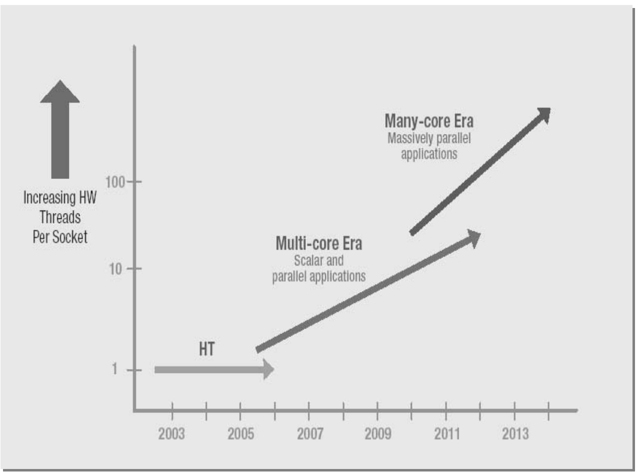
Improving Software Performance and Correctness with Intel® Threading Tools

Intel
Software and Solutions Group (SSG)
Developer Products Division (DPD)

Intel® Processor and Platform Evolution for the Next Decade

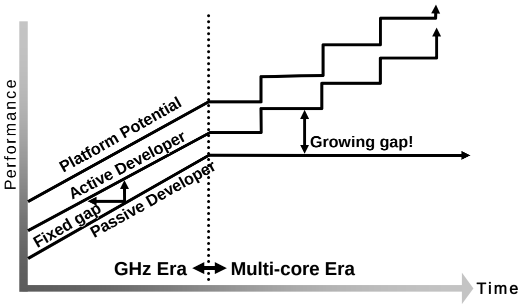


Source: "Platform 2015: Intel® Processor and Platform Evolution for the Next Decade"

2

Paths to taking advantage of Multi-core Processors

- **Do Nothing:**
  - Background tasks benefit from more compute resources
  - Limited potential for single-application performance



- **Process-level parallelism**
  - Can be cumbersome to work on a shared data set
- **Application threading for performance:**
  - Use native threads or threading abstraction libraries
  - OpenMP is a cross-platform standard useful for quickly parallelizing with domain decomposition
  - Intel software tools can aid developer in efficiently threading

3

A Generic Development Cycle

Analyze

–VTune™ Performance Analyzer

Design (Introduce Threads)

–Intel® Performance libraries: IPP and MKL

–OpenMP* (Intel® Compiler)

–Explicit threading (Win32*, Pthreads*)

–Intel® Threading Building Blocks

Debug for correctness

–Intel® Thread Checker

–Intel® Debugger

Tune for performance

–Intel® Thread Profiler

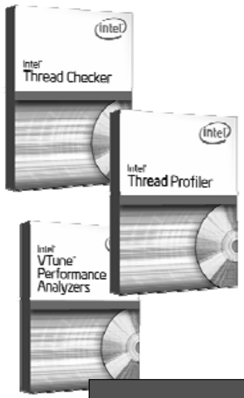
–VTune™ Performance Analyzer

4

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

Multi-core Software Tools

How can we help more developers use parallelism?



Intel® Thread Checker
pinpoints latent threading errors

Intel® Thread Profiler
Insight into threaded application/lock level performance

Intel® VTune™ Performance Analyzer
Insight into system level performance

We help make it easier

Intel® Thread Checker

Create Threads Faster

Key Features

- Detects challenging data races and deadlocks
- Pinpoints errors to the source code line
- Works on standard debug builds without recompiling
- Recommends modules to instrument by usage (windows* product)
- Scriptable interface for test environment integration
- Supports 32 and 64-bit applications

Thread Checker: Overview

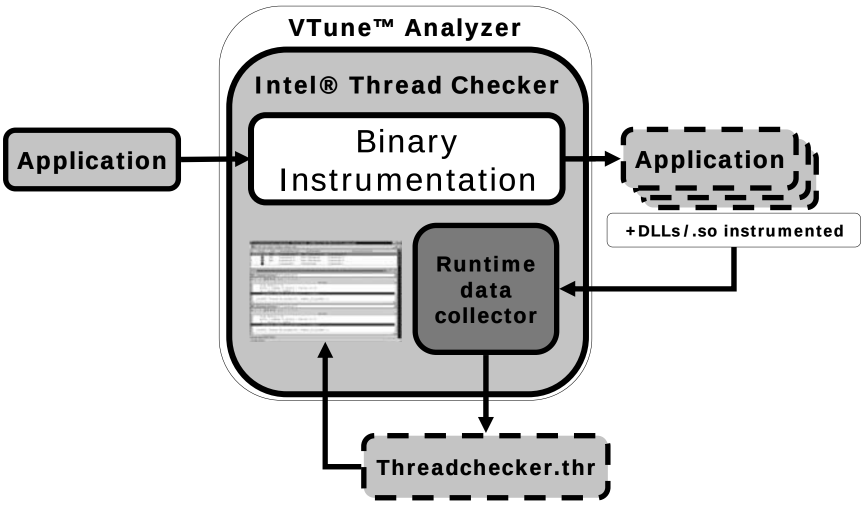
Features & Benefits

- Pinpoint the function, context, line, variable, and call stack in the source code to aid analysis and repair of bugs
- Identify nearly impossible-to-find data races and deadlocks using an advanced error detection engine
- Instrumental for effective design of threaded applications
- Errors do not need to actually occur to be detected

Platforms

- Supports Windows* threads or OpenMP* applications on Windows for IA32/EM64T
- Supports POSIX* threads or OpenMP* for applications on Linux for IA32/EM64T/IPF from a Windows host
- Command-line-only version for Linux is in beta

Thread Checker Phases



Thread Checker Analysis

Dynamic analysis as software runs

- Data (workload)-driven execution
- If code path not executed, no analysis of path

Includes monitoring of:

- Thread and synchronization API's used
- Thread execution order
  - Scheduler impacts results
- Memory accesses between threads

Thread Checker: Before You Start

Instrumentation: Background

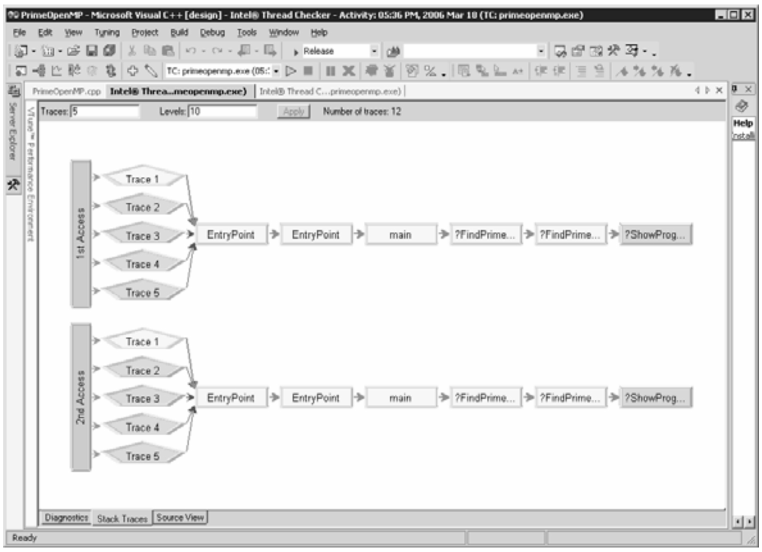
- Adds calls to library to record information
  - Thread and synchronization API's, memory accesses
- Increases execution ***time*** and ***size***

Use *small* data sets (workloads)

- Execution time and space is **expanded**
- Multiple runs over different paths yield best results

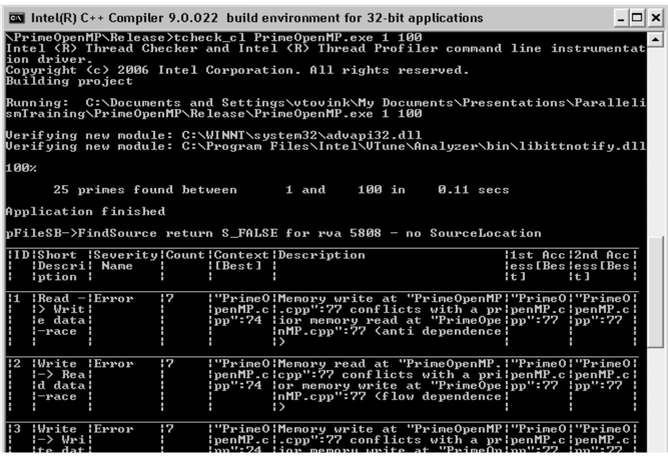
Workload selection is important!

