by Walter Shands,
Software Development Engineer

Using Intel® Software Development Tools to **Analyze HMMER**

This paper will highlight the features of Intel® Parallel Studio XE 2013 by using them to build and analyze HMMER (http://hmmer.janelia.org/). HMMER is a set of applications, which includes two, hmmsearch and hmmbuild, which are components of SPECint. We make use of event-based sampling analysis in Intel® VTune™ Amplifier XE to find out which code paths, context switches, or threading inactivity cause performance problems in hmmsearch. And, we'll utilize the code optimization features of the Intel® Composer XE compiler to improve the performance of hmmsearch on Intel® Xeon® E5 processors. In addition, we will show you how to use Intel® Inspector XE to locate memory and threading errors introduced into hmmsearch.

hmmsearch is used to search a protein sequence database for homologs of protein sequences using profiles called hidden Markov models. globins4.hmm contains the profiles and uniprot_trembl.fasta is a 10 GB sequence database.

hmmsearch is available in an MPI version, but we restricted our experiments to the non-MPI flavor. We ran hmmsearch on a computer with an 8-core Intel® Xeon® E5-2680 hyperthreaded processor at 2.7 GHz with 23.4 GB of memory. We ran the application using GCC and the Intel® C compiler, in both cases using the settings provided by the configure script. The initial GCC default switches were:

```
gcc -std=gnu99 -O3 -fomit-frame-pointer -ma-
lign-double -fstrict-aliasing –pthread –msse2
```

The application requires support for the SSE2 instruction set at a minimum to support an algorithm optimized using intrinsics oriented toward SSE2.

The default Intel® compiler flags were:

```
icc -O3 -ansi_alias -pthread
```

A challenge in porting applications from one compiler to another is making sure that there is support for the compiler options you use to build your application. The Intel C compiler supports many of the options that are valid on other compilers you may be using, such as GCC. The compiler generates object files that are compatible with GCC-generated object files, so you can compile part of your application using the Intel compiler and the rest using GCC.

The `-fomit-frame-pointer` option is set when you specify option `-O1`, `-O2`, or `-O3` when using the Intel C compiler (so there is no need to include it). The -malign-double option aligns double, long- double, and long-long types for better performance for systems based on IA-32 architecture and is available in the Intel C compiler.

We started the application with this command line:

```
./hmmsearch globins4.hmm ../../
uniprot_trembl.fasta
```

The next step is to locate the hotspots in the application using Intel VTune Amplifier XE. This profiler tool uses low overhead techniques to quickly find multicore performance bottlenecks, without needing to know the processor architecture or assembly code. Note that we do not need to add code to the application to collect data.

To view source code lines of hmmsearch in VTune Amplifier XE, we need to include symbols in the release build—so we add the `–g` flag. We added the `–fno-inline-functions` flag as well; this allows us to see all of the code in question in the VTune Amplifier XE source view.

The VTune Analyzer XE hotspots analysis shows where most of the CPU activity is occurring in the application and the amount of CPU activity on the threads over time. (**Figure 1**)

The VTune Amplifier XE hotspots view tells us that the function consuming the most CPU time is p7_MSVFilter, and double-clicking on the function name displays the SSE intrinsics calls used in optimizing the performance of the function. The assembly view shows us that the Intel compiler utilized vector instructions, but is not taking advantage of the 256-bit registers or AVX instructions on the Intel Xeon processor. (**Figure 2**)

It's possible that we could compile the original C code for p7_MSVFilter with the Intel compiler and help the compiler vectorize the function for the instruction set available on the target machine, so that the function is not limited to using 128 bit registers.

The thread timeline view shows that there is not much CPU time used in the worker threads, but a large amount is used in one thread. This turns out to be the thread that is reading the sequence database file. (**Figure 3**)

"To achieve more significant performance gains, the problem of serialization of the application due to the file read has to be solved."

