

Debugging Parallel Programs

Arch D. Robison
Intel Corporation

Slides on Thread Checker and Thread Profiler courtesy of Paul Petersen

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

All dates provided are subject to change without notice.

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

Copyright © 2005, Intel Corporation.

Outline

- This is about what I have used in practice
 - Much good theoretical work ignored
- The problem(s)
- Serial techniques that become more important
- Trace logs
- Intel[®] Threading Tools
 - Intel[®] Thread Checker and Intel[®] Thread Profiler

Except for these, this talk could have been given in the 1980s!

May You Live in Interesting Times

- Parallel processors used to be special-purpose machines
- Now multicore chips are making them ubiquitous.
- Debugging parallel programs is “interesting”
 - Events in parallel program are partially ordered
 - Different interleavings can occur on each run
 - Replay not generally supported

Heisenbugs

- Adding instrumentation (assertions, tracing) to program can change relative thread speeds
 - Possibly hide or expose race conditions.
 - Extremely frustrating when it happens
 - Try a different machine or optimization level until you can “shake loose” the bug.
 - Use tool like Intel® Thread Checker if possible
 - Finds *potential* races that actually did not happen in a particular run.

Race Conditions

- These are the primary correctness headache.
- Example:

Initial State

```
int X = 0;
```

Thread 1

```
t1 = X  
t1 = t1 + 1  
X = t1
```

Thread 2

```
t2 = X  
t2 = t2 + 2  
X = t2
```

Final State

```
X ∈ {1, 2, 3}
```

Somewhat Less Obvious

Thread 1
++X

Thread 2
X+=2

Hidden

Thread 1
OtherProgrammersRoutine(1);

Thread 2
OtherProgrammersRoutine(2);

Data Dependent

Thread 1
++X[i];

Thread 2
++X[j];

Higher-Level Race Conditions

- Composing thread-safe operations does not guarantee that result is thread-safe.
- Example: implementing a set using a thread-safe list
 - Invariant is “each key occurs once in list”

```
void add_if_not_present(key) {  
    if( !list.contains(key) )  
        list.append(key);  
}
```

Between the time we check for the key and insert it, another thread might append the same key.

Remember to lock outermost invariant.
Locks in inside levels just waste time.

This is a challenge for reusable components.

A Simple Trick

- If you are not sure if a serial loop can be parallelized, try reversing the iteration order.
 - Different result \Rightarrow there is definitely a dependence that prevents trivial parallelization.
 - Same result \Rightarrow no guarantees, but at least situation is not hopeless.

Memory Consistency

- Be very careful about accessing shared memory that is unprotected by a mutex.
 - Hardware can reorder accesses
 - Compiler can reorder accesses
 - volatile keyword in C/C++ is *not* sufficient (except on Itanium processors)
 - volatile keyword in C# (and recent Java*) is sufficient (more or less)
- Definitions
 - sequential consistency
 - All operations happened as if in a particular total order
 - All observers agree on the order
 - relaxed consistency
 - Observers can disagree.
 - Enables significant hardware and compiler optimizations
 - Breaks Dekker's algorithm ☹

Example

```
// Initial state  
volatile int Message=0;  
volatile bool Flag=false;
```

```
// Thread 1  
Message = 42;  
Flag = true;
```

If writes are reordered,
then Flag is set prematurely.

```
// Thread 2  
while( !Flag )  
    continue;  
if( Message !=42 )  
    abort();
```

If reads are reordered,
Thread 2 uses out of date
value of Message.

Fence Instructions

- When writing lockless code, use fence instructions to prevent reordering
 - Mutexes typically imply these fences
 - So do not worry if using mutexes properly.
 - Do worry if “rolling your own” synchronization primitives
- Formal definitions of reordering rules are often hard to understand.
- Good rule of thumb:
 - Think of communicating via shared memory as sending and receiving messages.
 - Sender: write message; store fence; write flag
 - Receiver: read flag; load fence; read message

Double Check Idiom

- Used to lazily initialize a resource
 - First thread that needs resource initializes it.
 - Other threads wait on initialization
- Key feature
 - Checks “is ready” flag twice
 - Once outside critical section
 - Once inside critical section
 - Avoids acquiring lock in common case that resource is already initialized.

Infamous Double Check (Done Correctly)

```
// Initial global state  
T* volatile Flag = NULL;  
volatile T Message;
```

Alas, there is no portable way to write this in C++.

```
// Double-check idiom
```

```
T* f = Flag;  
_asm lfence;  
if( f==NULL ) {  
    acquire lock  
    if( Flag==NULL ) {  
        Message = ...;  
        _asm sfence;  
        Flag = &Message;  
    }  
    release lock  
}  
...= *f // read Message
```

lfence ensures that Flag is read before Message on all execution paths.

sfence ensures that Message is written before Flag.

sfence unnecessary on IA-32 if “non-temporal” stores are not employed.

Infamous Double Check (Itanium® Processor or .NET*)

```
// Initial global state  
T* volatile Flag = NULL;  
volatile T Message;
```

Memory operations not allowed to move up past volatile read (acquire).

```
// Double-check idiom  
if( Flag==NULL ) {  
    ▼ acquire lock  
    if( Flag==NULL ) {  
        Message = ...; ▲  
        Flag = &Message;  
    }  
    release lock  
}  
...= *f // read Message
```

Memory operations not allowed to move down past volatile write.(release)

References on Memory Consistency

“Shared Memory Consistency Models: A Tutorial”, Sarita V. Adve and Kouros Gharachorloo,
<ftp://gatekeeper.dec.com/pub/DEC/WRL/research-reports/WRL-TR-95.7.pdf>

<http://www.cs.umd.edu/~pugh/java/memoryModel/>

Java* memory model - recently repaired.

“Memory Ordering”, Section 7.2 of IA-32 Intel® Architecture Software Developer’s Manual
Volume 3: System Programming Guide

<http://www.intel.com/design/mobile/manuals/243192.htm>

C++ and the Perils of Double-Checked Locking, Scott Meyers
and Andrei Alexandrescu,

http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf

Discusses why portable
double-check idiom is
impossible in current C/C++.

A Formal Specification of Intel® Itanium® Processor Family Memory Ordering

<http://www.intel.com/design/itanium/downloads/251429.htm>

“Memory Consistency & .NET”, Arch D Robison,
Dr. Dobb’s Journal*, April 2003.

Itanium® architecture and ECMA .NET* have
the most relaxed memory consistency in the
industry. Understand them, and you have a
solid grip on the issues.

Get Serial Version Right First

- Get sequential version right first
 - If total order does not work, partial order certainly won't!
 - Design program so that there *is* a sequential version
 - Counter example: some producer-consumer programs
- Peter Principle for Programmers
 - Programs rise to level of their programmer's incompetence.
 - Forced by economic competition
 - Creeping featurism etc.
 - Reserve mental capital for dealing with parallel issues
 - Keep code squeaky clean and easy to understand
 - Use memory bounds and leak checkers on serial version
 - Or use a type-safe garbage collected language
 - Check error codes or use exceptions

Unit Testing

- Assembling untested parts is disaster
- Doing so in parallel program is worse
- Test smallest components possible
 - Semi-exhaustive tests
 - Corner cases
 - Regression tests for reported errors
- Test subassemblies too, at each level of integration.
- Test with multiple threads

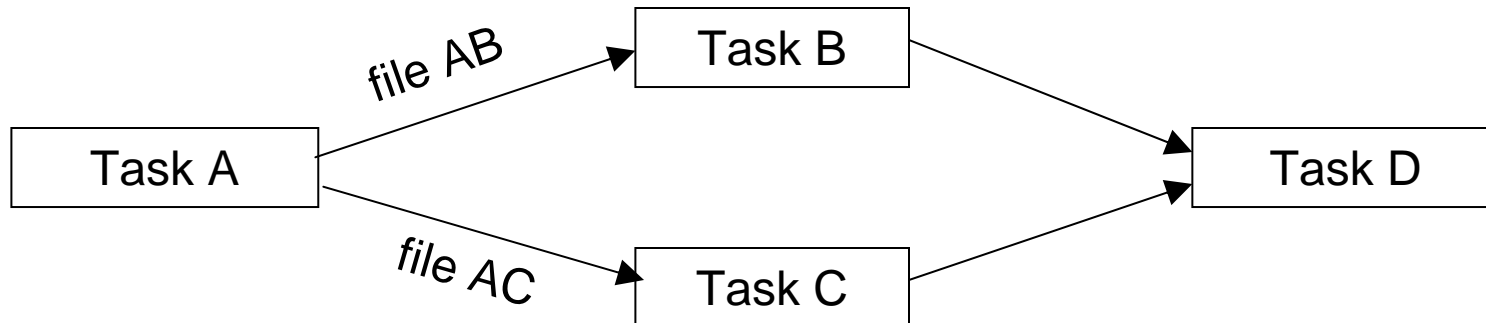
Code Reviews

- Different people see things different ways
- If you cannot explain how your code works
 - It may be wrong.
 - It certainly cannot be maintained by others

Design Technique That Works Where Applicable

- *If* program can be broken down into graph where:
 - Each thread is a vertex
 - Threads communicate only by message passing
- *Then*
 - Provide alternate communication layer where threads can read messages from input file(s), and write messages to output file(s).
 - Debug each thread in topological sort order.

