Now .NET developers can boost application performance and increase the code quality and reliability needed for high-performance computing and enterprise applications.

by Levent Akyil and Asaf Yaffe, Software and Services Group, Intel Corporation
Developer-friendly tools that find errors early in the development cycle can have a great payoff. Errors that make it to a released product may damage the product’s reputation and are generally very costly to fix. The earlier we can detect and correct hard-to-find errors such as threading (data race and deadlocks) and memory errors (memory leaks) in any application, the easier it’s likely to fix them. The confidence and the performance tools aid developers by attributing problems to source code lines, along with call stack and timeline visualization of events, giving developers a clear picture about the issues in their software. Intel® Parallel Studio XE enables .NET developers to identify critical performance problems such as the most time-consuming functions or lines of code, scalability issues, and time spent waiting on synchronization constructs and I/O activity. While doing this, Parallel Studio XE reveals potential micro-architectural bottlenecks caused by issues such as branch mispredicts, cache misses, and memory bandwidth problems.

In this article, we highlight examples of how two tools in Intel Parallel Studio XE, the Intel® Inspector XE correctness tool, and the Intel® VTune™ Amplifier XE performance analysis tool, are valuable to developers of .NET code, native code, and “mixed” (.NET and native) applications during the development cycle. After explaining the current .NET support in Inspector XE and VTune Amplifier XE, we demonstrate the key features in action on C# applications.

Figure 1: Microsoft Visual Studio® Debug Properties for a .NET project

Current .NET support in
Intel® VTune™ Amplifier XE and
Intel® Inspector XE

Inspector XE and VTune Amplifier XE products support the analysis of pure .NET applications, as well as “mixed” applications that contain both managed and unmanaged code.

The Inspector XE thread analyzer can detect potential deadlocks and data races in .NET programs, in a similar way as it does for native code. Inspector XE monitors object allocations and accesses to shared memory on the garbage-collected heap and the static data areas, and flags unsynchronized accesses (at least one of which is a write operation) of multiple threads to the same object/class data member as a potential data race. Inspector is also aware of all the .NET 2.0 through .NET 3.5 locking APIs, and can detect deadlocks and lock hierarchy violations.

VTune Amplifier XE assists developers in fine-tuning serial and parallel applications for optimal performance on modern processors and makes it simple for .NET developers to quickly find performance bottlenecks in their pure .NET or mixed applications. VTune Amplifier XE’s hotspot analysis highlights the functions and source locations where the application spends most of its execution time. Concurrency and Locks & Waits analyses visualize the work distribution between threads as well as thread synchronization points, and helps users identify work distribution problems and excessive thread synchronization which prevent parallel execution. VTune Amplifier XE can also help developers identify microarchitectural performance issues by using the CPU’s Performance Monitoring Unit (PMU) to sample processor events and identify the architectural bottlenecks on a given Intel® processor.

Configuring .NET analysis in
VTune Amplifier XE and Inspector XE

Users can select whether to analyze the managed parts (“managed” mode), native parts (“native” mode) or both (“mixed” mode). These types are supported as follows:

› **Native** mode collects data on native code only and does not attribute data to managed code.
› **Managed** mode collects data on managed code only and does not attribute data to native code.
› **Mixed** mode collects and attributes data to both native and managed code as appropriate. Use this option when analyzing managed executables that make calls to native code.
› **Auto** mode automatically detects the type of the target executable. It switches to mixed mode when a managed application is detected and to native mode when a native application is detected.
The way to configure the analysis mode depends on the way one uses the tools. When using the tools from the command line, specify the mode using the "-mrte-mode" switch. When using the tools within Microsoft Visual Studio®, the analysis mode is automatically selected based on the active project type: for native projects (C/C++ applications) the default analysis mode in both VTune Amplifier XE and Inspector XE is set to "native." For .NET projects (C# applications), the default analysis mode in VTune Amplifier XE is set to "managed," while the default analysis mode in Inspector XE is set to "mixed." Users can use the Visual Studio Debug Properties page to select a different mode. To enable "mixed" analysis mode for a .NET project, enable the "unmanaged code debugging" feature (Figure 1). Similarly, to enable "mixed" analysis mode for a native project, set the Debugger Type property to "mixed" (Figure 2). When using the standalone graphical interface of VTune Amplifier XE or Inspector XE, users can configure the analysis mode from the Project Properties dialog (Figures 3 and 4).
namespace VTuneAmplifierXE.Examples
{
    public class POTENTIAL_MT
    {
        private static Thread[] threads = new Thread[Benchmarks.gThreadCount];
        private static WorkerThread[] workerThreads = new WorkerThread[Benchmarks.gThreadCount];
        private static object[] threadParams = new object[Benchmarks.gThreadCount];