Thread Checker Views



11

Thread Checker Command-Line

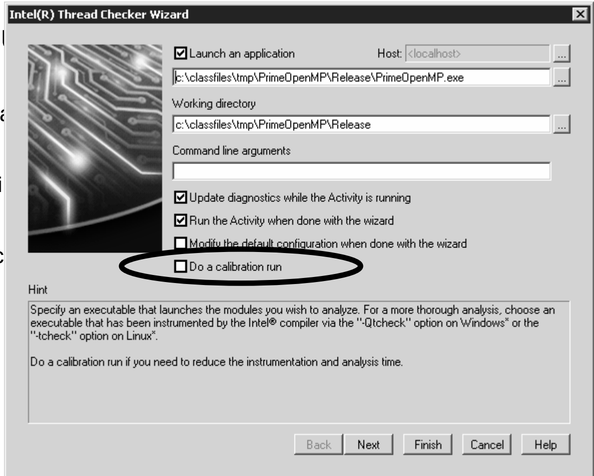


Thread Checker can be integrated into automatic test system

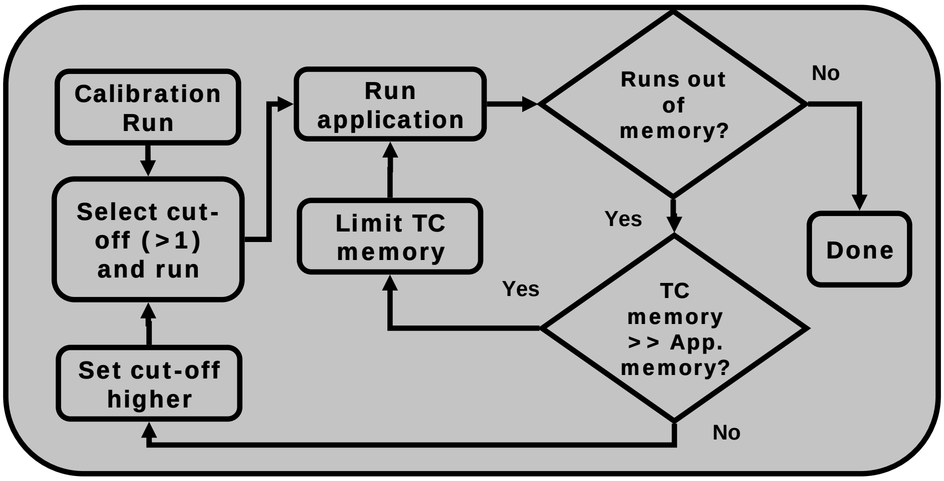
Getting Applications to Run

Solution - U

- **Step 0:**
  - Use sma
- **Step 1:**
  - Use cali
- **Step 2:**
  - Select c



Selective Instrumentation Using Calibration



Dealing with High Diagnostics Count

Where do you begin debugging?

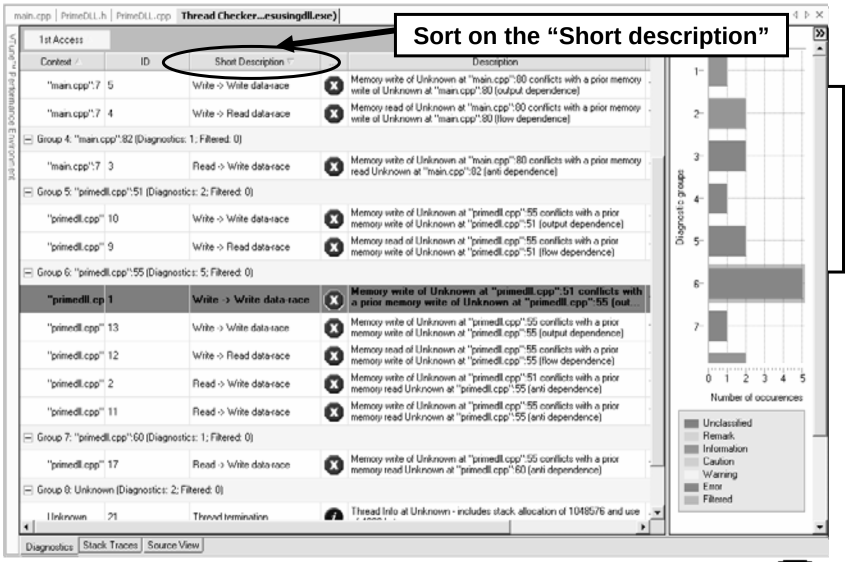
Are all the diagnostic messages equally important/serious?

Steps:

- Add “1st Access” column
- Group by “1st Access”
- Sort by “Short Description” column

15

Dealing with High Diagnostics Count



16

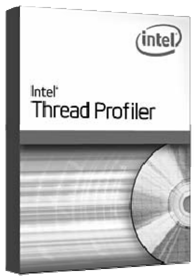
Thread Checker Summary

- Thread Checker functions as a design, debugging, and quality aid
  - Automatically detects hard-to-find data races and deadlocks in multi-threaded applications
- Workload selection is extremely important for successful runs
  - Default configuration works well for unit tests; larger fully integrated applications require the use of calibration run

Reduce time to market for threaded applications by speeding up the development process

Intel® Thread Profiler 3.0 for Windows*

Optimize Threads Faster



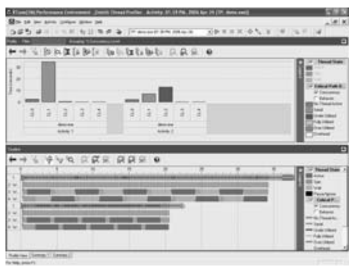
Key Features:

Understand Threading Behavior

- View potential core utilization

Optimize Threading Performance

- Fully utilize available cores
- Identify which synchronization objects are contended and which waits actually affect performance
- Highlight workload imbalance
- Pinpoints issues to the source code

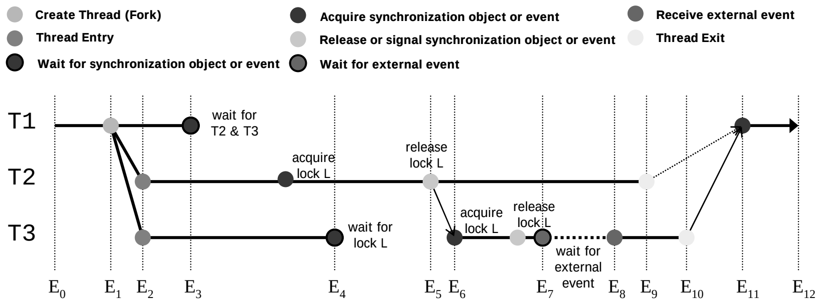


Supported Environments:

- Supports 32 and 64-bit applications
- Native thread-API on Microsoft Windows* (Win32 Threads)
- Native thread-API on Linux* (PThreads)
- OpenMP* threads
- Intel's new parallel programming model
- Graphical visualization available on Windows*

Thread Profiler Execution Flow

- •Thread Profiler instruments system API calls
- •Run-time engine intercepts API calls and records events when instrumented application is running
  - – calling thread ID, time, duration, sync object ID, call-stack
- •In the end of the program run TP data file gets produced
- •Thread Profiler GUI visualizes threads behavior



System APIs – Windows* Threads and POSIX* threads

- Thread and Process Control APIs
  - – Fork, Create, Terminate, Suspend, Resume, Exit
- Synchronization APIs
  - – Mutexes, Critical Sections, Locks, Semaphores, Thread Pools, Timers, Messages, APCs, Events, Condition Variables
- Blocking APIs
  - – Sleeping, Timeouts
  - – I/O: Files, Pipes, Ports, Messages, Network, Sockets
  - – User I/O: Standard, GUI, Dialog Boxes

* POSIX* threads API is supported for Linux apps

User Synchronization

If custom synchronization is used, Thread Profiler provides an API for users to instrument user synchronization.

