from a function entry to exit point.
Self time	Total time in a function excluding time spent in its children.
Total wait time	Time spent in a function and its children when the thread is blocked.
Wait time	Time spent on a function when the thread is suspended excluding time spent on the child functions.
Calls	Number of times the function is called.
Class	Class or the COM interface function to which the function belongs.
Callers	Number of caller functions that called the function.
Callees	Number of callee functions the function called.

Using VTune for Threaded Applications

There are two uses of the Intel® VTune™ Performance Analyzer related to threading performance:

- **To improve the threading model:**
  - If your application is single-threaded, the first step in improving scaling is to add multithreading.
  - If your application is already multithreaded, you can use sampling and call graph to improve performance and scaling on a multiprocessor system.

- **To improve the efficiency of computation:**
  - You can identify code regions that have a high impact on application performance by using sampling and call graph data collectors.
  - You can also analyze data in the Call Graph view to determine your program flow and identify the most time-consuming function calls and call sequences.

VTune Analyzer on Serial Applications

You can use the Intel® VTune™ Performance Analyzer on serial applications to:

- Track areas of serial code that might parallelize.
- Determine what parts of the application you can thread to speed up the application.

Three areas of program execution can cause performance slowdowns:

- CPU-bound processes
- Memory-bound processes
- I/O-bound processes

VTune Analyzer on Multithreaded Applications

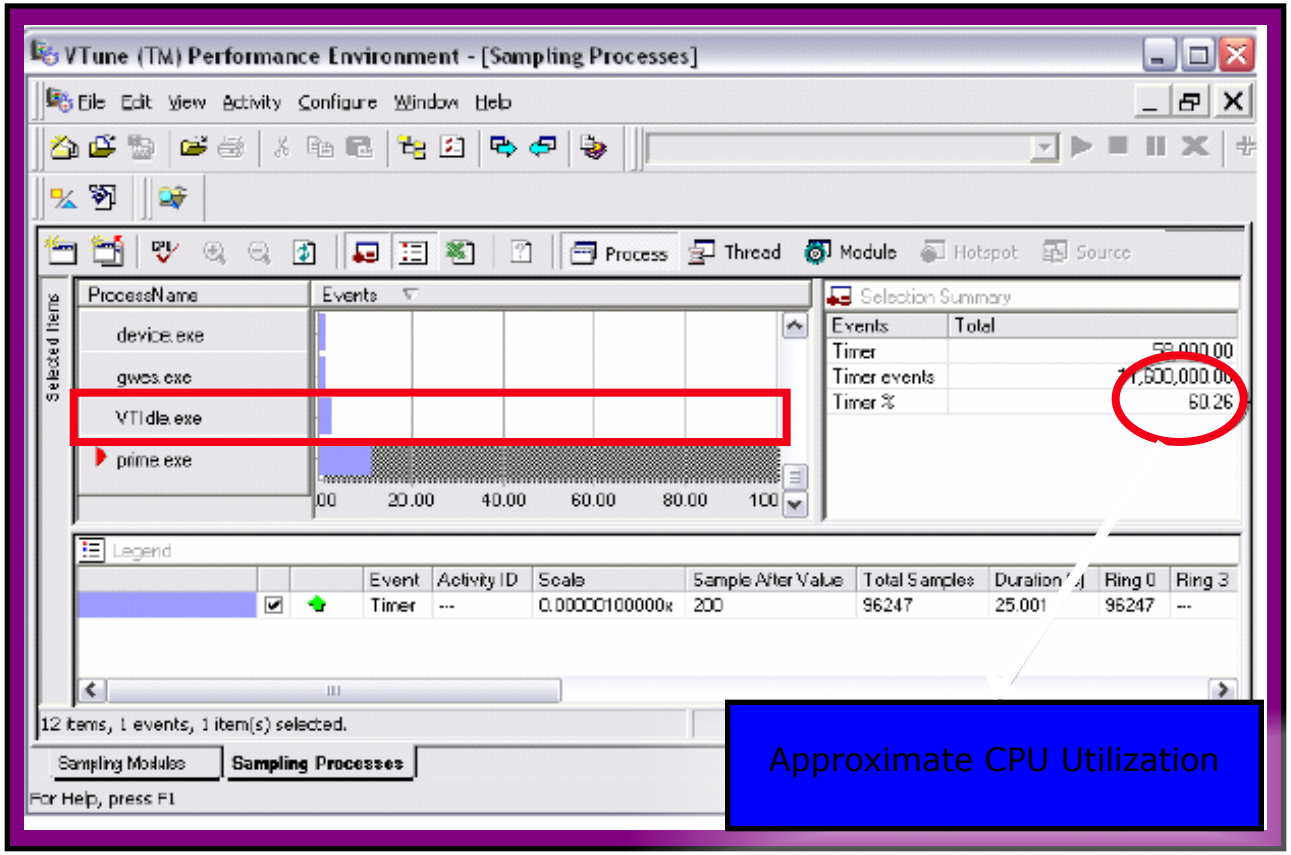
You can use the Intel® VTune™ Performance Analyzer on multithreaded applications to determine if your current threading model is balanced.

You can detect load imbalance in two ways:

- By inspecting the amount of time that threads take to execute by using both sampling and call graph.
- By viewing the CPU information by clicking the CPU button 

on the VTune Performance Analyzer toolbar.

CPU Utilization



Process View of Process During Sampling Results