Example



1. Debug A in serial debugger. Now we have files with input messages for B and C.
2. Debug B and C separately in serial debugger.
3. Debug D last.

A sometimes forgotten theorem:

A message-passing program with no “wildcard receives” is deterministic. Consider the gain/pain before you introduce a “wildcard receive”.

Generalization

- If at all practical, build in a means for replay.
 - Replay must force each non-deterministic action to be the same as before.
 - Not too hard for message passing
 - Expensive for shared memory

Keep Track of Your Dependences

- Write squeaky-clean readable code
- Minimize use of shared mutable state
 - Document dependences that you cannot avoid
- Protect hidden shared state with a lock
 - If you give client “separate objects”, it ought to be safe to use them as if they really are separate objects.
 - Example: use atomic reference counting if doing copy-on-write
- Document thread safety and non-safety
- Typical convention for object-oriented programming
 - Instance methods on the same instance are not thread safe
 - Static methods are thread safe
 - Unless otherwise specified (e.g. a concurrent hash table)

Model Your Algorithm

- Formal model checkers have become very effective
 - You describe algorithm as finite-state machine
 - Can find most errors for small N
 - Explores all or random subset of possible interleavings
 - Theory allows shortcuts – do not have to actually enumerate entire space.
- SPIN is very effective tool for modeling concurrent algorithms.
 - C-like syntax makes it easy to write the models
 - <http://spinroot.com>
 - It's free

Spin Excerpt

```
inline writerLock(i)
{
    printf("writer %d: entered writerLock\n", i);
    pointer pred;
    fetch_and_store(pred,tail,i);
    if
    :: pred!=NIL ->
        if
        :: (pred&FLAG) ->
            pred = pred - FLAG;
        :: else -> skip
        fi;
        d_step {
            assert( live[pred] );
            assert( next[pred]==NIL );
            next[pred] = i;
        }
        printf("writer %d: waiting the lock\n", i);
        (going[i]==1)
    :: else -> skip
    fi;
    number_of_writers_in_critical_region++;
    printf("writer %d acquired with pred=%d\n",i,pred);
}
```

Language is hybrid of Dijkstra's guarded command language and C.

Assertions

- Assertions state invariants
- Locks protect invariants
- Therefore, use pre and post conditions in critical sections.

```
acquire lock;  
assert( precondition );  
... do update ...  
assert( postcondition );  
release lock;
```

Not a Panacea

- If a thread forgets to acquire lock, it may trash the invariant between the times it was checked.

Thread 1

```
acquire lock;  
assert( precondition );  
  
... do update ...  
assert( postcondition );  
release lock;
```

Forgetful Thread 2

```
assert( precondition );  
... do update ...  
assert( postcondition );
```

postcondition of correct thread fails

Do Not Despair

- At least assertion caught fact that something is wrong.
- A tool like Intel® Thread Checker can catch the lack of synchronization between writes in the update and the reads in the assertions.
- Another defense
 - Encode some of the information redundantly
 - Example: store values x and $\sim x$ in separate locations, and check that locations are complements.

Interactive Debuggers

- Most debuggers these days have way to switch between threads
- Great for inspect program state after assertion failure.
- Two problems:
 - The non-failing threads may make a little progress before debugger stops them, which may “cover the tracks” of the erroneous thread.
 - The debugger may perturb thread scheduling to the point that the program starts working.

Event Tracing

- Record events so that they can be played back later
 - Try to minimize perturbation of the program
 - No perfect solution
- Event record should contain:
 - Thread id
 - Values of interest
 - Enough information to show the events in temporal order

Example: Fast Circular Buffer With Atomic Index

```
static unsigned long EventCount;

static const long N_Event = 1<<20; // Must be power of two

struct Event {
    int thread_id;
    const char* what;
    int arg1;
};

static Event EventArray[N_Event];

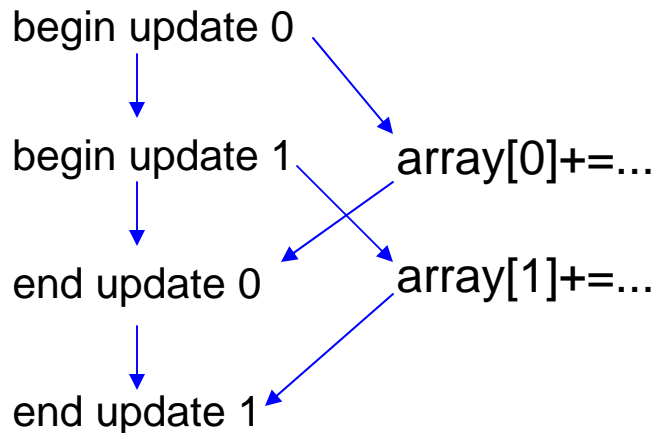
static void RecordEvent( const char* what, int arg1=0 ) {
    Event& e = EventArray[InterLockedIncrement(&EventCount) % N_Event];
    e.thread_id = ...some piece of information to identify the thread...;
    e.what = what;
    e.arg1 = arg1;
}
```

Example Usage

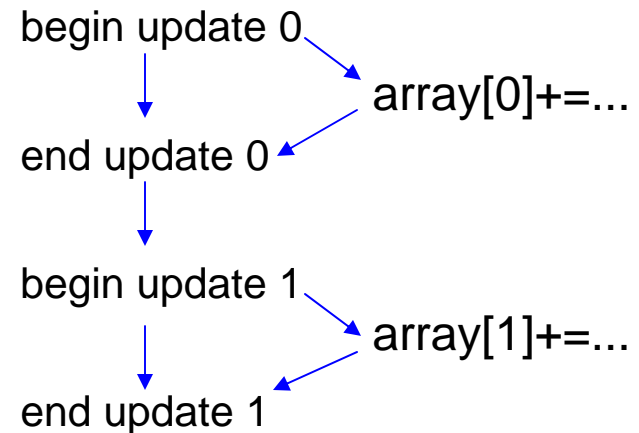
Should bracket action with two events.

```
RecordEvent("begin update %d", index);  
... array[index] += ...  
RecordEvent("end update %d", index);
```

Trace A: possibly concurrent updates, or maybe not



Trace B: update 0 definitely preceded update 1



Tradeoffs on Buffer Designs

- Single shared buffer with atomic update of index
 - Relatively fast and accurate
 - Small perturbations introduced:
 - Memory fences
 - Pipeline flush on out-of-order processor
 - Bad: introduces perturbations memory fences
- Single shared buffer with non-atomic update of index
 - Very fast, but possibly inaccurate
 - No fences or pipeline flushes
 - Buffer messages may be overwritten/garbled
- Thread-private buffers with timestamps
 - Merge and sort by timestamp
 - Must record timestamp
 - Sometimes fast and easy timestamp support exists
 - e.g. RDTSC instruction
 - And sometimes it does not.
 - Some versions of Linux* do *not* synchronize hardware clocks

Intel[®] Threading Tools

Intel[®] Thread Checker

Pinpoint notorious threading bugs like data races, stalls and deadlocks

<http://www.intel.com/software/products/threading/tcwin>

Intel[®] Thread Profiler

- Visualize your threads over time
- Identify synchronization objects that cause delays

<http://www.intel.com/software/products/threading/tp>

Example: Prime Number Generation

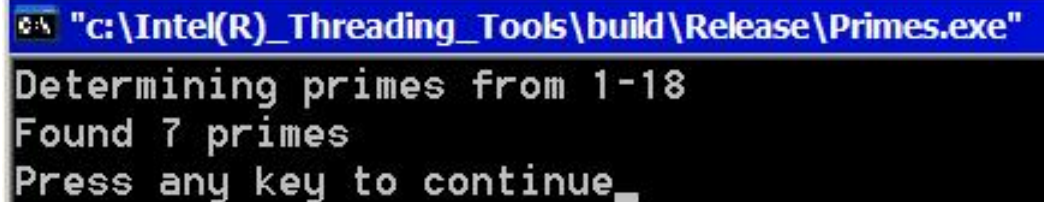
2
3 **3**
5 **3,5**
7 **3,5,7**
9 **3**
11 **3,5,7,9,11**
13 **3,5,7,9,11,13**
15 **3**
17 **3,5,7,9,11,13,15,17**

```
#include <stdio.h>
const long N = 18;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d \n", N );
    primes[ number_of_primes++ ] = 2; // special case
    for (long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
            primes[ number_of_primes++ ] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```

Example: Prime Number Kernel

2
3 3
5 3,5
7 3,5,7
9 3
11 3,5,7,9,11
13 3,5,7,9,11,13
15 3
17 3,5,7,9,11,13,15,17