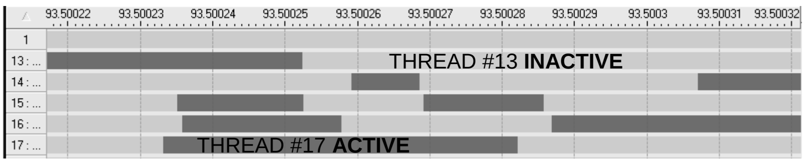
```
__itt_notify_sync_prepare( &spin );
while( wait for spin ) {
  if( timeout ) {
        __itt_notify_sync_cancel( &spin );
        return;
  }
}
__itt_notify_sync_acquired( &spin );

do stuff;

__itt_notify_sync_releasing( &spin );
release spin;
```


Thread Profiler: Terminology

Thread Activity

- Types of thread activity
  - Thread active: runs or ready to run
  - Thread inactive: waits for sync object, blocks on external event
- Thread Profiler view

Time ➔



- Rationale
  - You don't want to have too many threads active and competing for cores
  - You want to keep all cores busy with work when number of active threads equals to number of cores
  - Thread Profiler collects call-stack for wait and blocking APIs – you can locate the problem area in the source file

Transitions

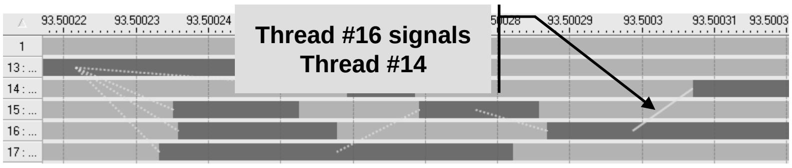
- Represents "signal" sent by one thread to another by releasing synchronization object
- Attributes:
  - Signaling and receiving thread IDs
  - Overhead – time spent between "send" and "receive" events
  - "Send" and "receive" events call-stacks

Time ➔

0: Thread 1 starts	0: Thread 2 starts
1: Thread 1 acquires Critical Section	1: Thread 2 processes private data
2: Thread 1 processes shared data	2: Thread 2 attempts to acquire Critical Section and blocks
3: Thread 1 releases Critical Section	3: Thread 2 acquires Critical Section
4: …	4: Thread 2 processes shared data
	5: Thread 1 releases Critical Section

Transitions (cont.)

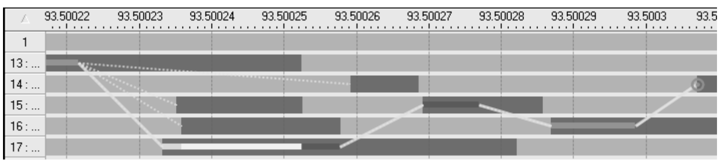
- Types of transitions
  - Contended: receiving thread had to wait for the signal
  - Uncontended: thread acquired the sync object without contention
- Thread Profiler view



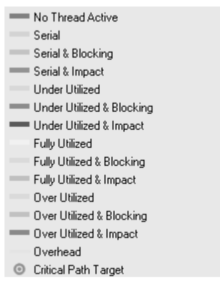
- Rationale
  - Thread Profiler collects call-stack for transitions – you can locate the problem area in the source file
  - Thread Profiler attributes transition overhead to the specific sync object – you can find the most expensive sync objects
  - Thread Profiler visualize transitions – you can spot the area with the most excessive synchronization and focus your analysis effort on it

Critical Path

- Definition
  - Longest execution flow; emphasizes segments of the threaded program that are worth optimizing
  - Characterized behavior of active threads
- Thread Profiler view

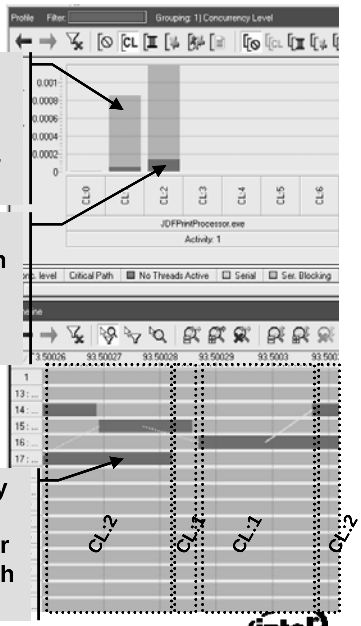


- Rationale
  - Useful for throughput analysis
  - Helps understand threads behavior



Concurrency Level (CL)

- Definition
  - Number of active threads
- Rationale
  - Thread Profiler summarizes threads activity information and groups it by concurrency levels – you can understand overall CPUs utilization
  - Thread Profiler Timeline and Profile view are synchronized – you can zoom-in and filter data and get from summary view into detailed view
  - CL grouping (like any other view) may be combined with critical path data – you can view threaded behavior at a higher level

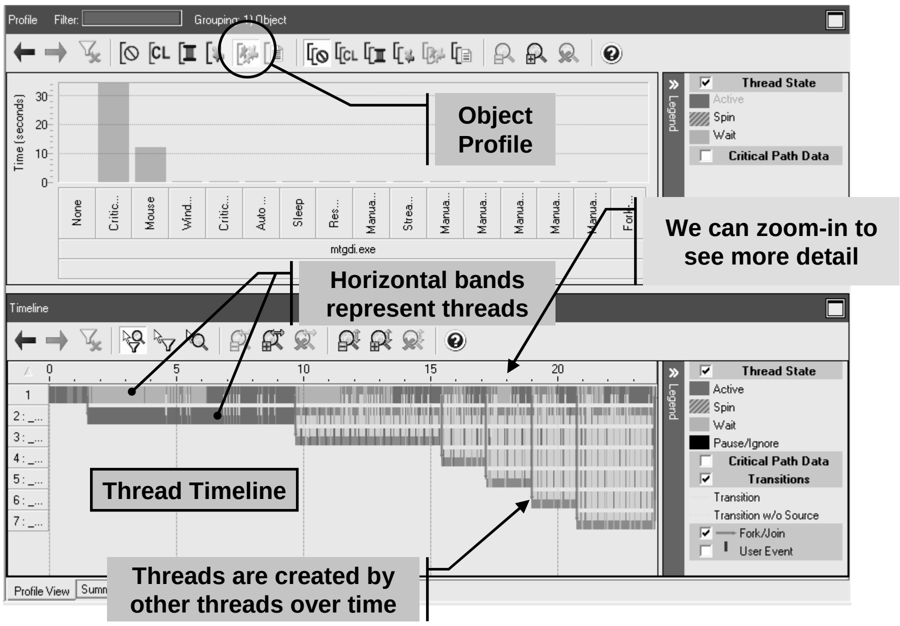


Groupings and Filtering

- Profile View
  - Grouping by Concurrency Level
  - Grouping by Threads
  - Grouping by Objects
  - Grouping by Source Locations
- Timeline View
  - Manual filtering of any region of interest
  - Synchronized automatic filtering: double-click on any bar at Profile View and Timeline data gets filtered