        // Start and done signals
        static AutoResetEvent[] goSignals = new AutoResetEvent[Benchmarks.gThreadCount];
        static AutoResetEvent[] doneEvents = new AutoResetEvent[Benchmarks.gThreadCount];

        public static double potential = 0.0;
        public double potentialTotal = 0.0;

        public class WorkerThread
        {
            private volatile bool stopNow = false;
            private ThreadParameters threadParameters;

            public void RequestStop()
            {
                stopNow = true;
            }

            private void computePot_MT(int tid)
            {
                int start, end;
                double distx, disty, distz, dist;

                start = threadParameters.chunkBegin;
                end = threadParameters.chunkEnd;
                potential = 0.0;

                for (int i = start; i < end; i++)
                {
                    for (int j = 0; j < i - 1; j++)
                    {
                        distx = Math.Pow((r[0][j] - r[0][i]), 2);
                        disty = Math.Pow((r[1][j] - r[1][i]), 2);
                        distz = Math.Pow((r[2][j] - r[2][i]), 2);
                        dist = Math.Sqrt(distx + disty + distz);
                        potential += 1.0 / dist;
                    }
                }
            }

            public void doWork(object parameter)
            {
                threadParameters = (ThreadParameters)parameter;
                Console.WriteLine("Thread: {0} ready to start \{1} - \{2}\",
                    threadParameters.tid,
                    threadParameters.chunkBegin,
                    threadParameters.chunkEnd);

                while (!stopNow)
                {
                    threadParameters.goSignal.WaitOne();
                    computePot_MT(threadParameters.tid);
                    threadParameters.eventDone.Set();
                }

                Console.WriteLine("worker thread: terminating gracefully.");
            }
        } // end WorkerThread
    } // end POTENTIAL_MT
} // end namespace VTuneAmplifierXE.Examples

---

Figure 5
Sample Code

To demonstrate how Inspector XE and VTune Amplifier XE support .NET applications, we use a C# program computing the potential energy of a system of particles based on the distance in three dimensions. This is a threaded application which uses the .NET thread pool to create as many threads as the number of cores available. The goal of this article is not to introduce C# threads or how to thread efficiently with .NET framework, but rather to demonstrate how the tools can help to identify threading issues and significantly aid in developing high-performing, scalable parallel applications.

The code below shows the part of the application executed by each worker thread. The computePot method is where the action happens. Each thread uses the stored boundaries indexed by the thread’s assigned identification number (tid). This helps to fix the start and end range of particles to be used. After each thread initializes its iteration space (start and end values), it starts computing the potential energy of the particles.

Intel® Inspector XE in Action

Let’s start by running Inspector XE on our sample code. From the Visual Studio Tools menu, select “Intel Inspector XE 2011,” and then “New Analysis” (Figure 6). In the Configure Analysis Type page that opens, select “Locate Deadlocks and Data Races” (Figure 7), and click the “Start” button to start threading correctness analysis.

Running Inspector XE on our sample code reveals that we have a data race (Figure 8).

The Problems pane lists individual problems. Code Locations shows source code locations that are relevant for the selected problems. The Filters pane allows you to filter the Problems view by severity, problem type, module, and source files.

Figure 6: Starting a new Inspector XE analysis from Visual Studio

Figure 7: Selecting an Inspector XE analysis type

Figure 8: A potential data race identified by Inspector XE 2011
Double-clicking on the problem takes us to the Sources page (Figure 9) where we can further investigate the problem.

This page shows two representative threads of execution that perform an unsynchronized access to a shared memory location, including a detailed call stack for each thread. Using this view we can quickly determine that we have an unsynchronized access to the “potential” static class member—a classic data race. We can double-click any source line to jump directly into the source code and fix the issue.

One trivial solution to this data race is to make sure all access to the “potential” class member is properly synchronized with a lock. However, this solution will introduce a serial region (a critical section) into our parallel code and will likely affect performance. A better solution is for each thread to store a private copy of the potential in a thread local variable, and then accumulate the results to compute the final value. This solution reduces dependencies and synchronization between threads and is likely to speed up the parallel code.

A Few Words about Memory Checker

The Inspector XE memory analyzer is also aware of .NET code and can be used for finding memory leaks and errors in mixed applications that contain managed and native code. While the memory analysis is conducted only for the native portions of the applications (as most of the detectable errors are irrelevant for .NET code), the analysis results show complete stack traces, including the .NET call chain that led to the memory error in native code.

Combining the threading analysis and memory analysis capabilities make Inspector XE a powerful and invaluable tool for analyzing the correctness of complex applications that combine .NET and native code in a single program.