```
for ( long number = 3; number <= N; number += 2 )  
{  
    long factor = 3;  
    while ( number % factor ) factor += 2;  
    if ( factor == number )  
        primes[ number_of_primes++ ] = number;  
}
```



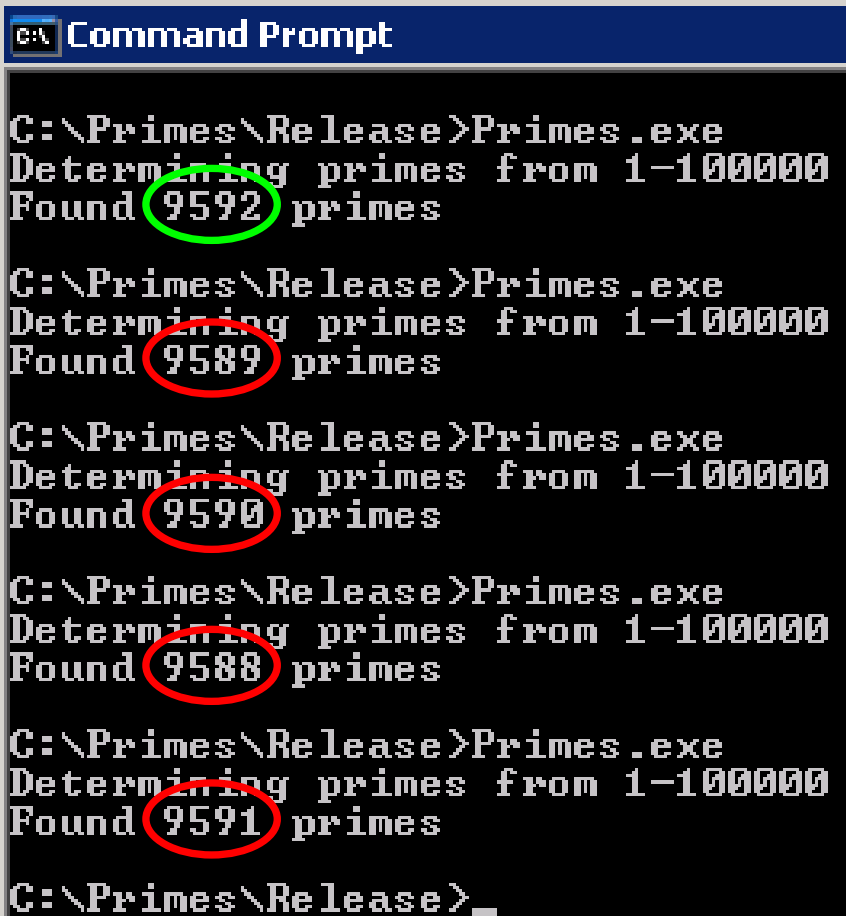
```
"c:\Intel(R)_Threading_Tools\build\Release\Primes.exe"  
Determining primes from 1-18  
Found 7 primes  
Press any key to continue_
```

Example: Not Quite Right

```
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d \n", N );
    primes[ number_of_primes++ ] = 2; // special case
    #pragma omp parallel for
    for ( long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
            primes[ number_of_primes++ ] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```

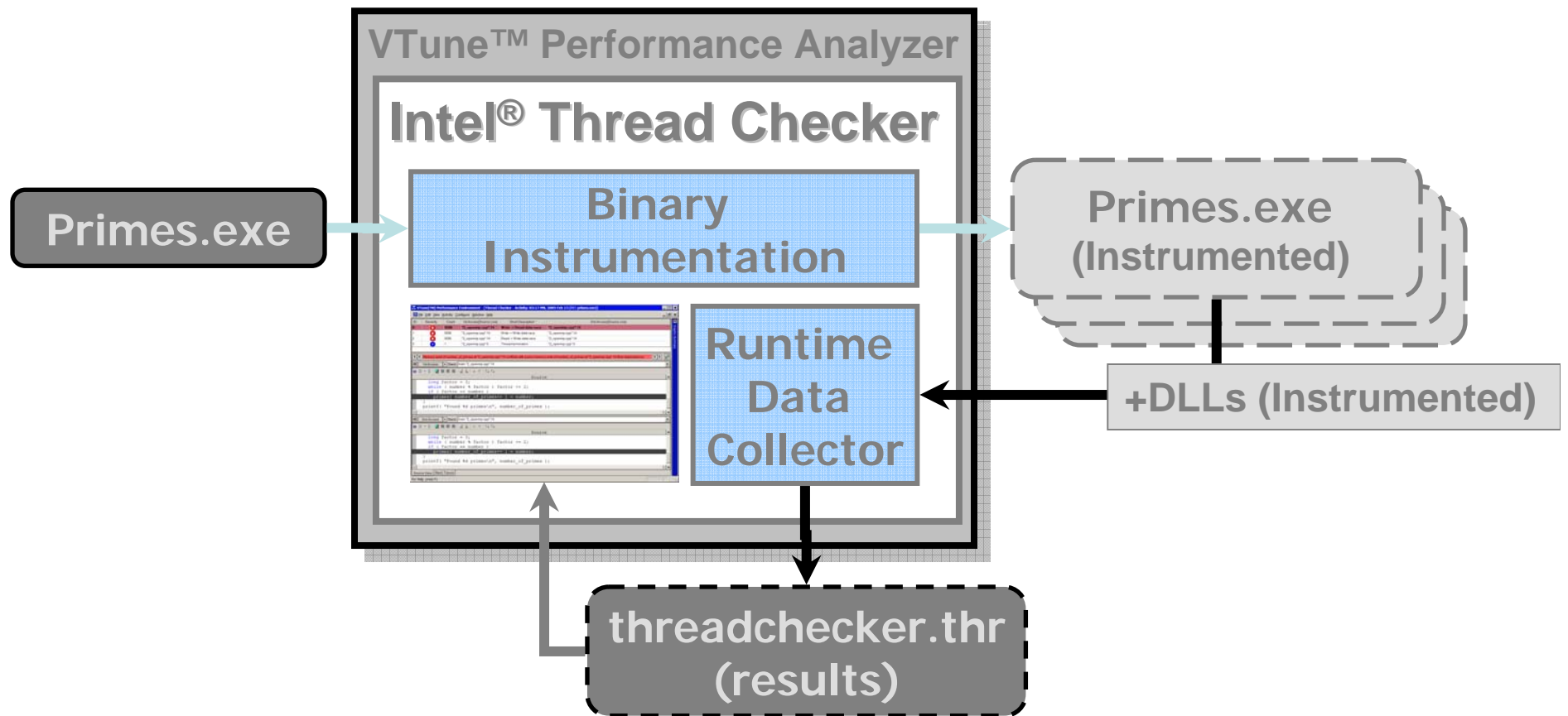
Example: Not Quite Right

```
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes;
main()
{
    printf( "Determining primes from 1-100000\n" );
    primes[ number_of_primes++ ] = 2;
    #pragma omp parallel for
    for ( long number = 3; number <= 100000; number++ )
    {
        long factor = 3;
        while ( number % factor ) factor++;
        if ( factor == number )
            primes[ number_of_primes++ ] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```



```
Command Prompt
C:\Primes\Release>Primes.exe
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>Primes.exe
Determining primes from 1-100000
Found 9589 primes
C:\Primes\Release>Primes.exe
Determining primes from 1-100000
Found 9590 primes
C:\Primes\Release>Primes.exe
Determining primes from 1-100000
Found 9588 primes
C:\Primes\Release>Primes.exe
Determining primes from 1-100000
Found 9591 primes
C:\Primes\Release>
```

Data Flows



Win32* threads, POSIX* threads, OpenMP*

Intel® Thread Checker

The screenshot displays the Intel Thread Checker interface within the VTune Performance Environment. The main window title is "VTune(TM) Performance Environment - [Thread Checker - Activity: 03:17 PM, 2005 Feb 13 (TC: primes.exe)]". The interface includes a menu bar (File, Edit, View, Activity, Configure, Window, Help) and a table of detected errors.

ID	Severity	Count	1st Access[Source Line]	Short Description	2nd Access[Source Line]
0		9590	"2_openmp.cpp":14	Write -> Read data-race	"2_openmp.cpp":14
1		9590	"2_openmp.cpp":14	Write -> Write data-race	"2_openmp.cpp":14
2		9590	"2_openmp.cpp":14	Read -> Write data-race	"2_openmp.cpp":14
3		1	"2_openmp.cpp":5	Thread termination	"2_openmp.cpp":5

Below the table, a detailed view of a memory race is shown. The description reads: "Memory read of number_of_primes at '2_openmp.cpp':14 conflicts with a prior memory write of number_of_primes at '2_openmp.cpp':14 (flow dependence)".

The interface shows two source code views for the conflicting accesses:

- 1st Access:** Stack: main "2_openmp.cpp":14. The code shows a loop where `primes[number_of_primes++] = number;` is executed.
- 2nd Access:** Stack: main "2_openmp.cpp":14. The code shows the same loop, where a subsequent read of `number_of_primes` occurs.

The source code for both views is:

```
long factor = 3;
while ( number % factor ) factor += 2;
if ( factor == number )
    primes[ number_of_primes++ ] = number;
}
printf( "Found %d primes\n", number_of_primes );
```

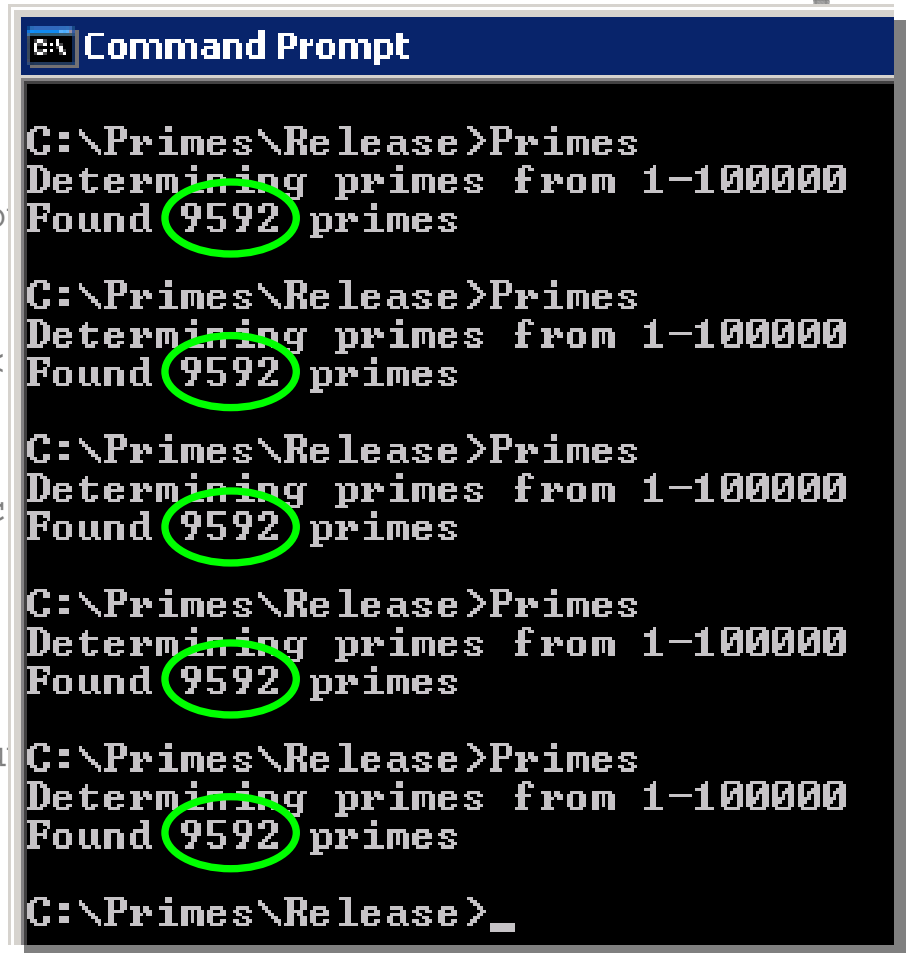
The interface also includes a "Graphic Summary" sidebar on the right and a status bar at the bottom with the text "For Help, press F1".

Example: A Bit Better

```
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes = 0;
main()
{
    printf( "Determining primes from 1-%d \n", N );
    primes[ number_of_primes++ ] = 2; // special case
#pragma omp parallel for
    for ( long number = 3; number <= N; number += 2 )
    {
        long factor = 3;
        while ( number % factor ) factor += 2;
        if ( factor == number )
#pragma omp critical
            primes[ number_of_primes++ ] = number;
    }
    printf( "Found %d primes\n", number_of_primes );
}
```

Example: A Bit Better

```
#include <stdio.h>
const long N = 100000;
long primes[N], number_of_primes
main()
{
    printf( "Determining primes from
    primes[ number_of_primes++ ] =
#pragma omp parallel for
    for ( long number = 3; number <
    {
        long factor = 3;
        while ( number % factor ) fac
        if ( factor == number )
#pragma omp critical
        primes[ number_of_primes++
    }
    printf( "Found %d primes\n", nu
}
```



```
Command Prompt
C:\Primes\Release>Primes
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>Primes
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>Primes
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>Primes
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>Primes
Determining primes from 1-100000
Found 9592 primes
C:\Primes\Release>_
```

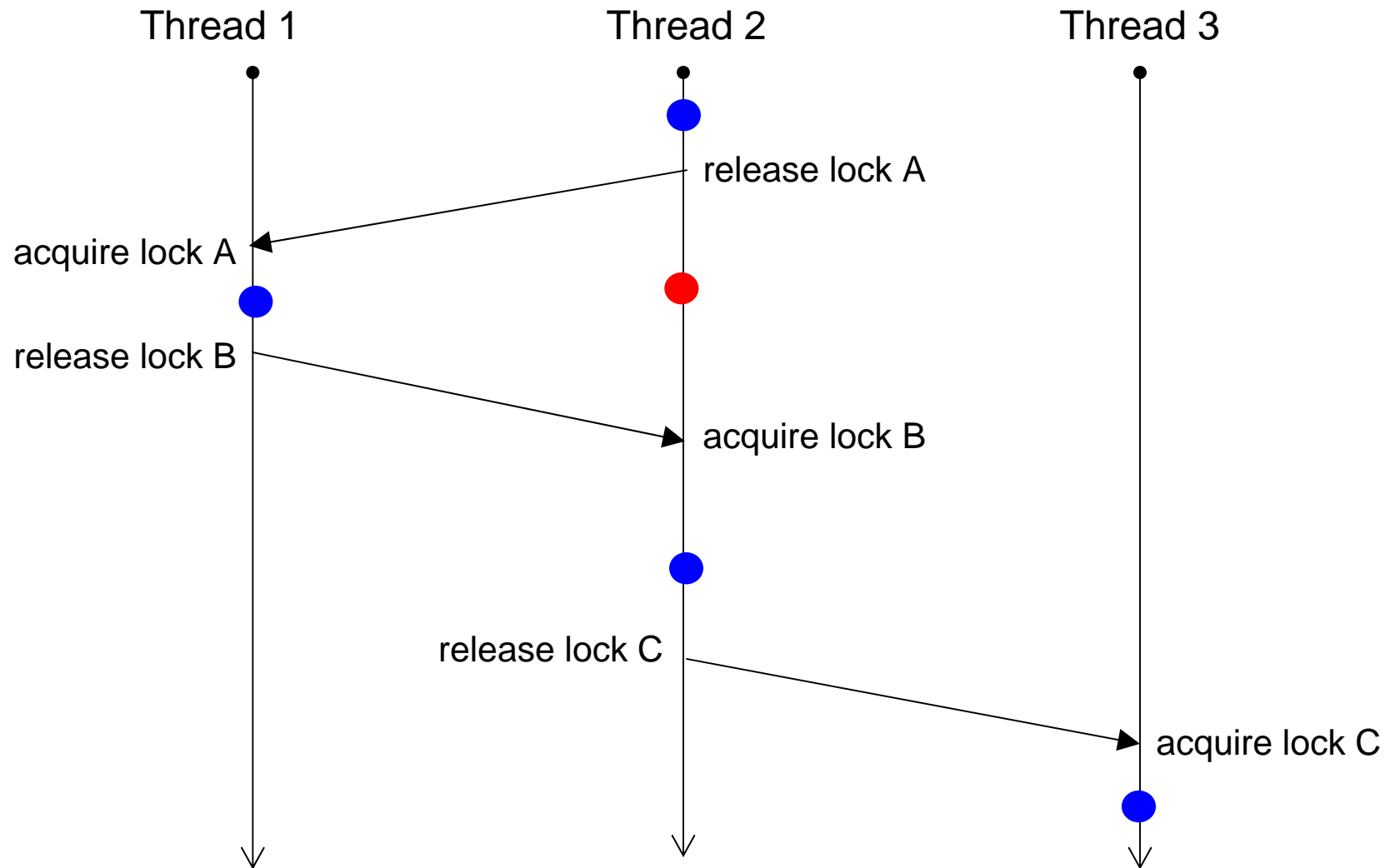
How It Works

- Looks for violations of Bernstein's Conditions
 - Unsynchronized read-write, write-read, or write-write
- Synchronization inferred from threading primitives
 - create thread
 - join with thread
 - send information to thread ("release")
 - receive information from thread ("acquire")
- Can instrument your own ad-hoc primitives

Example

blue dots are mutually synchronized

red dot is unsynchronized w.r.t. blue dots



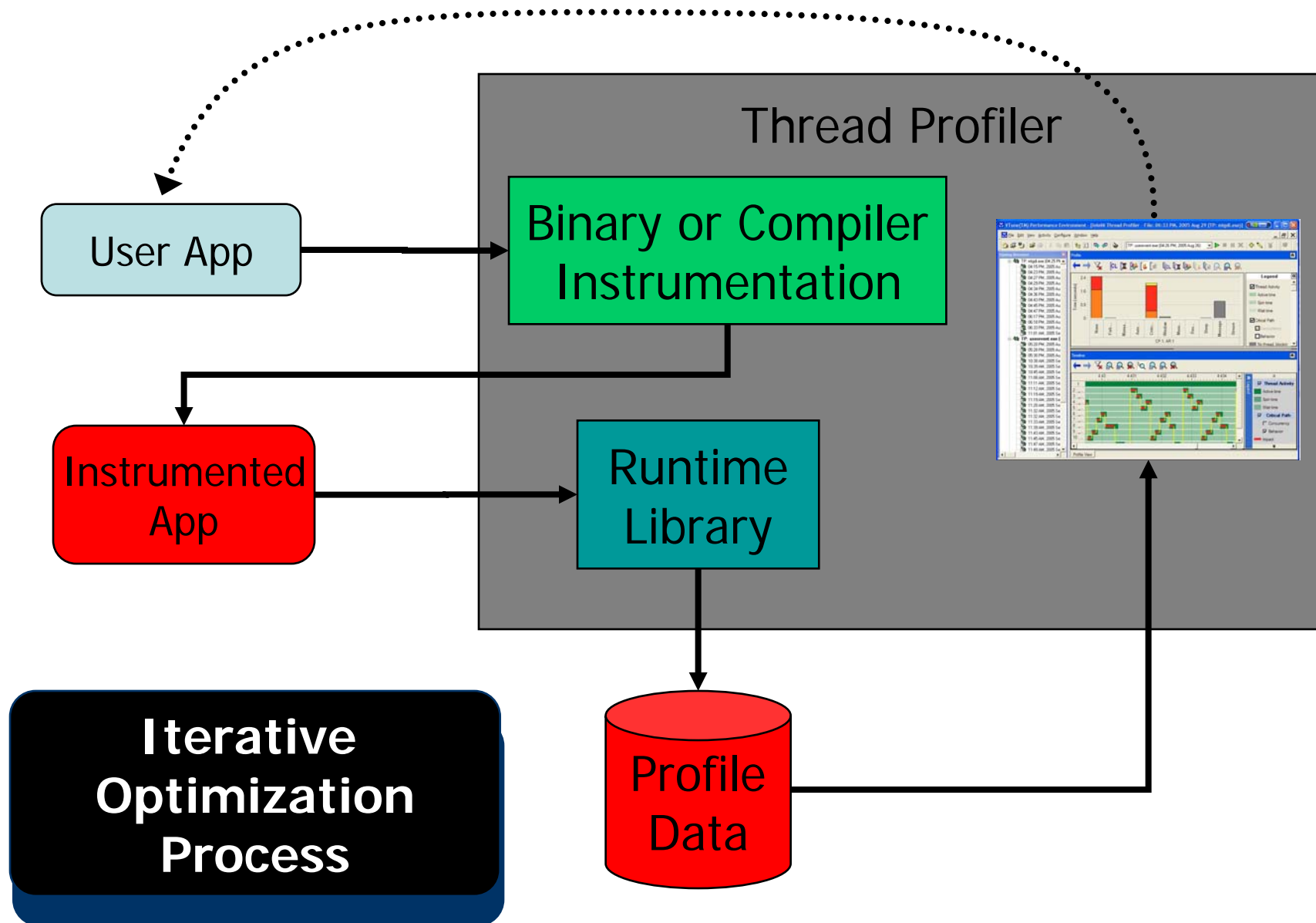
Summary : Thread Checker

- The Intel[®] Thread Checker provides a safety net for developers
 - Increases confidence in developers ability to debug threaded software.
- The techniques used by the Thread Checker allow analysis of “production” applications.
- Industry direction moving to multi-core solutions makes confidence tools even more critical to success

Intel® Thread Profiler

- The Thread Profiler consists contains two modes of analysis
 - OpenMP analysis
 - Single-level fork-join parallelism
 - Native Threading analysis
 - Arbitrary concurrency

Tool Components



System APIs

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

User Synchronization

- 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;

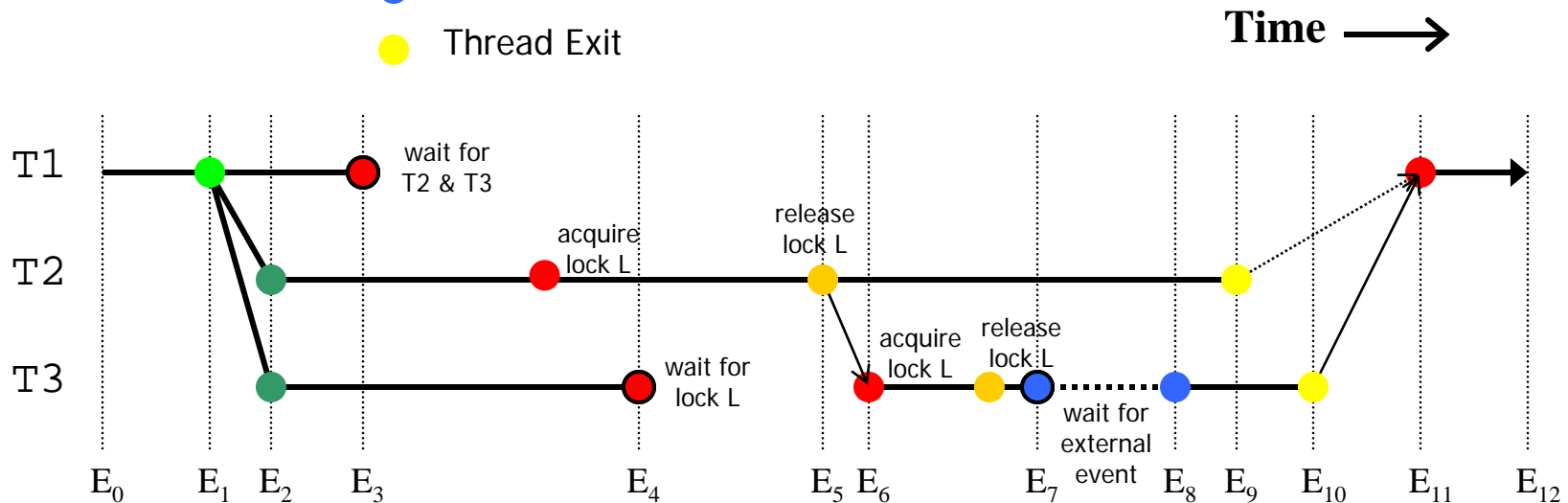
__itt_notify_sync_releasing( &spin );
release spin;
```