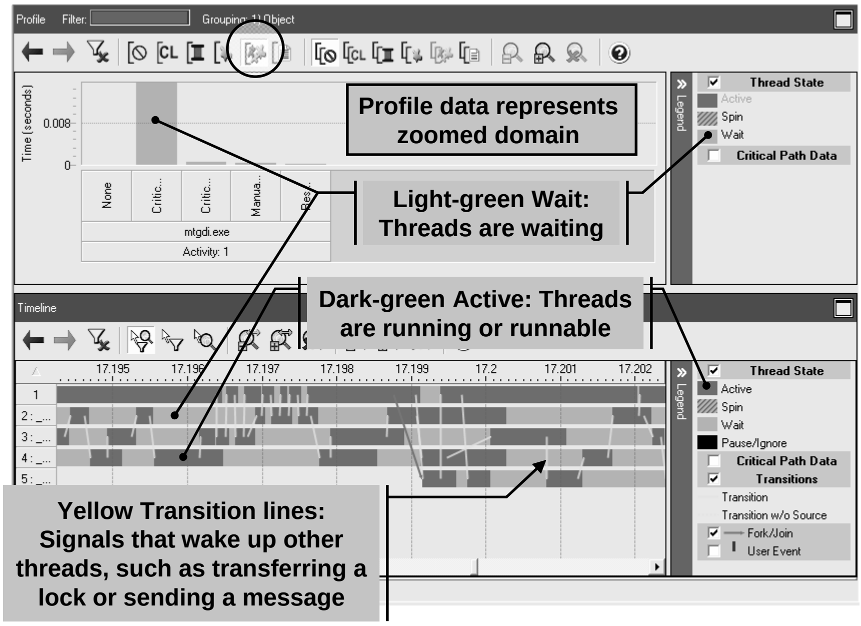
Understand Threading Behavior

Object Profile and Thread Timeline



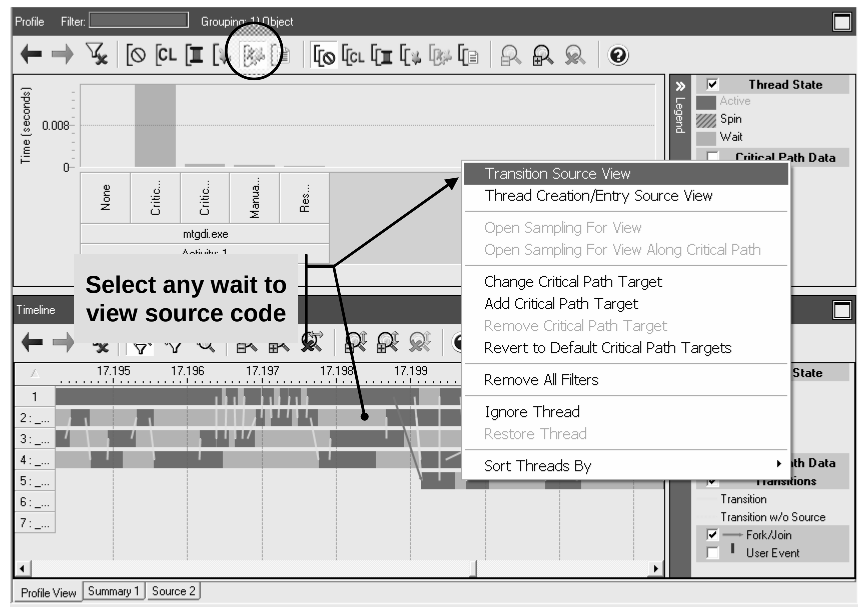
30

Zoom-in for more detail



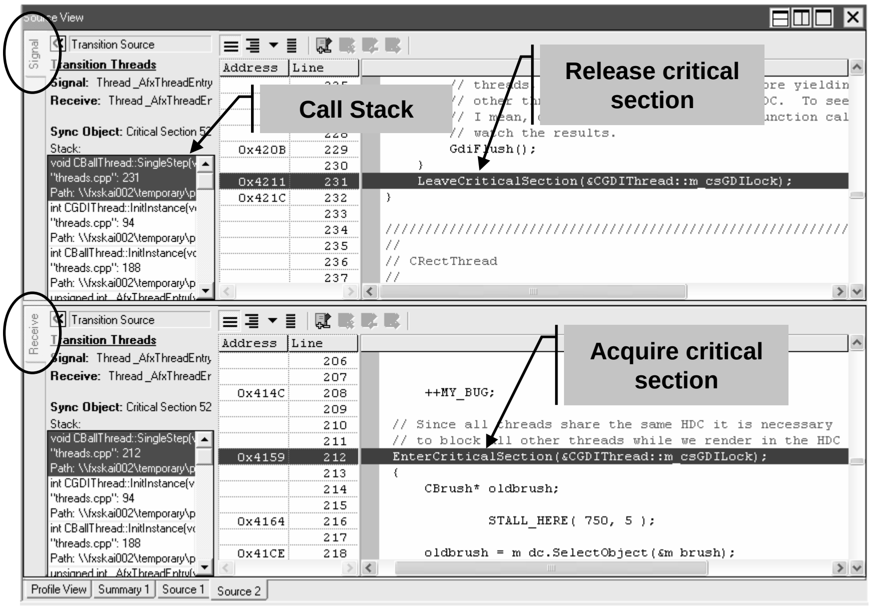
31

View source for any thread wait



32

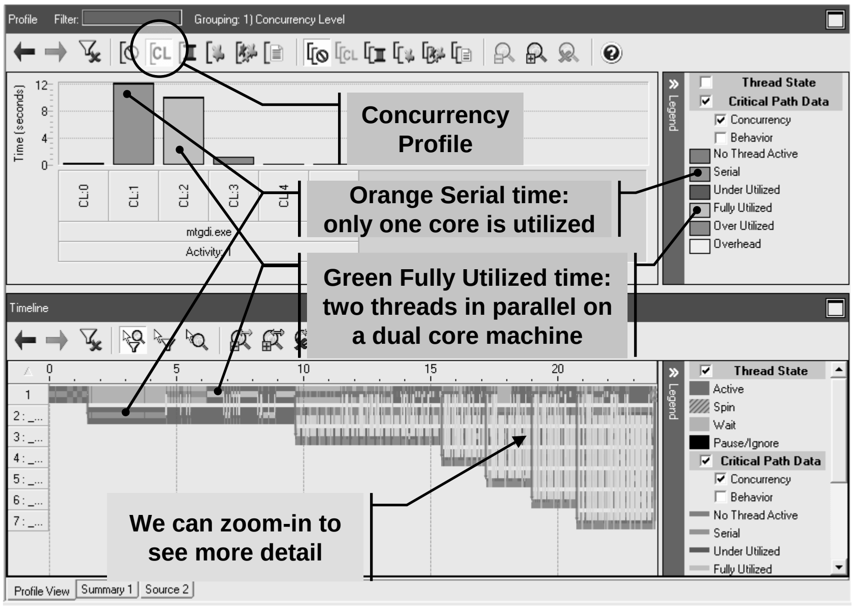
Synchronization source view



33

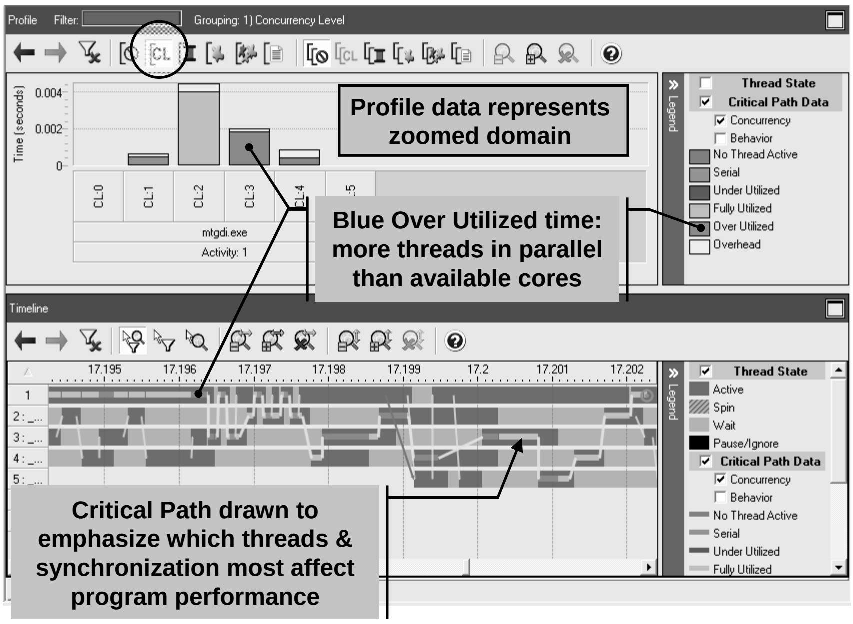
Optimize
Threading Performance

Concurrency Profile and Thread Timeline



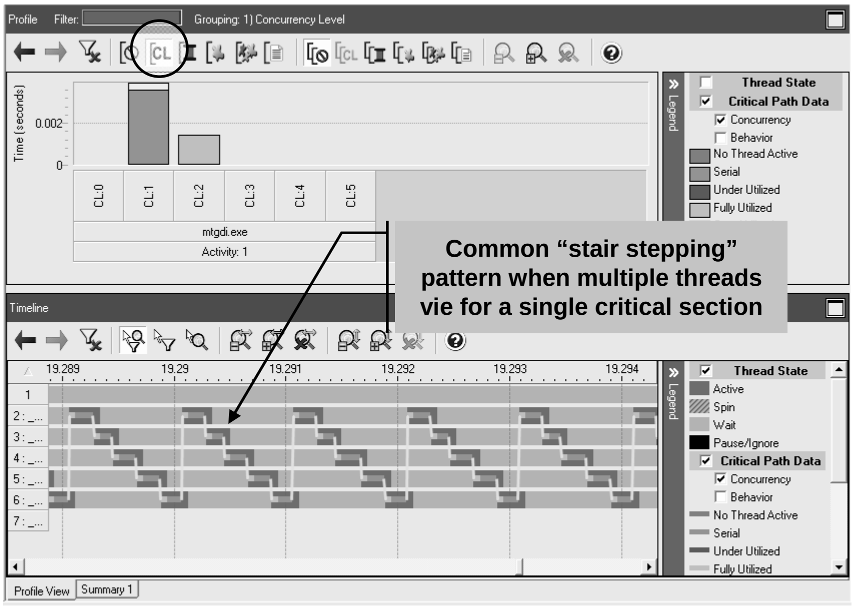
35

Zoom-in for more detail



36

Example lock “stair stepping” pattern