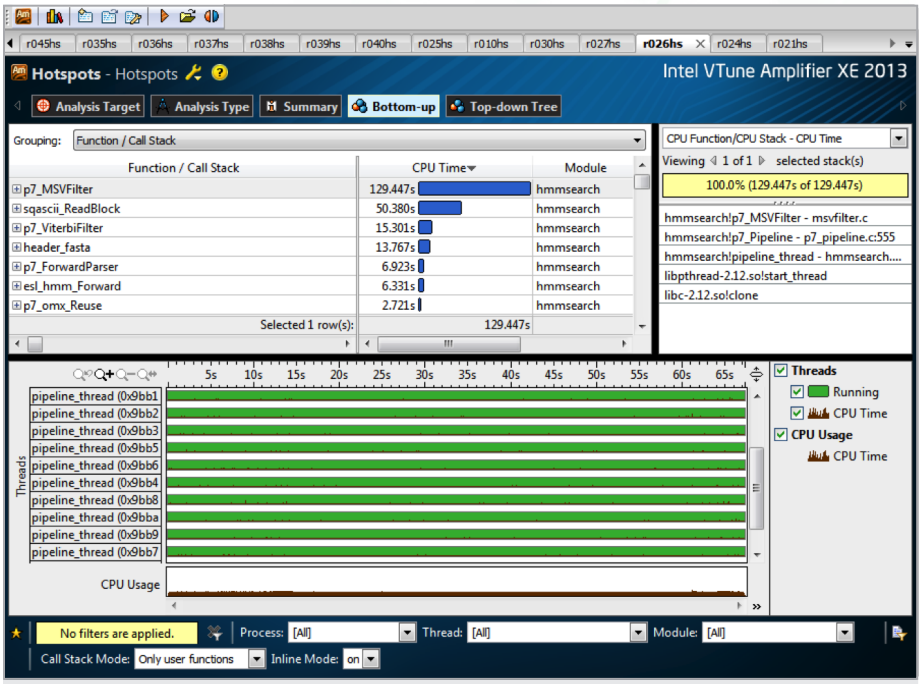

Figure 1

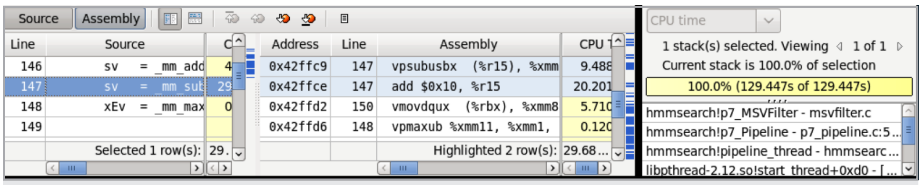

Figure 2

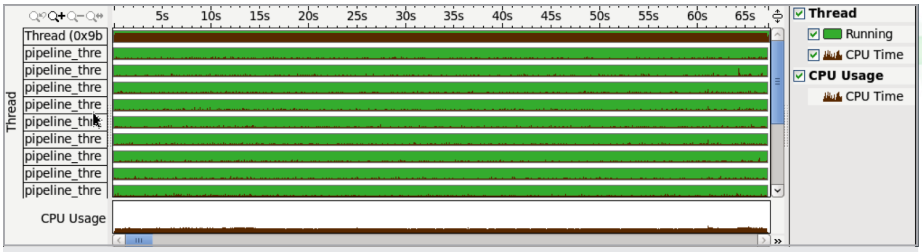

Figure 3

The application creates a number of threads equal to the number of HW threads on the machine plus one, which in the case of a hyperthreaded machine is equal to the number of hyperthreads plus one. In this case, there are 17 threads running. If we use the hmmsearch –cpu 4 flag to limit the threads to five threads, VTune Amplifier XE shows that the application scales well—unlike the situation with 17 threads. (**Figure 4**)

Evidence of this is the 67.418-second runtime with 17 threads, which is worse than the 62.561-second runtime with four threads.

We can see that the top thread is the one reading the data file by filtering the results by thread in the five-thread hotspot display. (**Figure 5**)

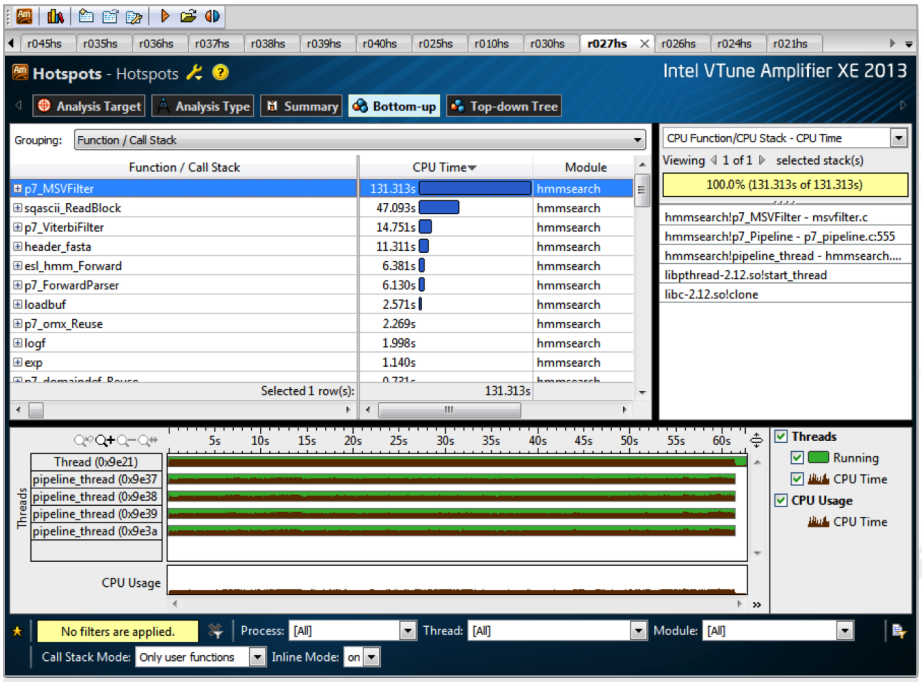

Figure 4

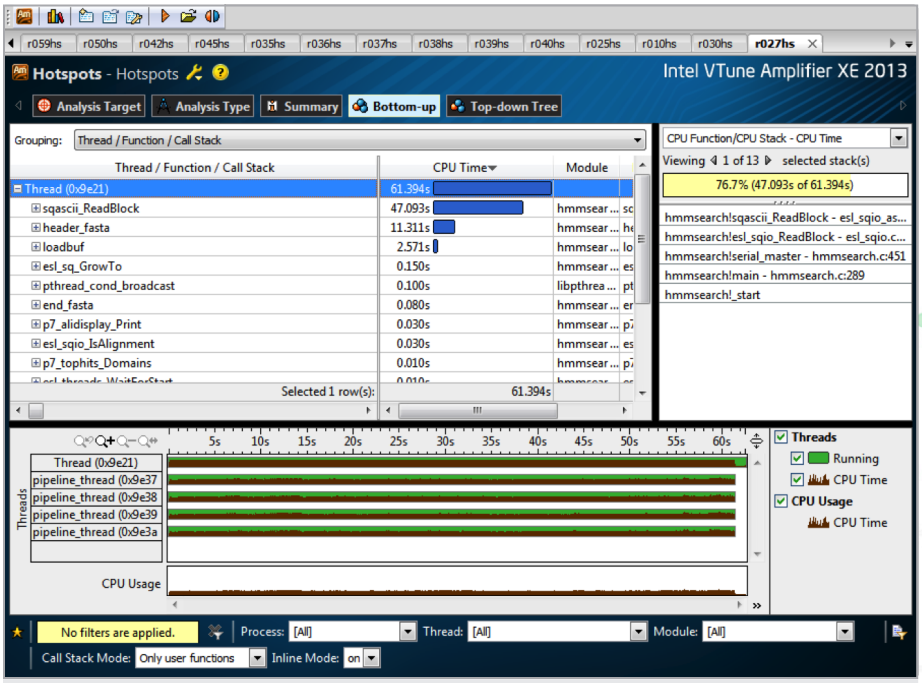

Figure 5

If we use the VTune Amplifier XE Locks and Waits feature on the run with 17 threads it shows us a large number of transitions, indicated by yellow lines from the thread reading the sequence database file to worker threads. (**Figure 6**)

hmmsearch uses a producer consumer model. This is where a producer thread (labeled Thread (0xa0) in the graphic) puts data to be processed on a queue that worker threads (labeled pipline_thread in the graphic) remove when the producer thread signals them with a broadcast message, resulting in a thread transition from the producer thread to the worker thread.