Instrumentation

- Usually low run-time overhead because only select events are instrumented
- Target overhead of less than 2x slowdown with reasonable synchronization
- common case is much less

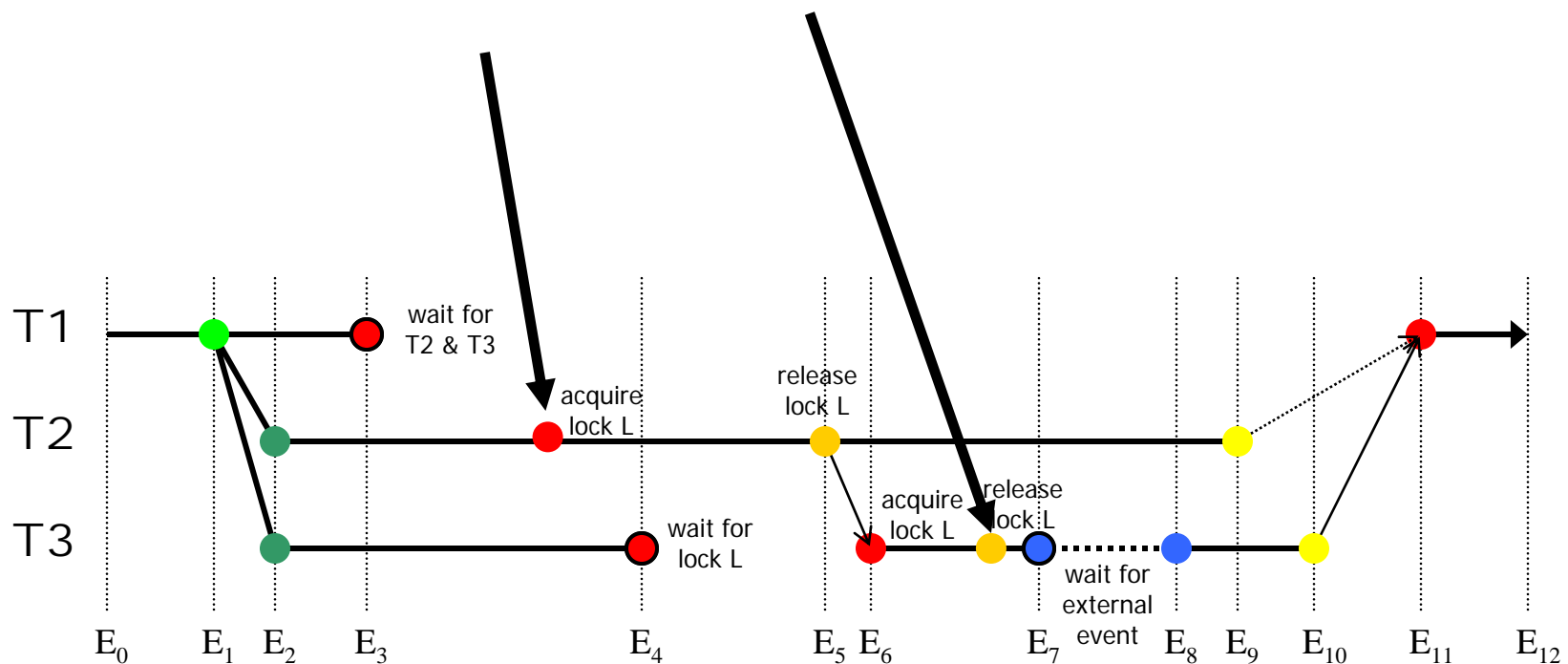
RECORDED EVENTS

- Create Thread (Fork)
- Thread Entry
- Wait for synchronization object or event
- Acquire synchronization object or event
- Release or signal synchronization object or event
- Wait for external event
- Receive external event
- Thread Exit