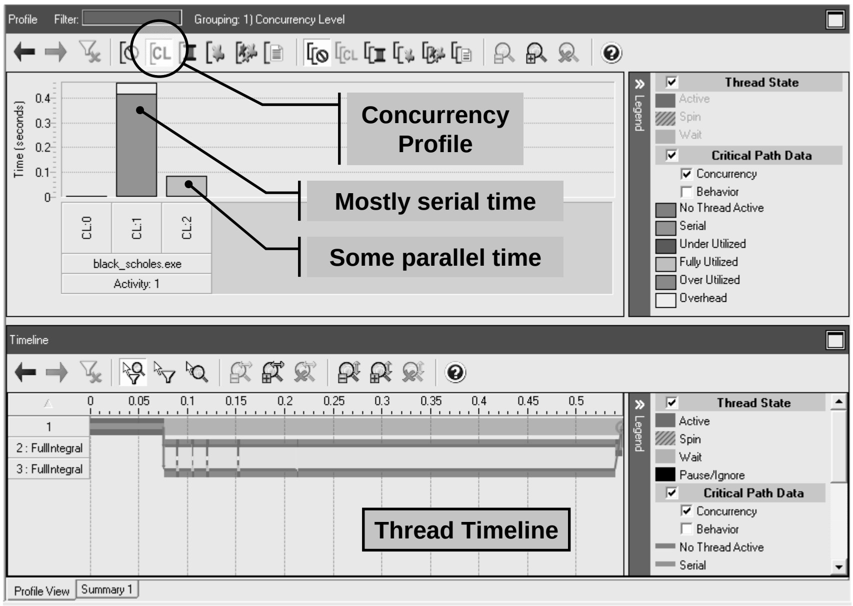


37

Example Performance Analysis

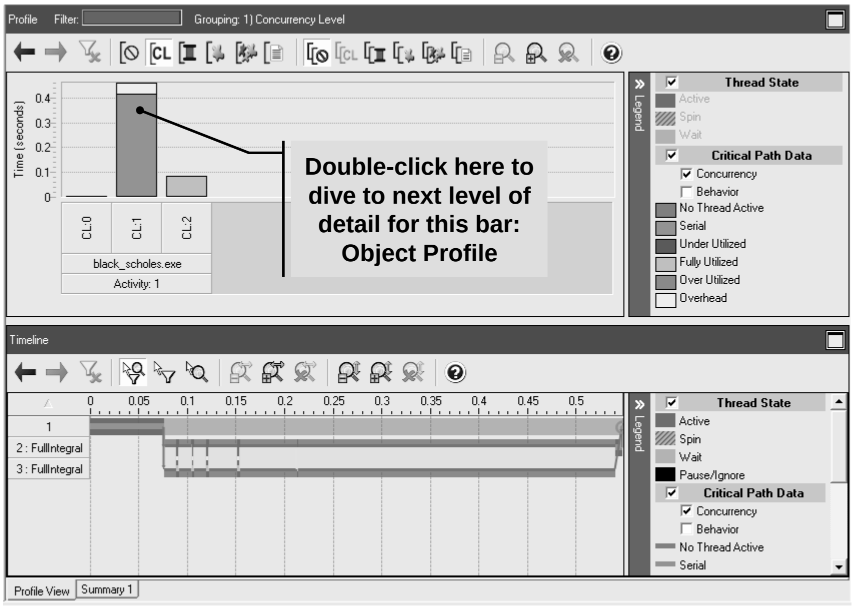
Numerical Simulation

View initial Thread Profiler results



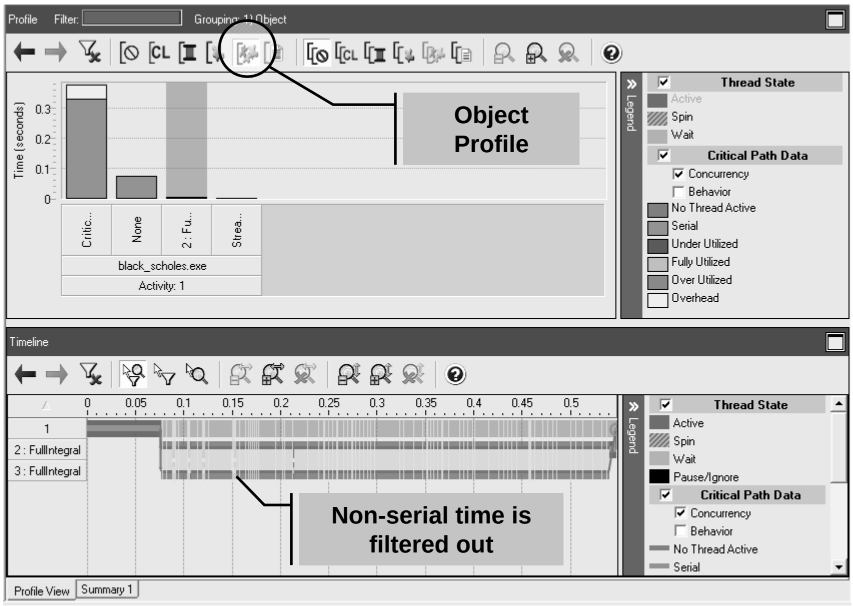
39

Dive for more detail about serial time



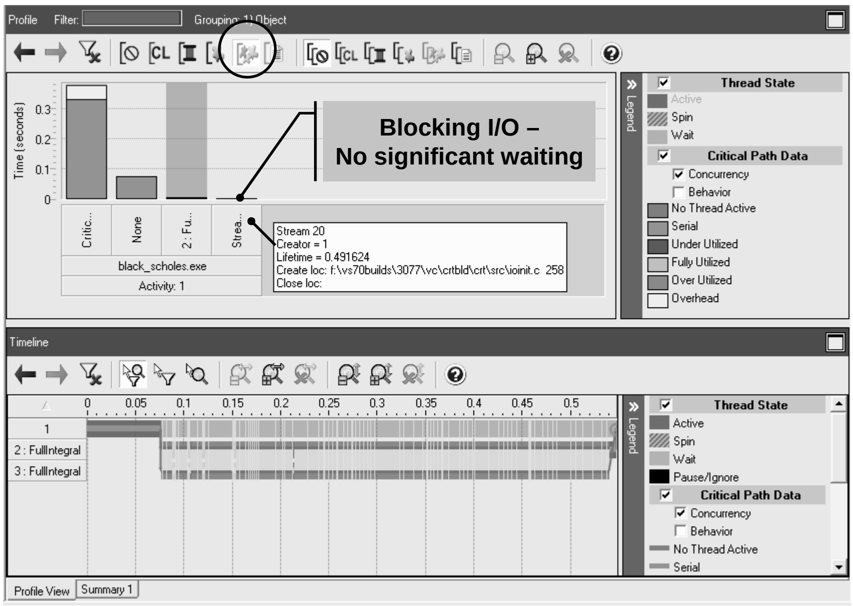
40

Object Profile for serial time



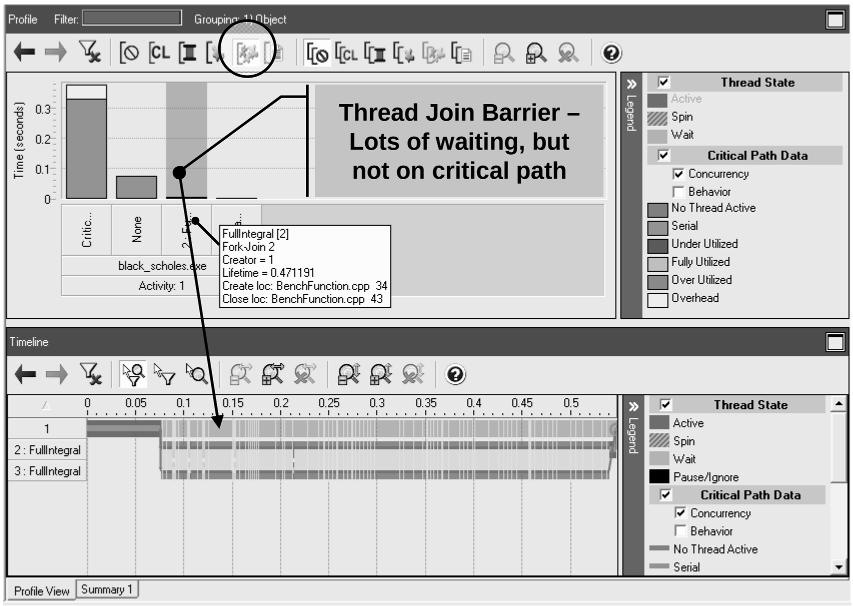
41

Object Profile – Blocking I/O



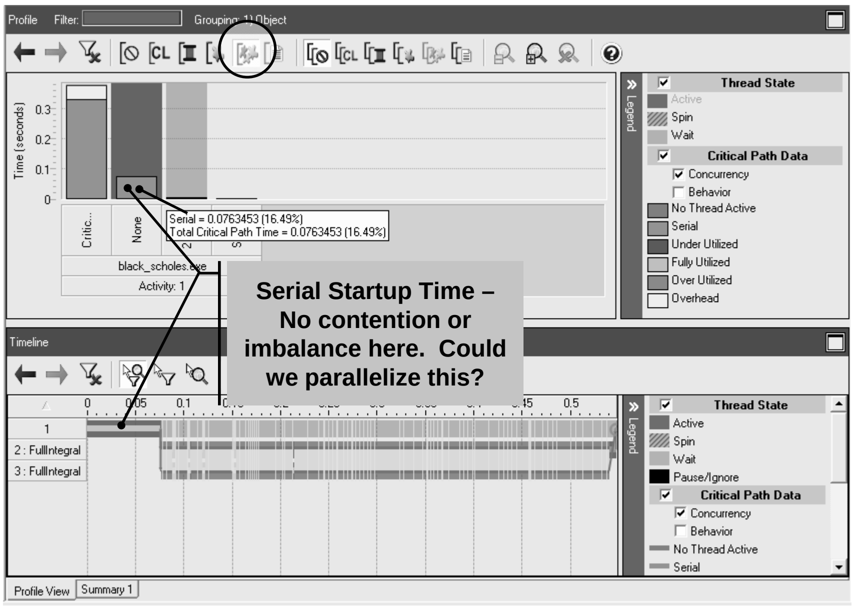
42

Object Profile – Thread Join Barrier



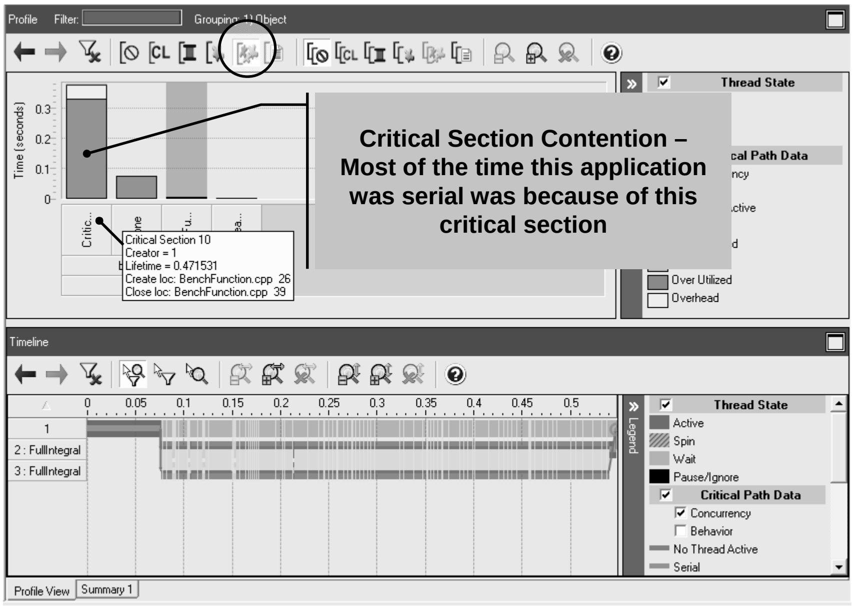
43