By zooming in, we can see that the amount of thread running time (dark green) is less than thread waiting time (light green), indicating lost time to do productive work. (**Figure 7**)

Compare this with a zoom-in on the thread view for hmmsearch using only four threads. Note that thread transitions from the thread reading the data file, the top thread, typically result in productive work to the worker thread. (**Figure 8**)

However, when using 17 threads in hmmsearch, many thread transitions do not result in work being done. (**Figure 9**)

Zooming in even closer on the 17 thread case, we can see these thread transitions are the result of a pthread_cond_broadcast call that tells the worker threads that a block of data is ready on the work queue to be processed. Only one thread at time can grab the block of data—so the other threads must wait again. (**Figure 10**)

When only five threads are used, only about two threads are waiting to get a block of data to process, and only one thread goes unsatisfied. (**Figure 11**)

All of this indicates that with more than four threads, the hmmsearch pipeline threads become starved for data. In other words, the thread reading the data file cannot provide data fast enough to keep up with computation in the worker threads.

From our analysis using VTune Amplifier XE, we know that the most time-consuming code is the MSV algorithm, which has been optimized with SSE intrinisics in p7_MSVFilter in the file msvfilter.c. The intrinsic-optimized code also contains some optimizations over and above vectorization, so it will be faster.

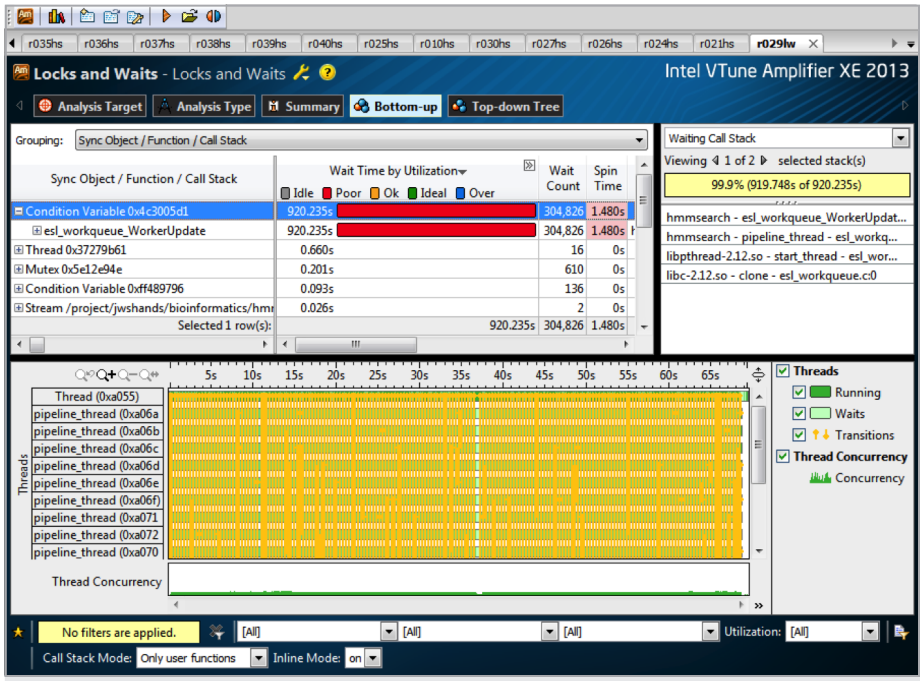

Figure 6

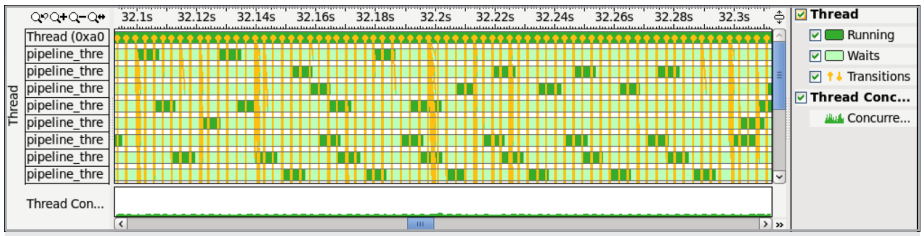

Figure 7

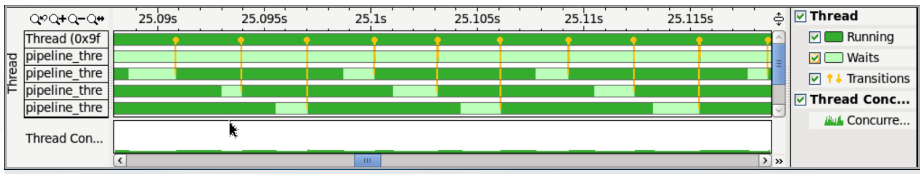

Figure 8

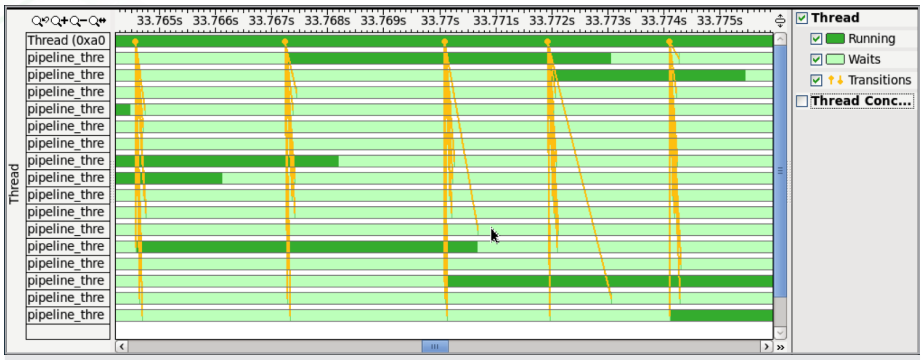

Figure 9

To see if the Intel compiler can effectively vectorize the nonintrinsic optimized code, we compiled the application to use the unoptimized C code in the function p7_GMSV in the file generic_msv.c. VTune Amplifier XE again shows that the MSV algorithm is the hotspot. (**Figure 12**)

VTune Amplifier XE also shows that the most time-consuming part of the MSV algorithm is a single loop that is not taking advantage of AVX instructions or YMM registers on the Intel Xeon processor. (**Figure 13**)

The runtime of hmmsearch using this code is about four minutes and 30 seconds.

```
# CPU time: 4137.39u 5.02s
01:09:02.41 Elapsed: 00:04:30.08
```

If we use the –opt-report flag for the Intel compiler, it will tell us what inlining, loop, memory, vectorization, and parallelization optimizations have been done for each function. For the p7GMSV function, it tells us the loop was not vectorized.

```
generic_msv.c(80:7-80:7):VEC:p7_GMSV: loop
was not vectorized: existence of vector
dependence
```

By restructuring the code, we can enable the compiler to vectorize the loop and generate code that takes advantage of Intel Xeon architecture. The optimization report from the compiler indicates that the two loops resulting from the restructuring were vectorized:

```
generic_msv.c(88:7-88:7):VEC:p7_
GMSV: LOOP WAS VECTORIZED

generic_msv.c(108:7-108:7):VEC:p7_
GMSV: LOOP WAS VECTORIZED
```

In addition, the VTune Amplifier XE assembly view shows that AVX instructions are being used along with the larger YMM registers. (**Figure 14**)

The resulting runtime of the application is close to half of the original runtime.

```
# CPU time: 2207.74u 4.96s
00:36:52.69 Elapsed: 00:02:28.16
```

We can use Intel Inspector XE to check hmmsearch for threading and memory errors. It gives detailed insight into application memory and threading behavior to improve application reliability, and its powerful thread checker and debugger make it easier to find latent errors on the executed code path. Intel Inspector XE also finds intermittent and nondeterministic errors, even if the error-causing timing scenario does not happen.