Intel® VTune™ Amplifier XE in Action

Now that we solved our correctness issues, let’s start analyzing the performance of our application. Figure 10 shows how to start the concurrency analysis within Visual Studio*. If our application is analyzed on a quad-core Intel® 2nd generation core architecture family processor running at 2.5GHz, we get the results summary, as shown in Figure 11.
Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were running simultaneously. Threads are considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrency is a measurement of the number of threads that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.

CPU Usage Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes what percentage of the wall time the specific number of CPUs were running simultaneously. CPU Usage may be higher than the thread concurrency if a thread is executing code on a CPU while it is logically...

Figure 11: Thread concurrency and CPU usage histograms for the analysis
Figure 11 shows that our threaded application is not fully utilizing all available cores. The bottom-up view (Figure 12) gives a closer look at the results. Our `workerThread::computePot_mt` method is consuming most of the CPU time, and has significant amount of time identified as poor (red) and okay (orange) CPU utilization. This indicates that this particular method is a hotspot (i.e., consuming most of the CPU Time) and threaded, but not fully utilizing the available cores. Therefore, it makes sense to zoom in to the timeline and look at each thread executing this particular method. Figure 13 makes it clear that four threads, which are executing the `workerThread::computePot_mt` method, consume different amount of CPU time, causing a load imbalance and sub-optimal utilization of the cores. Such load imbalance issues will prevent applications from scaling as desired on more cores and needs to be fixed.

Even though each thread executes the outer `for` loop the same number of times, the inner loop is executed more by the thread operating on the last chunk, and least by the thread operating on the first chunk. Distributing the iteration's cyclical offset by the thread count will fix the load imbalance and the threads will utilize the available cores better. The concurrency analysis and the results not only enable us to identify load imbalance issues, but also help us speed up the application. The change below allows all the threads to stay busy and keep running (Figure 16).

“Load imbalance issues will prevent applications from scaling as desired on more cores and need to be fixed.”
chunkBegin = tid * (nParts / threadCount);
chunkEnd = (tid + 1) * (nParts / threadCount);

start = threadParameters.chunkBegin;
end = threadParameters.chunkEnd;

for (int i = start; i < end; i++)
    for (int j = 0; j < i - 1; j++)
        distx = math.Pow((r[0][j] - r[0][i]), 2);
        disty = math.Pow((r[1][j] - r[1][i]), 2);
        distz = math.Pow((r[2][j] - r[2][i]), 2);
        dist = math.Sqrt(distx + disty + distz);
        potential += 1.0 / dist;

for (int i = tid; i < constants.POT_NPARTS; i += threadParameters.threadCount)
    for (int j = 0; j < i - 1; j++)
        distx = math.Pow((r[0][j] - r[0][i]), 2);
        disty = math.Pow((r[1][j] - r[1][i]), 2);
        distz = math.Pow((r[2][j] - r[2][i]), 2);
        dist = math.Sqrt(distx + disty + distz);
        potential += 1.0 / dist;

Figure 13: Showing the thread load imbalance on the timeline and CPU time of each thread executing

<table>
<thead>
<tr>
<th>Thread 1 (range)</th>
<th>Thread 2 (range)</th>
<th>Thread 3 (range)</th>
<th>Thread 4 (range)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original version</td>
<td>0 - 249</td>
<td>250 - 499</td>
<td>500 - 749</td>
</tr>
<tr>
<td>After changes</td>
<td>0, 4, 8, ..., 992, 996</td>
<td>1, 5, 9, ..., 993, 997</td>
<td>2, 6, 10, ..., 994, 998</td>
</tr>
</tbody>
</table>

Figure 14

Original Version

chunkBegin = tid * (nParts / threadCount);
chunkEnd = (tid + 1) * (nParts / threadCount);

start = threadParameters.chunkBegin;
end = threadParameters.chunkEnd;

for (int i = start; i < end; i++)
    for (int j = 0; j < i - 1; j++)
        distx = Math.Pow((r[0][j] - r[0][i]), 2);
        disty = Math.Pow((r[1][j] - r[1][i]), 2);
        distz = Math.Pow((r[2][j] - r[2][i]), 2);
        dist = Math.Sqrt(distx + disty + distz);
        potential += 1.0 / dist;

Load-balanced Version

for (int i = tid; i < constants.POT_NPARTS; i += threadParameters.threadCount)
    for (int j = 0; j < i - 1; j++)
        distx = Math.Pow((r[0][j] - r[0][i]), 2);
        disty = Math.Pow((r[1][j] - r[1][i]), 2);
        distz = Math.Pow((r[2][j] - r[2][i]), 2);
        dist = Math.Sqrt(distx + disty + distz);
        potential += 1.0 / dist;

Figure 15
Are we done? Not yet. Let's give a try to the architectural analysis of VTune™ Amplifier XE to check if the tool can identify more opportunities for performance improvements. To demonstrate the architectural analysis feature in VTune Amplifier XE, let's use General Exploration analysis pre-configured for 2nd Generation Core™ architecture.