Object Profile – Serial Startup



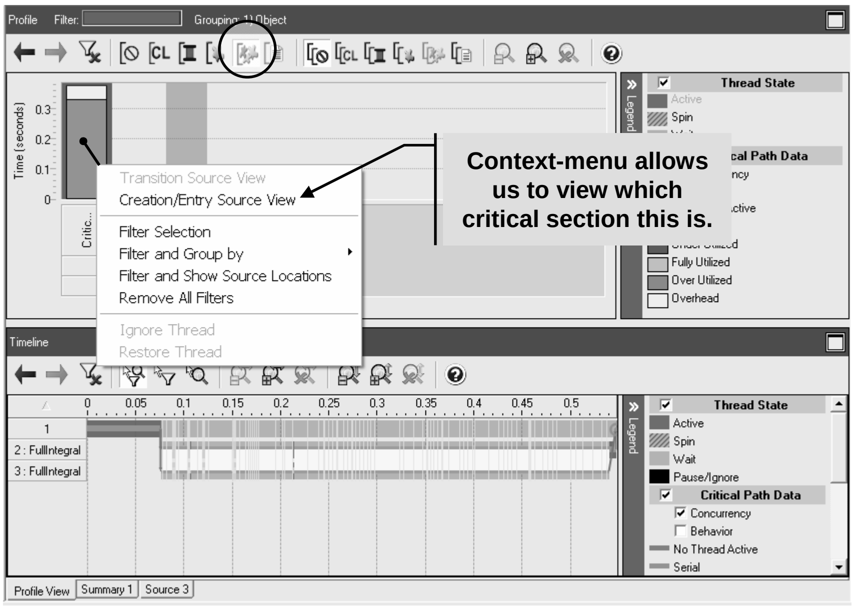
44

Object Profile – Critical Section



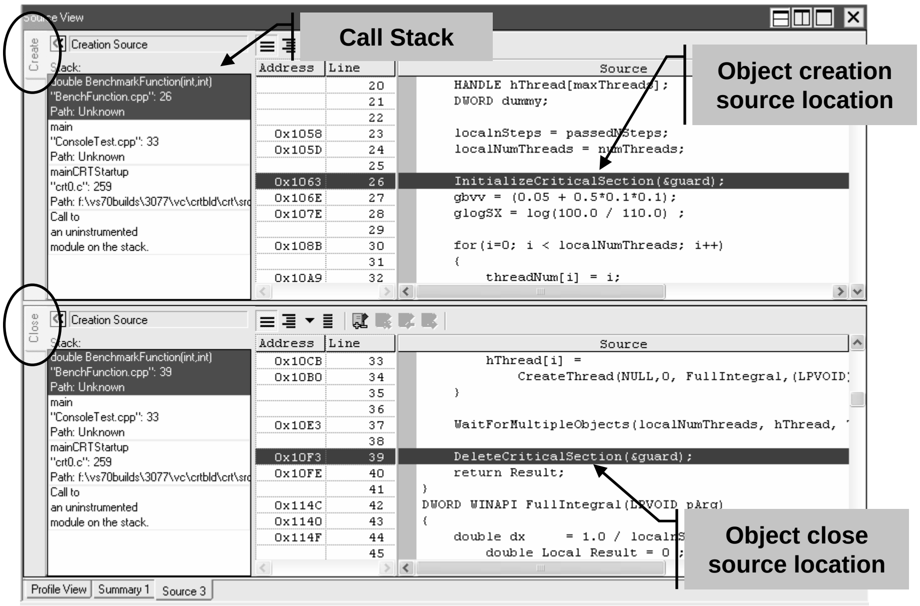
45

View source for object creation point



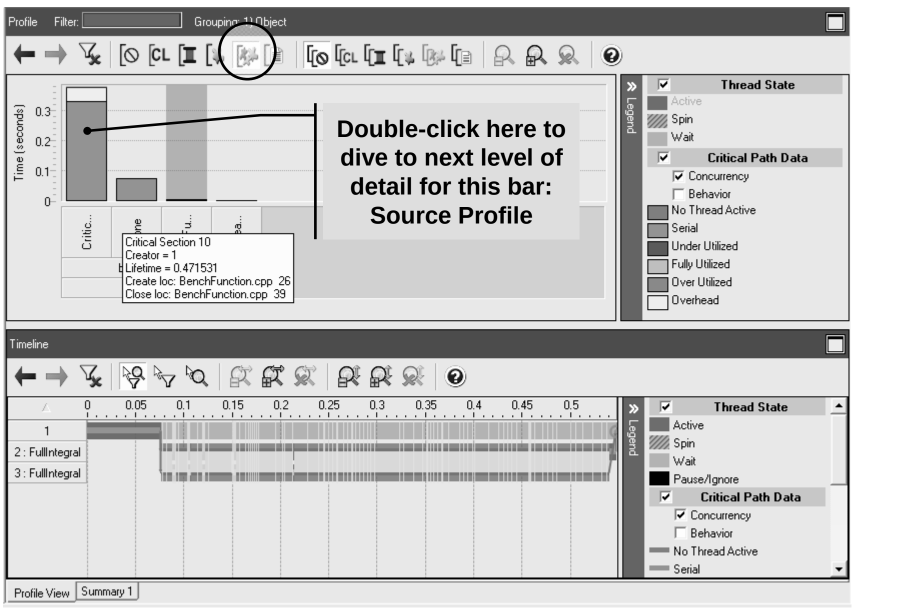
46

Object create / close source view



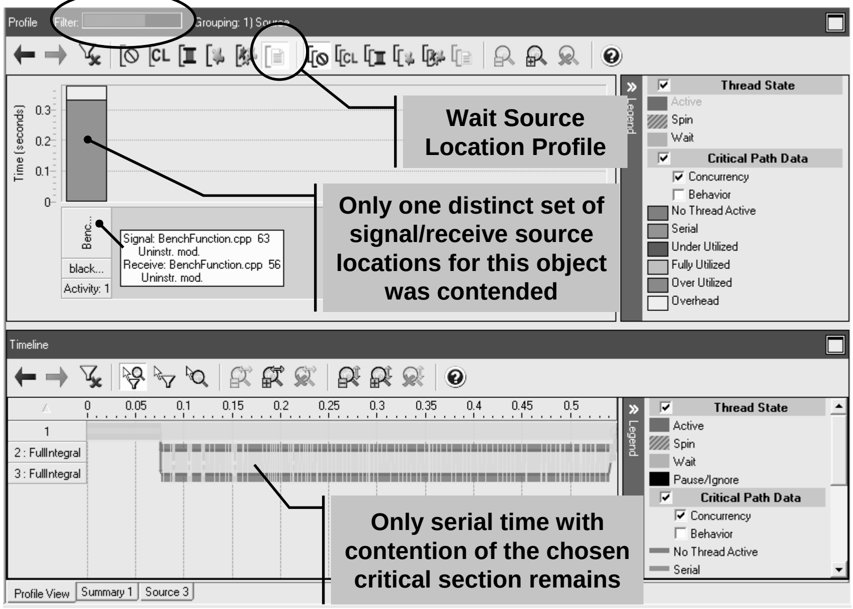
47

Dive for more detail about this object



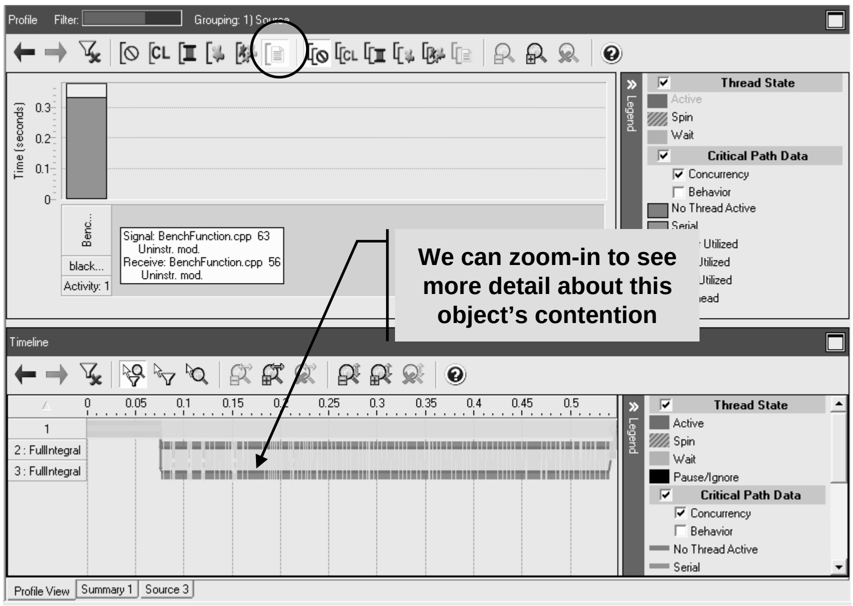
48

Wait source location profile



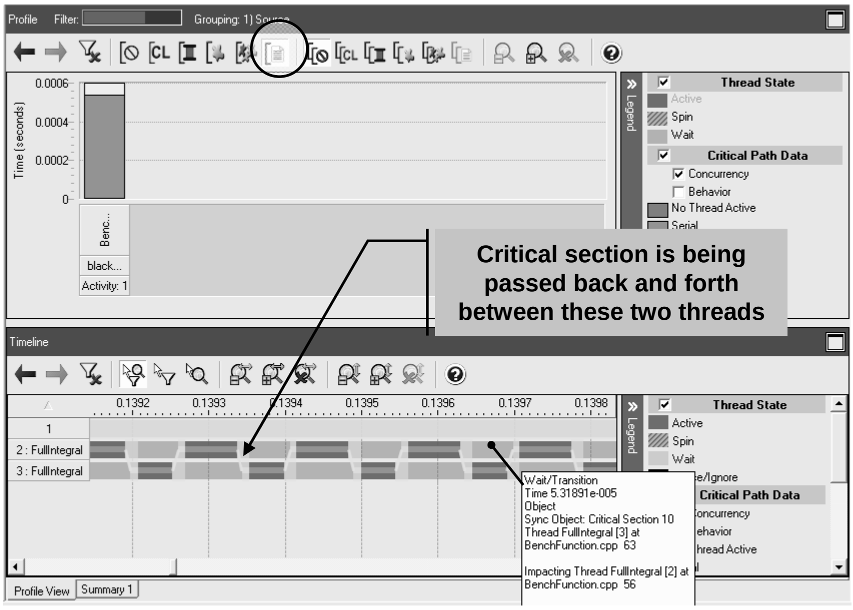
49

Zoom-in for more detail at any time



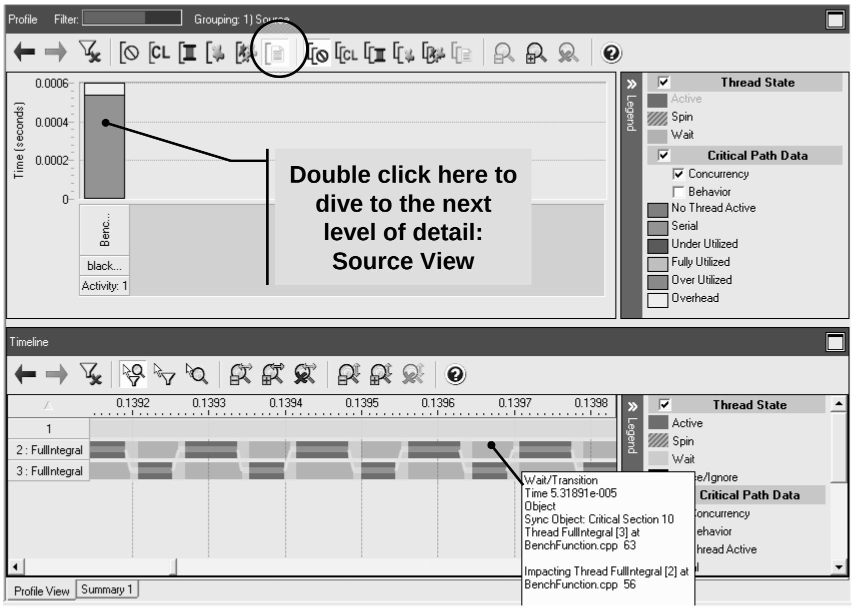