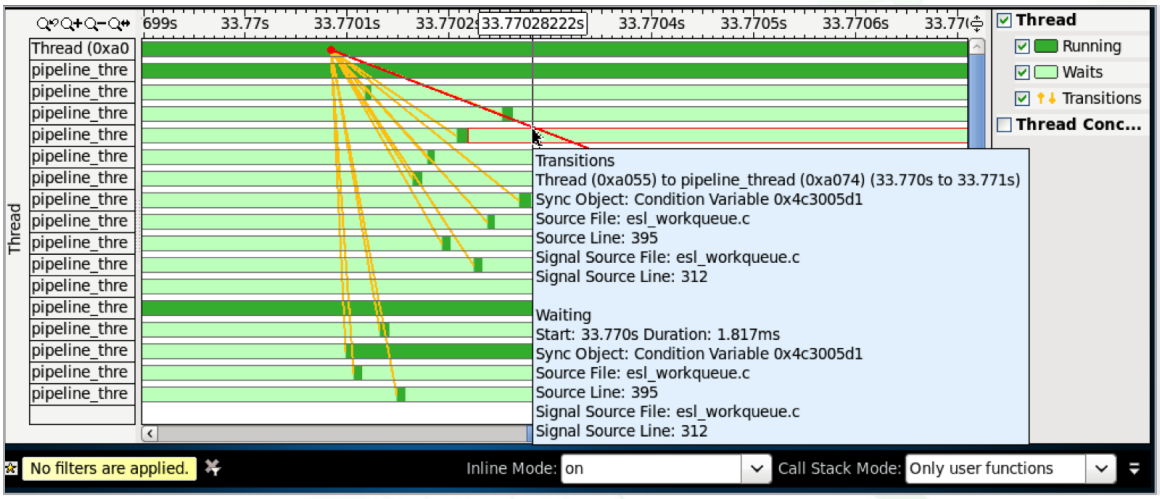

Figure 10

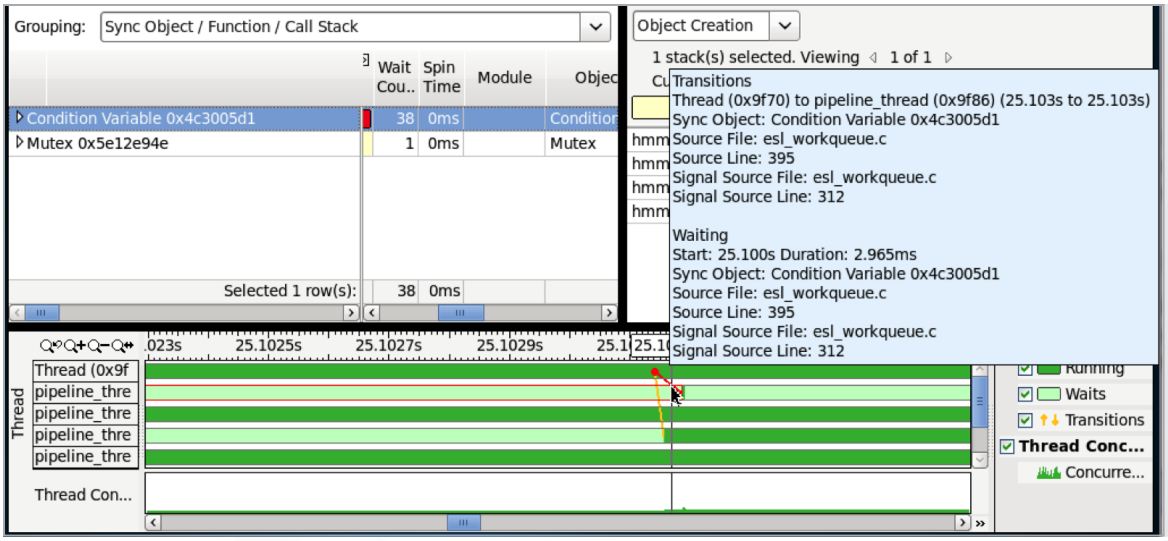

Figure 11

"The Intel® C compiler and libraries create faster code, Intel® VTune™ Amplifier XE finds bottlenecks, and Intel® Inspector XE pinpoints memory and threading errors before they happen. All this is of critical importance when developing applications like HMMER."

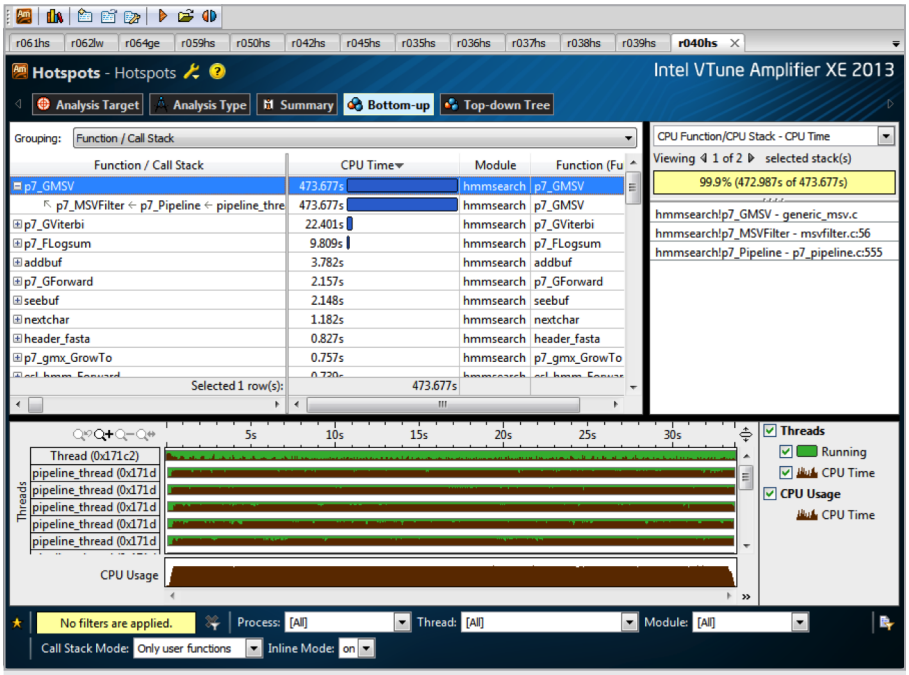

Figure 12

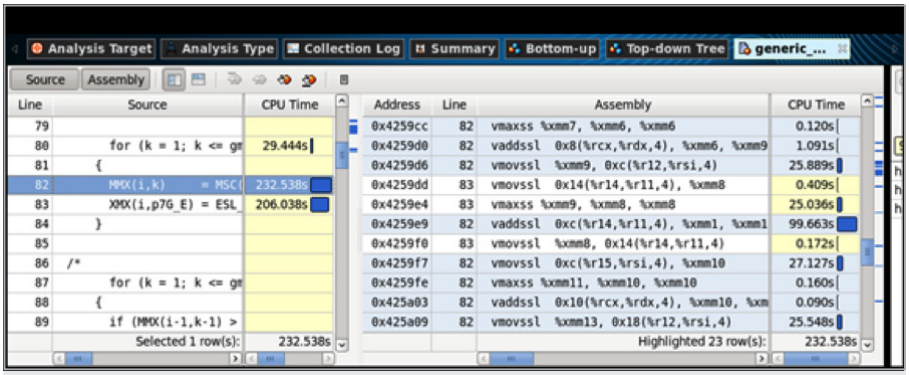

Figure 13

Intel Inspector XE finds memory leaks, corruption, and inconsistent memory API usage, as well as data races, deadlocks, and memory accesses between threads.

As with Intel VTune Amplifier XE, we don't need to create a special build or add code to the application to collect data.

Because there is significant overhead in detecting memory and threading bugs, we launch hmmsearch using a smaller sequence database file, as well as an application option that reduces the number of threads.

When we run Intel Inspector XE in the Detect Memory Problems mode, a few uninitialized memory accesses are exposed. (**Figure 15**)

Right-clicking on a line in the Detect Memory Problems pane brings up a description of an uninitialized memory access problem: (**Figure 16**)

Intel Inspector XE running in Locate Deadlocks and Data Races mode did not detect any issues. (**Figure 17**)

In order to increase application performance, we can take advantage of Intel® Cilk™ Plus in the Intel compiler. Cilk Plus is an extension to C and C++ that offers a quick, easy, and reliable way to improve the performance of programs on multicore processors. It is an open standard and will soon be available in GCC 4.7. Cilk Plus, included in the Intel® C/C++ compiler, allows you to improve performance by adding parallelism to new or existing C or C++ programs using only three keywords: `cilk_for`, `cilk_spawn`, and `cilk_sync`.