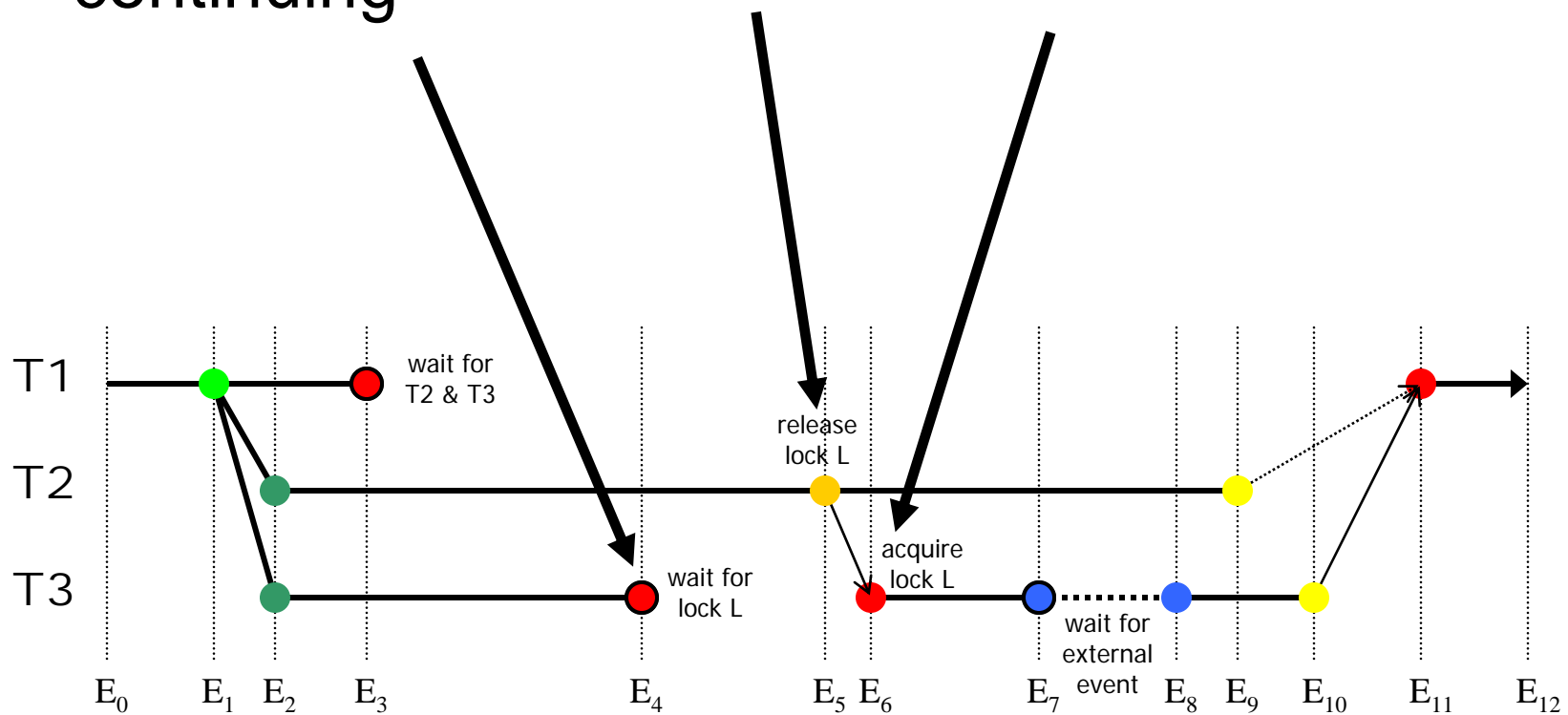
Uncontended Events

- i.e. Thread acquires lock that no one else holds. It continues without any pause.
- Thread Profiler will not consider these transactions.



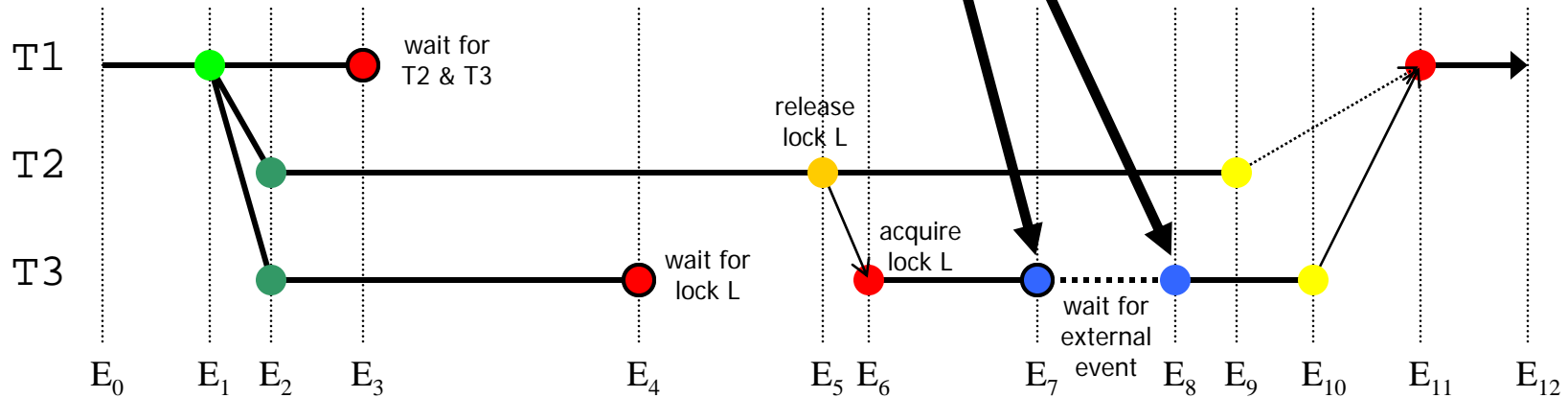
Contended Transitions

- i.e. Thread attempts to acquire lock that another thread holds. It must wait to acquire it before continuing



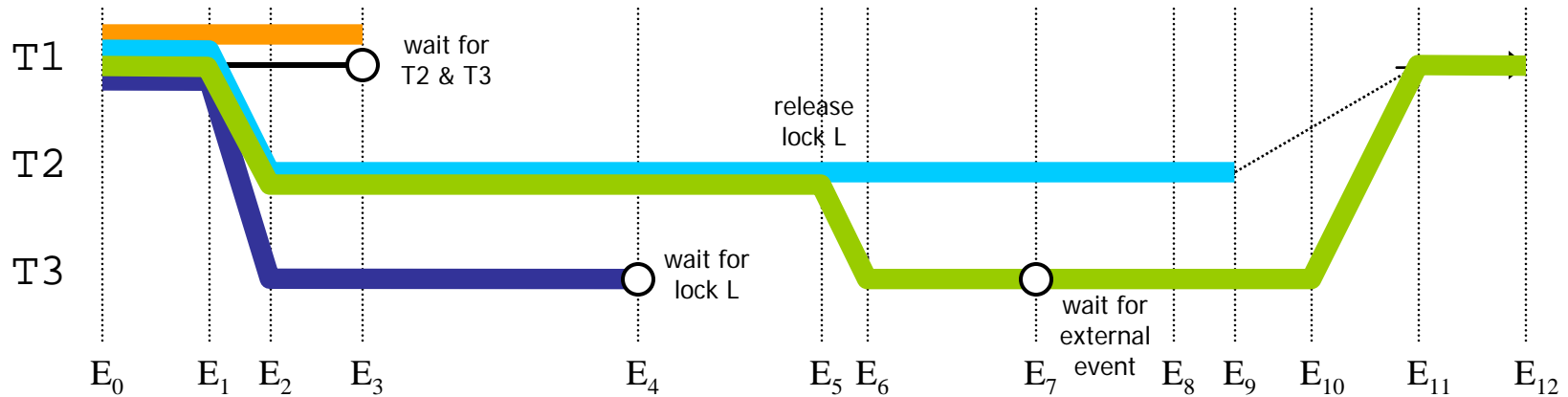
Thread Blocking

- Thread waits for external event such as user I/O, file operation, another process, &c.