50

Critical Section is causing serialization



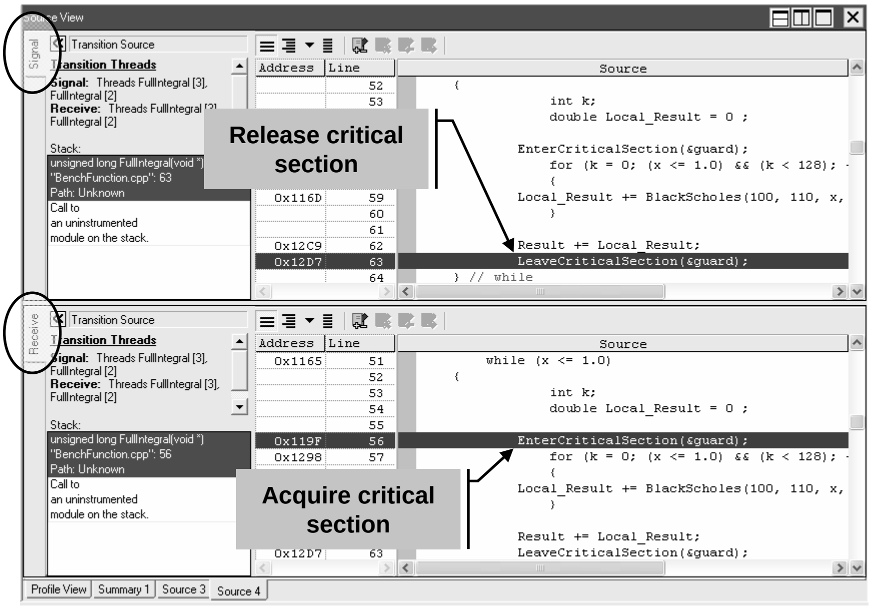
51

Dive for source view of this location



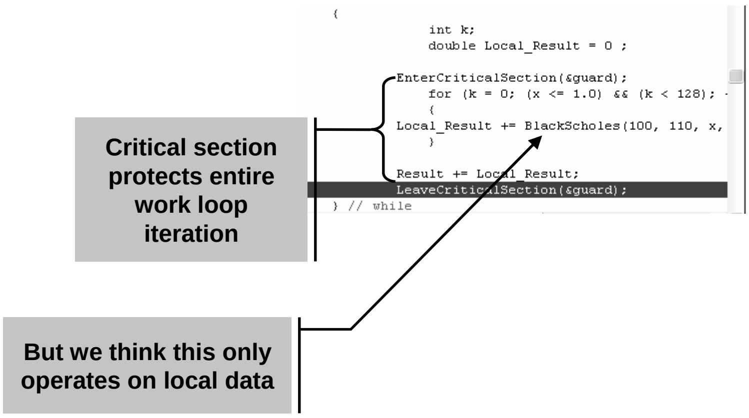
52

Synchronization source view



53

Analyze this critical section



54

Optimize this critical section

```
{
        int k;
        double Local_Result = 0 ;

    EnterCriticalSection(&guard);
        for (k = 0; (x <= 1.0) && (k < 128); ·
        {
    Local_Result += BlackScholes(100, 110, x,
        }

    Result += Local_Result;
    LeaveCriticalSection(&guard);
} // while
```