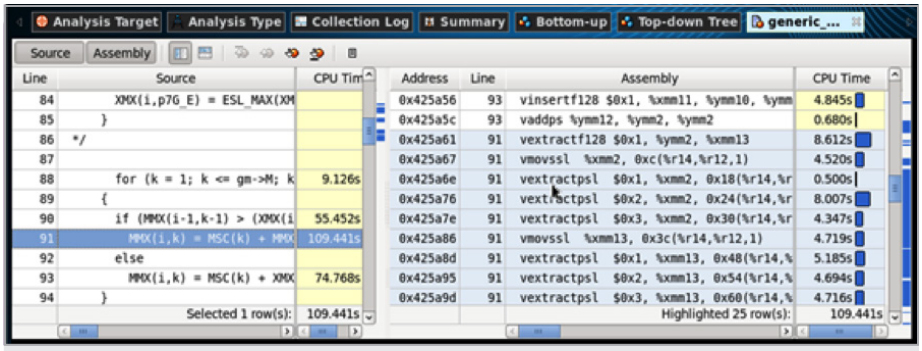

Figure 14

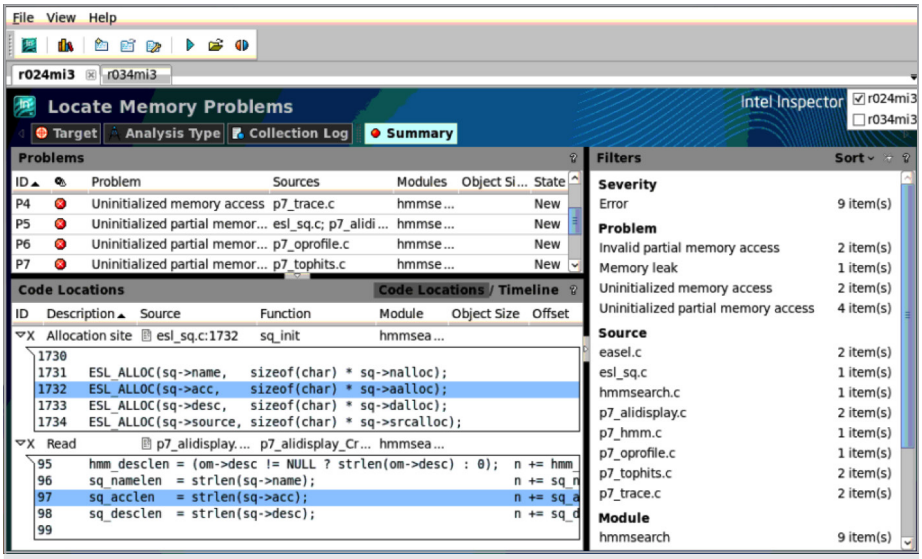

Figure 15

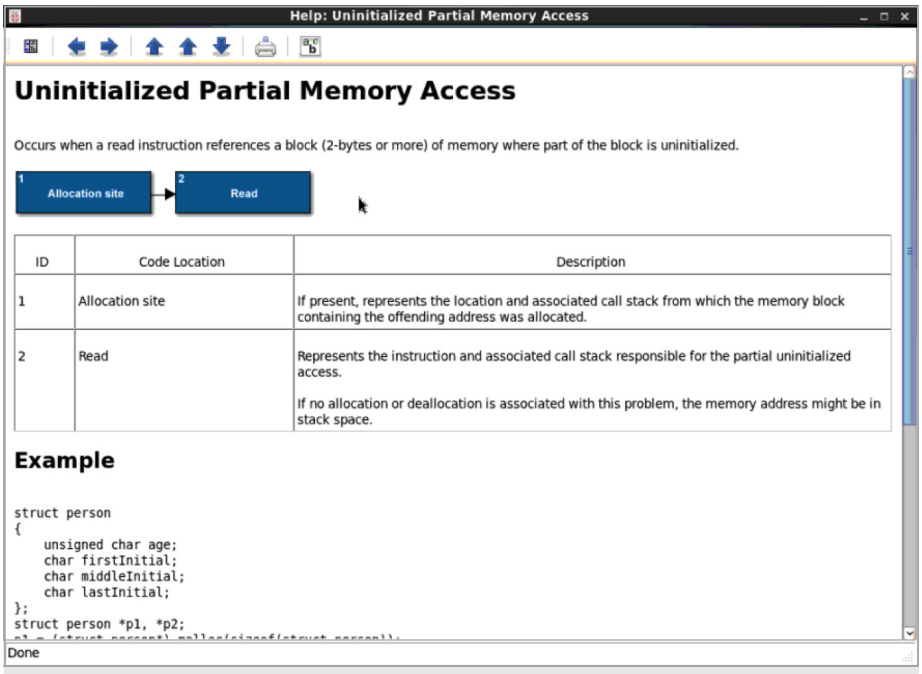

Figure 16

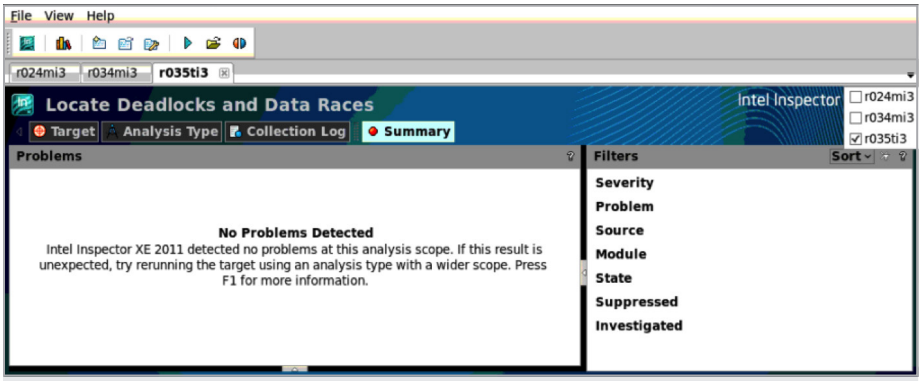

Figure 17

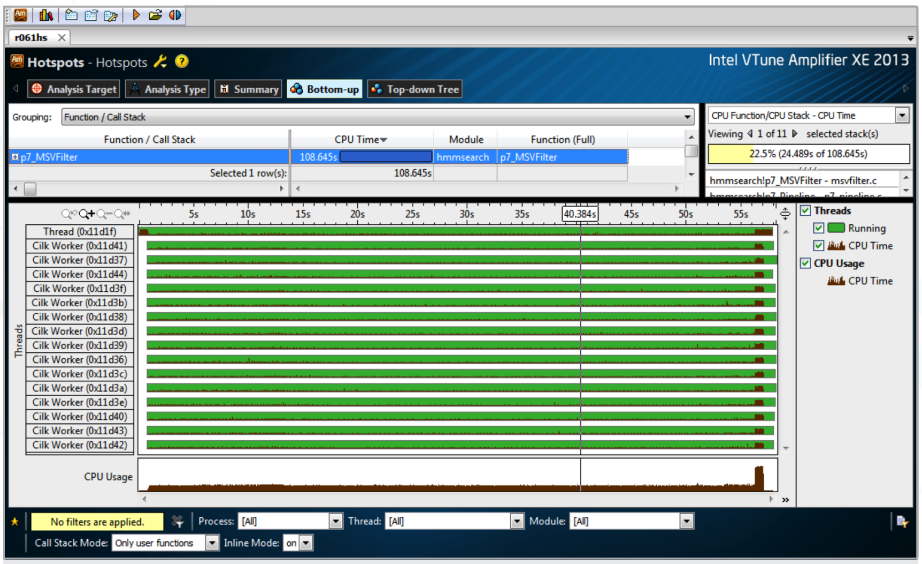

Figure 18

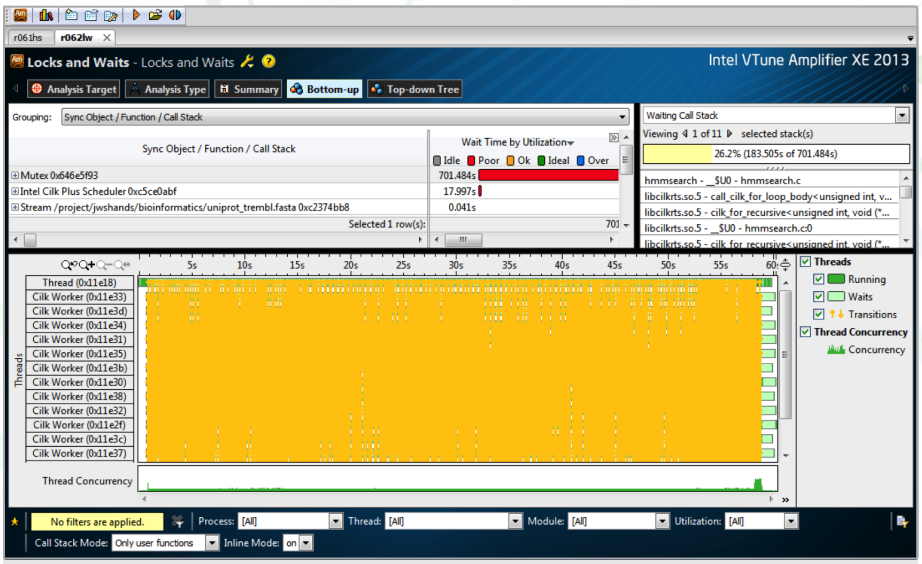

Figure 19

We use Cilk Plus to replace the code that manages threads, mutexes, condition variables, and the work queue with the added benefit of better scheduling. However, we must still synchronize threads on the data file read, which results in serializing a portion of the application.

In the Intel VTune Amplifier XE Hotspots graphic of an hmmsearch run, you can see that because of the synchronization resulting from mutexes around the code reading the sequence database file, the CPUs are not fully utilized. But the Cilk Plus implementation has a shorter runtime at 58.272 seconds compared to the original runtime of 67.418 seconds. (**Figure 18**)

If we run a VTune Amplifier XE locks and waits analysis we can see that there are still many thread transitions. (**Figure 19**)

If we zoom into the thread pane in the locks and waits analysis, we see that the thread transitions are between worker threads, and that they involve the mutex that protects the file read, which is now carried out by each worker thread. (**Figure 20**)