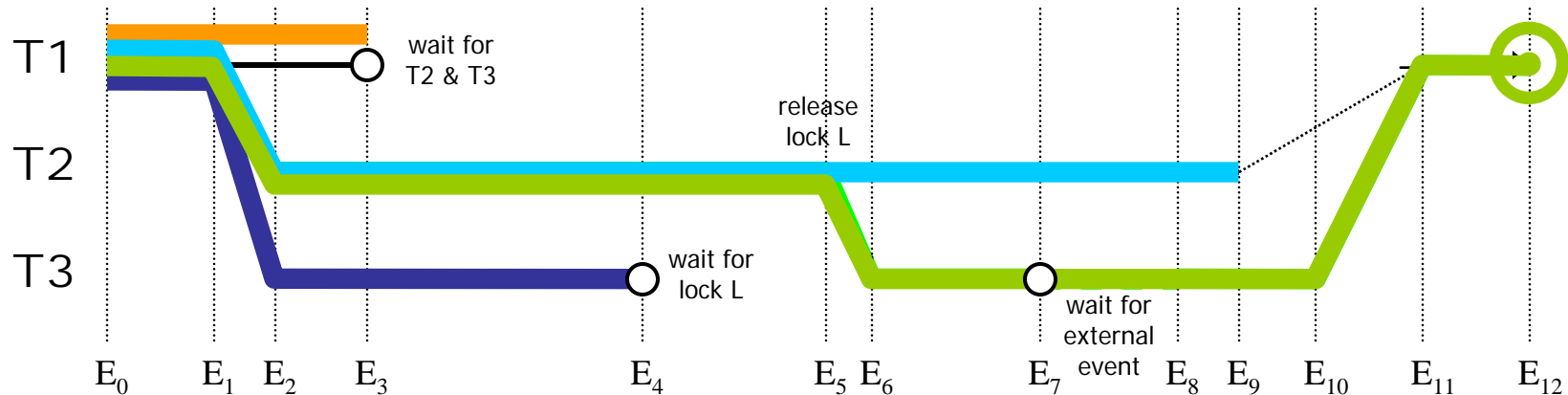
Execution Flows

- Flow **splits** when a thread **creates** a new thread or **signals (unblocks)** another thread
- Flow **ends** when a thread **waits** for another thread or **terminates**



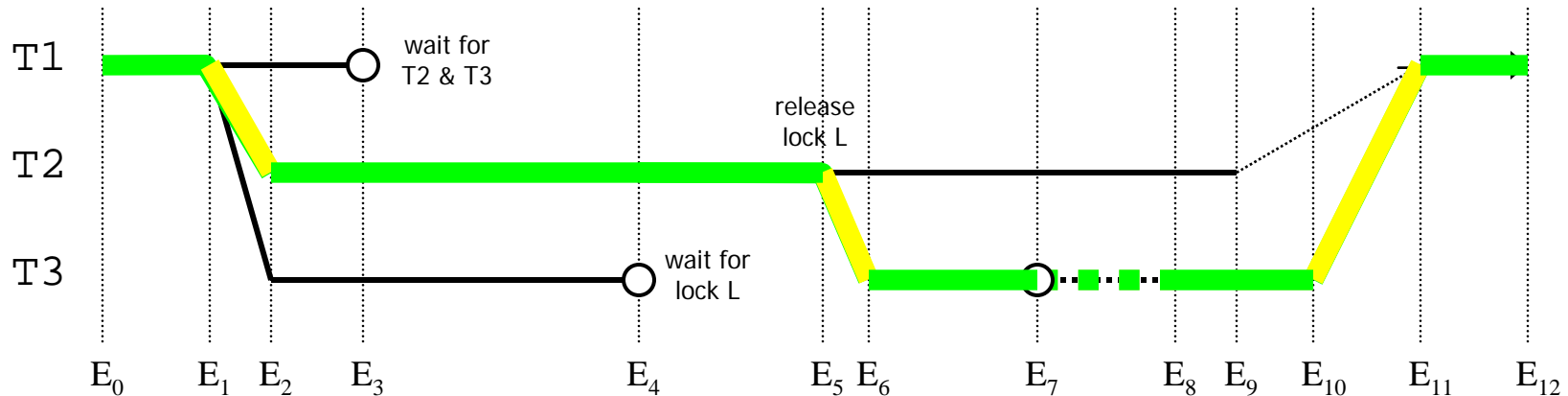
Critical Path Analysis

- The **continuous flow to target location** is the **critical path**
- Default target is the program termination
- Where to focus optimization energy
- Next, we will “color” the critical path to offer more information

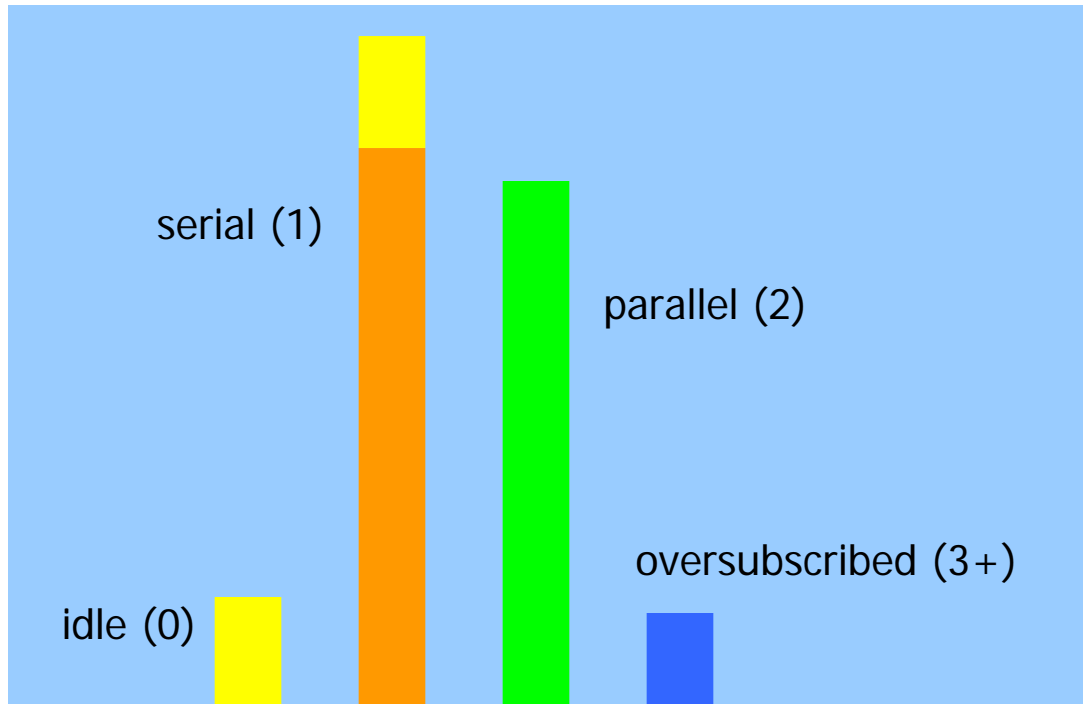


Transition Overhead Time

- Indicate the time spent between one thread signaling and another thread receiving the signal. This is marked as overhead time.

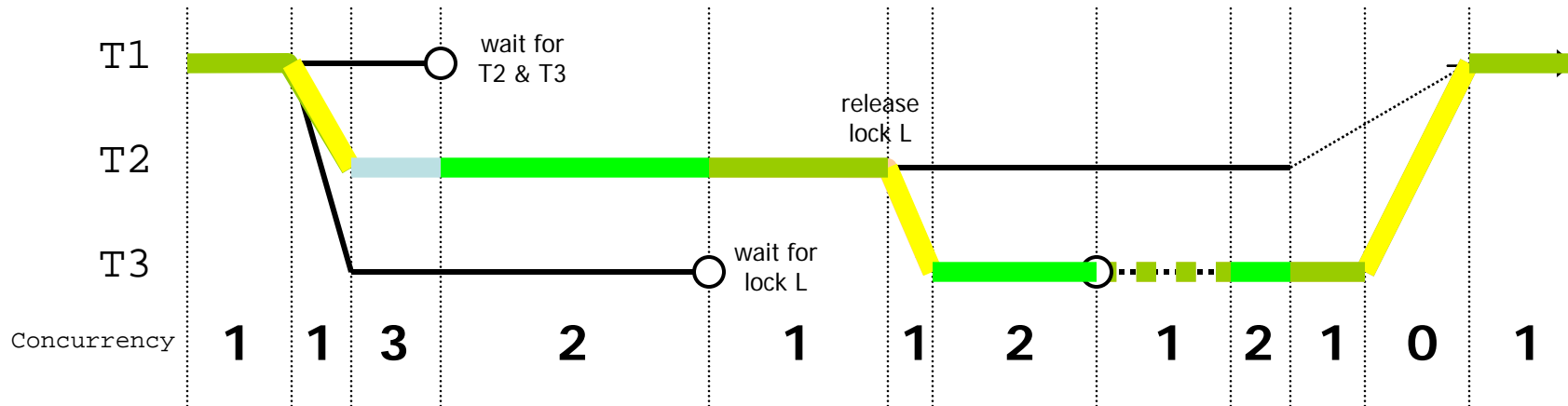


Processor Utilization



- Measure processor utilization so user can see how parallel their program really is
- Concurrency is the number of threads that are **active** (not waiting, sleeping, blocked, &c.) at a given time

(example reflects 2 processor machine)



Characterize parallel behavior of Critical Path

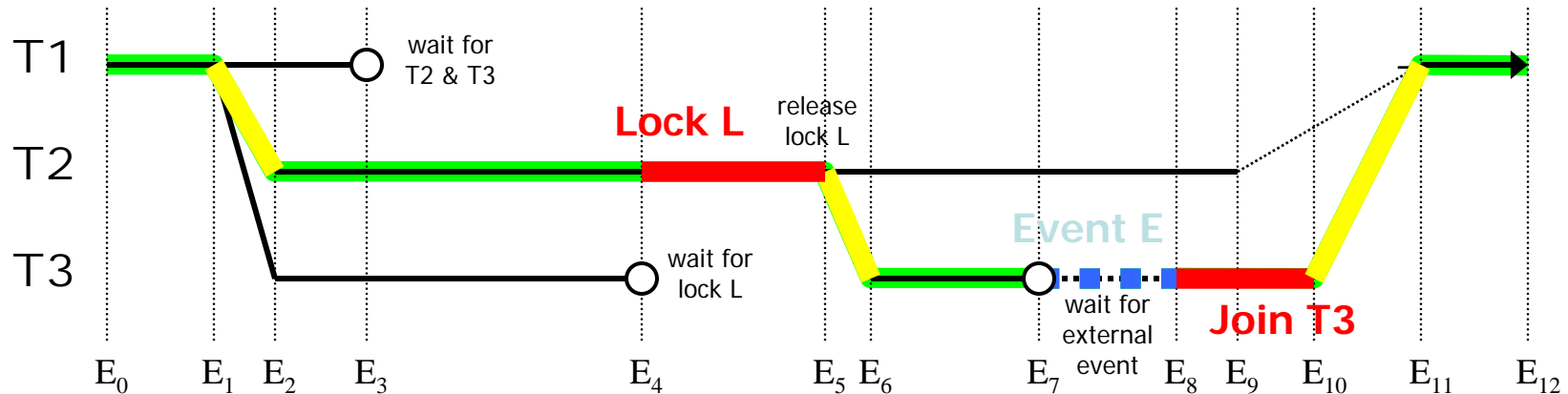
- Associate spans of time with the objects that caused the CP transitions
- This provides which objects/barriers cause parallel scalability problems that are directly affecting execution time to critical path target.