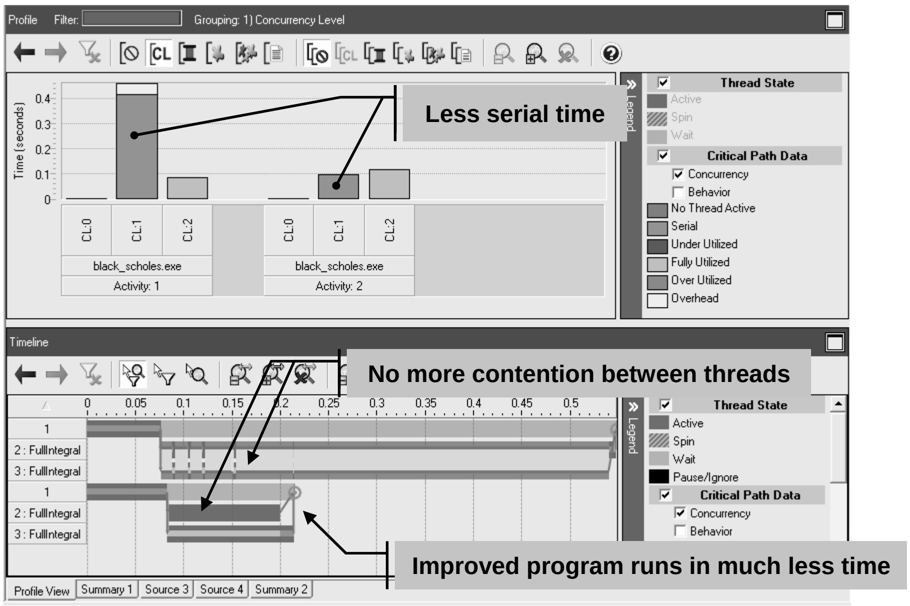
Let's reduce the size of the critical section so it only protects the reduction variable

```
{
        int k;
        double Local_Result = 0 ;

        for (k = 0; (x <= 1.0) && (k < 128); ·
        {
    Local_Result += BlackScholes(100, 110, x,
        }
    EnterCriticalSection(&guard);
    Result += Local_Result;
    LeaveCriticalSection(&guard);
} // while
```

Verify the correctness of this assumption with Thread Checker

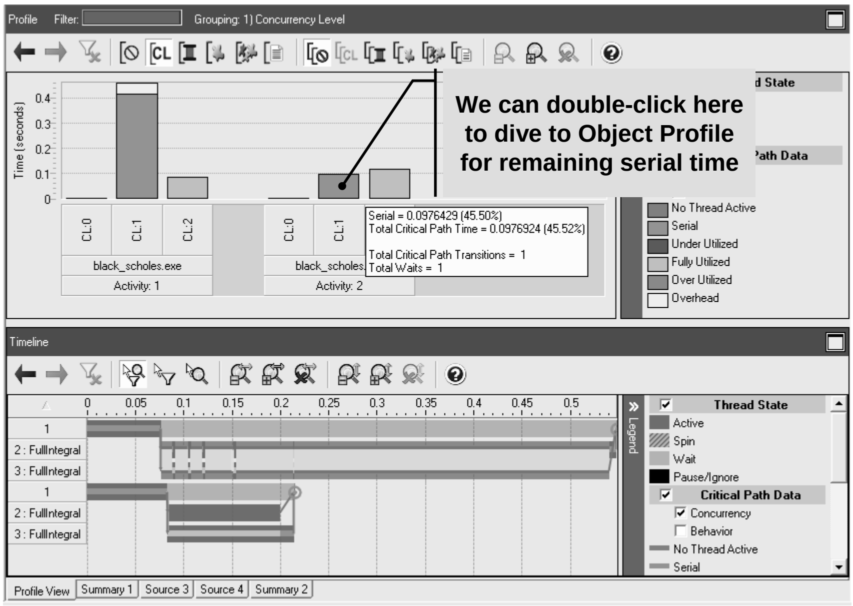
55

Compare results of new program run



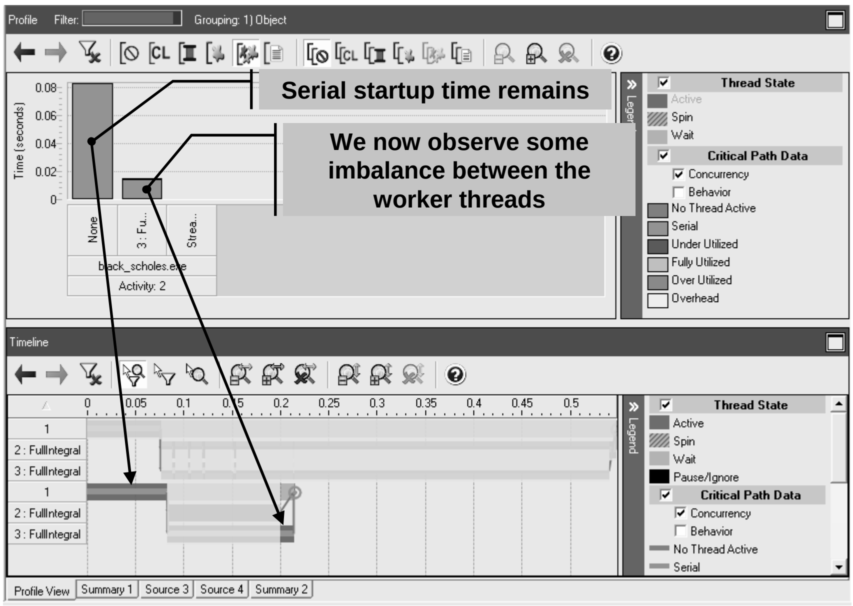
56

Dive to see what serial time is left



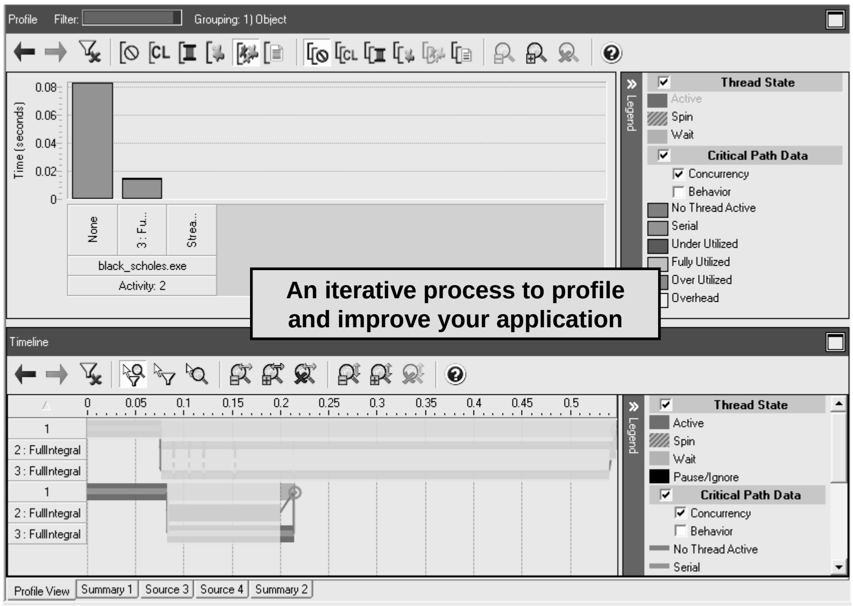
57

Object profile for serial time



58

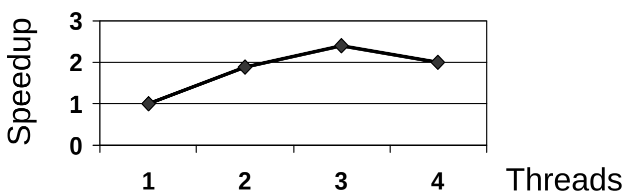
Iterative process



59

Different causes of poor scalability

A tool could help determine the actual cause



1. Insufficient parallel work
2. Synchronization overhead
3. Contention
4. Load imbalance
5. Task granularity
6. Memory bandwidth / false sharing
7. ...

Thread Profiler can help you detect many issues that limit scalability

60

61