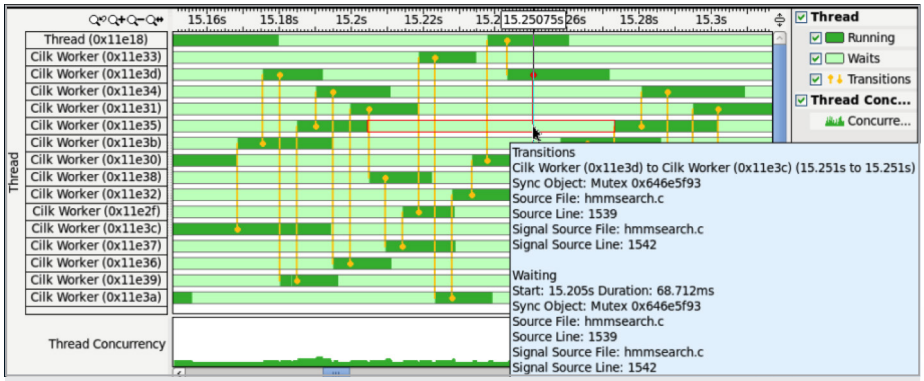

Figure 20

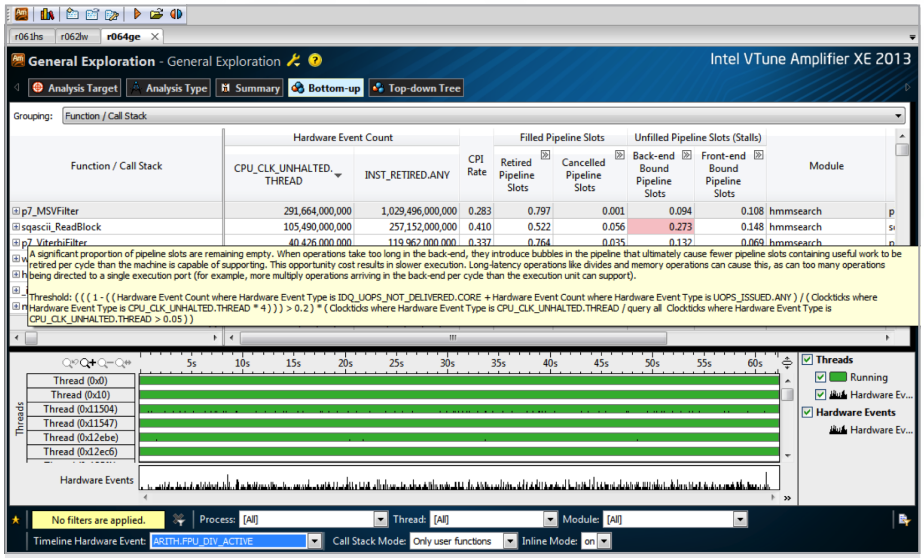

Figure 21

One of the other powerful features of Cilk Plus is the C/C++ language extension for array notations. This Intel-specific language extension provides data parallel array notations, which enable compiler parallelization and vectorization with less reliance on alias and dependence analysis.

To achieve more significant performance gains, the problem of serialization of the application due to the file read has to be solved. Reading the data into memory prior to computation is not realistic when using the uniprot_trembl.fasta data file, because we would exceed memory capacity on our machine, although if enough memory was available it would speed up subsequent computations using the same data.

Further performance gains can be achieved by taking advantage of Intel compiler options. Since the Intel compiler default instruction set is SSE2 and the target machine is Intel Xeon, it would be a good idea to take advantage of AVX instructions and larger register size by using the `-xhost` switch that will generate an instruction set up to the highest level supported on the compilation host.

Another important compiler option is `-ipo`, which enables interprocedural optimization between files. This is also called multifile interprocedural optimization (multifile IPO) or whole program optimization (WPO). When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

For help on finding out what to do to help the Intel compiler vectorize or parallelize loops we can use the `-guide` flag, which provides a report without producing objects or executables. The guided auto-parallelization feature of the Intel compiler is a tool that offers selective advice, resulting in better performance of serially coded applications. The advice typically falls under three broad categories: source code modification, use of pragmas, and addition of compiler options.

Here is one of the suggestions after using the option in hmmsearch:

```
esl_vectorops.c(161):
remark #30536: (LOOP) Add
-fargument-noalias option
for better type-based
disambiguation analysis by
the compiler, if appropriate
(the option will apply for
the entire compilation).
This will improve
optimizations such as
vectorization for the loop
at line 161.
```

Adding the `-parallel` switch allows the Intel compiler to detect simply structured loops that may be executed in parallel, and automatically generates multithreaded code for them. If you use guided auto-parallelization options along with `-parallel`, the compiler may suggest advice on further parallelizing opportunities in your application:

```
msvfilter.c(106): remark #30525: (PAR)
Insert a "#pragma loop count min(1024)"
statement right before the loop at line
106 to parallelize the loop. [VERIFY]
Make sure that the loop has a minimum
of 1024 iterations.
```

We can also use the VTune Amplifier XE hardware event counter collection to get insight into bottlenecks in application code affecting performance. VTune Amplifier XE highlights collected data indicative of performance problems that should be investigated. Here is one example of an hmmsearch run. (**Figure 21**)

Conclusion

Intel® Software Development Tools help you boost application performance and increase the code quality, security, and reliability needed by high performance computing and enterprise applications. The Intel C compiler and libraries create faster code, Intel VTune Amplifier XE finds bottlenecks, and Intel Inspector XE pinpoints memory and threading errors before they happen. All this is of critical importance when developing applications like HMMER for the latest generation of multicore processors. □

Learn how Intel® Advisor XE can help improve parallelization productivity.

BY RAVI VEMURI

What do space exploration, oil and natural gas exploration, Hollywood movies, and military operations have in common? Modeling, simulation, exploration, storyboarding, and reconnaissance are some of the phrases that come to mind. They are intended to reduce the cost of wrong choices, failures, and missteps, and help projects succeed and be more productive.

Software parallelization likewise can also benefit from parallelism reconnaissance in which code is evaluated for suitability for parallelization. Until now, there have been limited tools support to do this. However, Intel® Advisor XE 2013 changes this and helps the world of parallelization leapfrog forward. Intel® Advisor XE is the newest component of the Intel® Parallel Studio XE suite of products.

Software parallelization is potentially destabilizing to code, risky, expensive, and complex. Current trial and error approaches are not productive and there is considerable risk of dead ends. Embarking on code parallelization based on measured data (for example, hotspots) is perhaps better, but is likewise mostly a hit or miss. Code may or may not scale well. Stability issues due to incorrect parallelization also may lurk and surface long after the code is productized, and become costly to fix.

Intel® Advisor XE is built to help you find where to add parallelism to your code. Use it to discover the parallel performance (scalability) and code/data sharing issues (correctness) of possible parallel code regions. It lets you model several different regions within your program at once for parallel scalability and correctness. The results help you make judicious choices about which regions of code to not parallelize (to avoid dead ends), and which regions of code to actually parallelize to reap the multicore performance benefits.

Using this methodology helps you fix data sharing issues *before they happen*. Even as you prepare the code for parallelization by fixing the correctness issues, you can continue to use your existing test frameworks to validate your program—as it remains functionally unchanged and correct.

Use of Intel® Advisor XE in your parallelization efforts is very likely to reduce risk and increase the reward. Moreover, the tool empowers everyone in the software organization with the skill to productively parallelize, instead of the current situation where just the architects and senior engineers have this capability.

You can see how exciting the potential is for your applications. Please explore the product in greater detail at the Intel® Advisor XE product page, and let us know what you think. □

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