Cruise Time: Time a thread does not delay the next thread on the CP

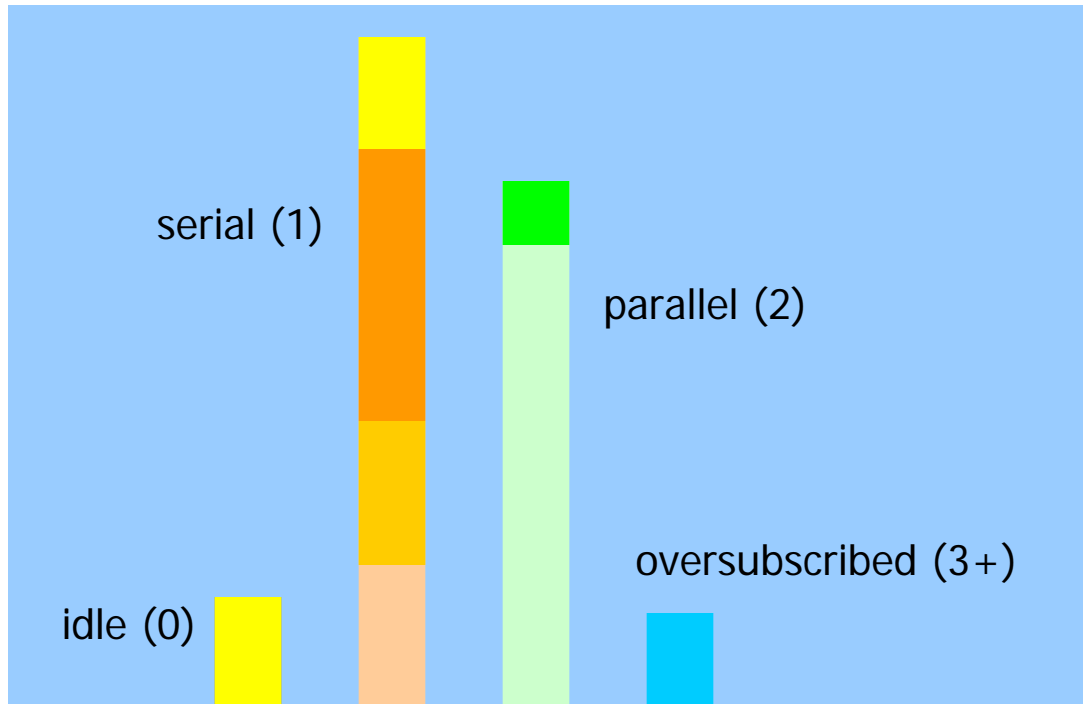
Overhead (transition) Time: Threading synchronization or OS scheduling overhead

Blocking Time: Time a thread spends waiting for an external event or blocking while still on the CP (includes timeouts)

Impact Time: Time a thread on the CP delays next thread from getting on the CP

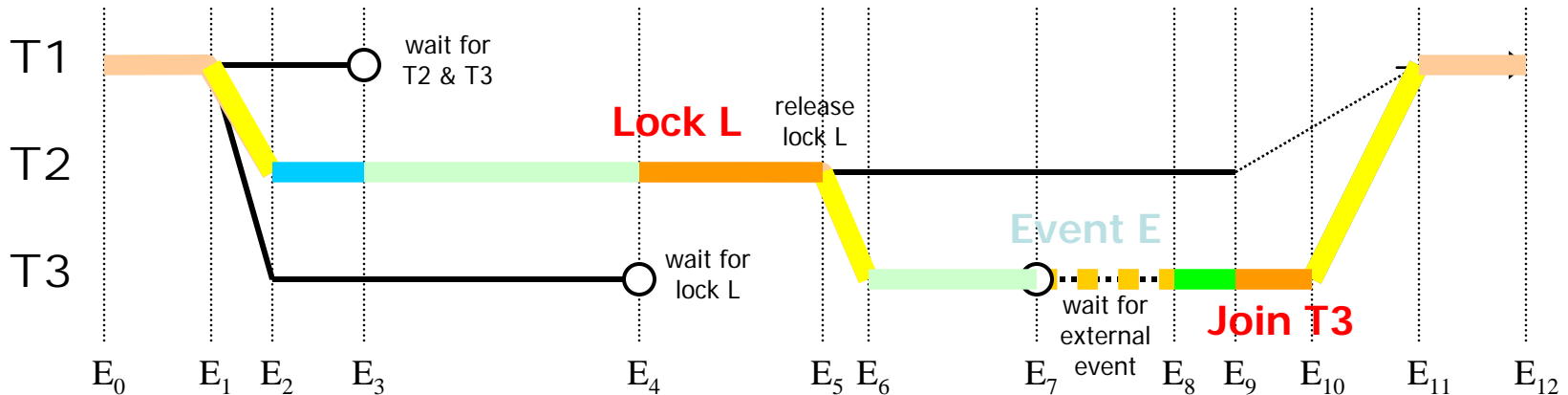


Combine concepts for Profile



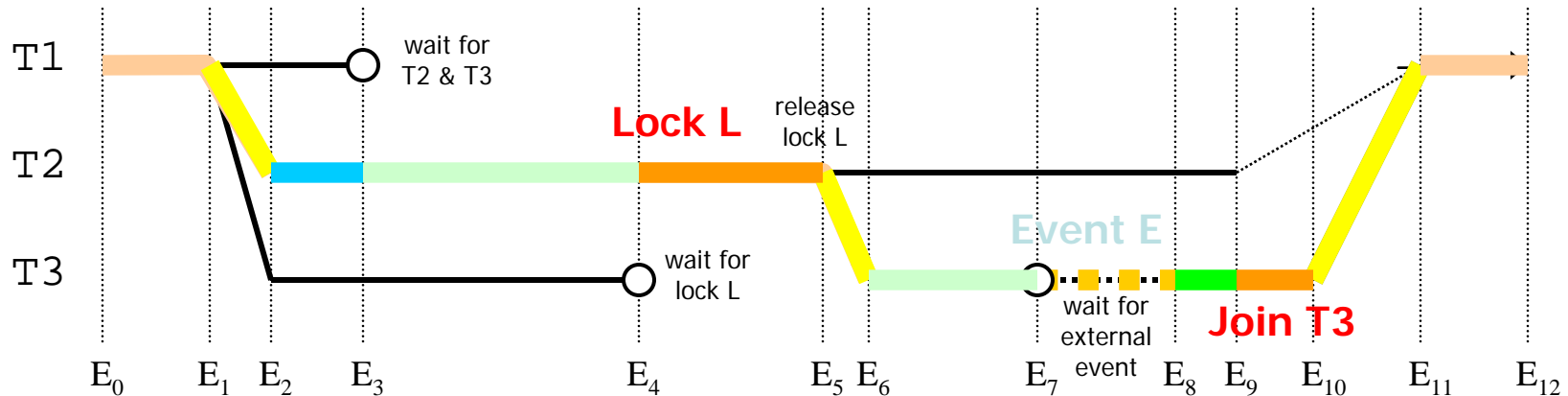
- Start with the CP
- Mark overhead
- Break it down by Utilization
- Further categorize by behavior

(example reflects 2 processor machine)



Analyze and Optimize Code

- Serial Optimizations
 - Serial optimizations along the critical path should affect execution time.
- Parallel Optimizations
 - Red/Orange: Look to locations where the concurrency level is less than optimal
 - Cruise, Blocking, Impact: Try increasing the level of parallelism
 - Blocking time: Try reducing wait for external object
 - Impact time: This represents parallel imbalance or synchronization object contention
- Analyze benefit of increasing number of processors



Research That Has Not Gone Mainstream Yet

- Replay/reverse debuggers
- Program Slicing

Summary

- Nondeterminism is the enemy
 - Race conditions
 - Memory consistency
- Get serial version right first
- Serial correctness techniques become even more important
 - Code reviews
 - Unit testing
 - Assertions
- Parallel techniques
 - Formal model checkers
 - Event tracing
 - Race detectors and profilers