The 2nd generation Core microarchitecture is capable of reaching Cycles Per Instructions as low as 0.25 in ideal situations. The greater value of CPI for a given workload indicates that there are more opportunities for code tuning to improve performance. Figure 18 shows the results of the General Exploration analysis. In this case, the invocation of the Math.Pow() function consumes significant amount of clockticks. Replacing Math.Pow() with a simple multiplication gives us much better performance and reduces the CPI ratio to 1.5.
Figure 18: General Exploration results highlighting problematic functions

```csharp
for (int i = tid; i < constants.POT_NPARTS; i += threadParameters.threadCount)
{
    for (int j = 0; j < i - 1; j++)
    {
        distx = Math.Pow((r[0][j] - r[0][i]), 2);
        disty = Math.Pow((r[1][j] - r[1][i]), 2);
        distz = Math.Pow((r[2][j] - r[2][i]), 2);
        dist = Math.Sqrt(distx + disty + distz);
        potential += 1.0 / dist;
    }
}
```

Figure 19

<table>
<thead>
<tr>
<th>Original Version (sec)</th>
<th>Load Imbalance Fixed (sec)</th>
<th>Math.Pow() replaced with * (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>4.65</td>
<td>0.39</td>
</tr>
</tbody>
</table>

Figure 20

“Intel® Parallel Studio XE enables .NET developers to identify critical performance problems such as the most time-consuming functions or lines of code, scalability issues, and time spent waiting on synchronization constructs and IO activity. While doing this, Parallel Studio XE reveals potential micro-architectural bottlenecks caused by issues such as branch mispredicts, cache misses, and memory bandwidth problems.”
For advanced and deeper microarchitectural analysis, the tool is equipped with predefined analysis types, which use Performance Monitoring Unit (PMU) to sample processor events to identify microarchitectural issues such as cache misses, stall cycles, branch mispredictions, and many more. The advanced analysis types are defined for processor architectures such as Intel® Core™ microarchitecture (aka Nehalem and Westmere) and Intel® 2nd Generation Core™ microarchitecture (aka Sandy Bridge). When these advanced predefined analysis types are used, the tool gives hints and suggestions by highlighting the problematic functions.

Supported .NET Versions

VTune Amplifier XE and Inspector XE support the basic synchronization mechanisms available in .NET versions 2.0 to 3.5. The tools do not support the new synchronization APIs introduced in .NET 4.0 and the new Task Parallel Library.

Summary

Inspector XE and VTune Amplifier XE provide valuable technologies to .NET developers. These tools combine error checking and performance profiling tools under Intel Parallel Studio XE. They help boost application performance and increase the code quality and reliability needed by high-performance computing and enterprise applications. At the same time, the suite eases the procurement of all the necessary tools for high performance, and simplifies the transition from multicore to manycore processors for the future. View the source code >

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Cycles Per Instruction retired, or CPI, is a fundamental performance metric indicating approximately how much time each executed instruction took, in units of cycles. Modern superscalar processors issue up to four instructions per cycle, suggesting a theoretical best CPI of 0.25. Various effects (long-latency memory, floating-point, or SIMD operations; non-retired instructions due to branch mispredictions; instruction starvation in the front-end) tend to pull the observed CPI up. Nonetheless, CPI is an excellent metric for judging an overall potential for application performance tuning.

Performance Tuning Metric

For advanced and deeper microarchitectural analysis, the tool is equipped with predefined analysis types, which use Performance Monitoring Unit (PMU) to sample processor events to identify microarchitectural issues such as cache misses, stall cycles, branch mispredictions, and many more. The advanced analysis types are defined for processor architectures such as Intel® Core™ microarchitecture (aka Nehalem and Westmere) and Intel® 2nd Generation Core™ microarchitecture (aka Sandy Bridge). When these advanced predefined analysis types are used, the tool gives hints and suggestions by highlighting the problematic functions.

Supported .NET Versions

VTune Amplifier XE and Inspector XE support the basic synchronization mechanisms available in .NET versions 2.0 to 3.5. The tools do not support the new synchronization APIs introduced in .NET 4.0 and the new Task Parallel Library.

Summary

Inspector XE and VTune Amplifier XE provide valuable technologies to .NET developers. These tools combine error checking and performance profiling tools under Intel Parallel Studio XE. They help boost application performance and increase the code quality and reliability needed by high-performance computing and enterprise applications. At the same time, the suite eases the procurement of all the necessary tools for high performance, and simplifies the transition from multicore to manycore processors for the future. View the source code >

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

© 2012, Intel Corporation. All rights reserved. Intel, the Intel logo, Atom, Cilk, Intel Core, VTune, and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries. *Other names and brands may be claimed as the property of others.