Free Intel Webinar for C, C++, and Fortran Developers

Join technical experts for our biannual webinar series, featuring techniques and tools that can help optimize your applications, and increase performance and reliability.

Look for a special, two-day webinar on the new Intel® Xeon Phi™ coprocessor, June 25-26, 2013. This coprocessor, based on Intel® Many Integrated Core (MIC) architecture, is enabling dramatic performance gains on demanding applications.

Embedded and Mobile Systems Developers

Attend focused webinars with the insight that can help accelerate time to market and reliability.
CONTENTS

Letter from the Editor
Crossing the Divide, BY JAMES REINDERS ................................................................. 4

Intel® System Studio: Complete Development Solution for Intelligent Systems, BY NOAH CLEMONS AND ROBERT MUELLER-ALBRECHT ................. 6
Introduces a new integrated software development suite that provides deep hardware and software insights to speed development, testing, and optimization of Intel®-based embedded devices and intelligent systems running embedded Linux®, including Wind River Linux® and Yocto Project®.

Shockwaves! CloverLeaf Meets the Intel® Xeon Phi™ Coprocessor,
BY VICTOR GAMAYUNOV AND STEPHEN BLAIR-CHAPPELL ................................................ 24
Covers the steps taken to make the CloverLeaf benchmark "Intel® Xeon Phi™ coprocessor-ready," and the significant performance gains achieved.

Programming Models for the Intel® Xeon Phi™ Coprocessor and Intel® MPI Library, BY MICHAEL CHUVELEV, DMITRY DURNOV,
SERGEY KAZAKOV, ALEXANDER SUPALOV, AND ANDREW TANANAKIN .............................. 29
Demonstrate how new programming models enabled by the Intel MPI Library, combined with the simplicity of its user interface, facilitates porting existing high performance computing (HPC) applications to Intel Xeon Phi coprocessor-equipped platforms.

From Legacy Serialism to Parallelism: Converting a Real-World Application Using Intel® Cilk™ Plus, BY CHRIS CHRULSKI .................. 32
Using a combination of serial optimizations and parallelism from Intel Cilk Plus, a 25X improvement in runtime is achieved on an important workload when running on an 8-core system—without a significant recoding effort of the legacy serial application.

© 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Intel Core, Intel Inside, Cilk Plus, Pentium, VTune, VPro, Xeon and Xeon Phi 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.

The Parallel Universe is a free quarterly magazine. Click here to sign up for future issue alerts and to share the magazine with friends.
LETTER FROM THE EDITOR

Today's development tools are enabling an unprecedented wave of cross-platform development, application porting and migration, and software innovation for complex embedded and mobile systems. The performance and capability potential is significant, and in this issue our authors share techniques for taking your applications farther—whether your goal is "run anywhere" or HPC acceleration.

The feature article, Complete Development Solution for Intelligent Systems: Intel® System Studio, explores the new cross-development tool suite for intelligent systems running embedded and mobile Linux*. The suite facilitates development of mature, highly optimized custom system software stacks, and of application layer components for the highly integrated complex chipsets found in today's embedded devices and mobile systems.

In Shockwaves! CloverLeaf Meets the Intel® Xeon Phi™ Coprocessor, a small benchmark that solves two-dimensional, compressible Euler equations on a staggered Cartesian grid, is vectorized to take full advantage of the Intel Xeon Phi coprocessor. The result? A performance boost of 80 percent.

Programming Models for the Intel® Xeon Phi™ Coprocessor and Intel MPI Library, looks at how the Intel MPI Library for Linux* can be used to port existing high performance computing (HPC) applications to Intel Xeon Phi coprocessor-equipped platforms.

From Legacy Serialism to Parallelism: Converting a Real-World Application Using Intel® Cilk™ Plus, walks through a high-level workflow and the results achieved when converting a real-world, legacy serial application into a parallel application using Intel® Cilk™ Plus technology.

New tools, like the Intel Xeon Phi coprocessor, can help your applications cross platforms—and define user expectations. As you cross the next barrier to achieving greater capacity and innovation, we support your inspiring efforts.

James Reinders
May 2013
Intel® System Studio:
A Complete Development Solution for Intelligent Systems

by Noah Clemons,
Technical Consulting Engineer, DPD Embedded Computing, Debuggers and Libraries, Intel

and

Robert Mueller-Albrecht
DPD Embedded Computing, Debuggers and Libraries, Intel
Intel® System Studio is a cross-development tool suite for intelligent systems running embedded and mobile Linux*. It facilitates development of mature, highly optimized custom system software stacks, and of application layer components for the highly integrated complex chipsets found in today’s embedded devices. In this article, we’ll take a look at this exciting new product that addresses the needs of the modern embedded software developer. We’ll provide a technical overview showing the components that make up Intel System Studio, and how they can be applied to your intelligent system or traditional embedded device.

If you could improve one thing about your embedded design activities, what would it be?

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugging Tools</td>
<td>22%</td>
</tr>
<tr>
<td>Engineering Team/Skill Levels</td>
<td>16%</td>
</tr>
<tr>
<td>Schedule</td>
<td>15%</td>
</tr>
<tr>
<td>Programming Tools</td>
<td>8%</td>
</tr>
<tr>
<td>Microprocessor</td>
<td>8%</td>
</tr>
</tbody>
</table>

Embedded developers were recently polled on the aspects of embedded tool suites that could most be improved. Debugging tools received the highest ranking. Intel System Studio was created with debugging as a cornerstone, and the suite integrates the following into a single tool suite:

- JTAG debugger for the system software layer
- Application debugger (GDB)
- Low-overhead whole-SoC data event tracing
- SoC-wide power and performance profiling tools
- Memory checking and thread profiling tools
- Industry-leading C and C++ compiler
- Performance libraries

Integrated development environment (IDE) integration for many Intel System Studio components relies on the popular Eclipse IDE.* Software for embedded systems is frequently restricted to work with a specific iteration of a hardware vendor’s platform, and developers are tempted give up portability due to limitations of the target hardware or the software environment.

Intel System Studio allows developers to overcome these limitations by extending the reach of Intel® developer tools. This gives embedded computing developers the option to rely on common or open standards, industry standards, or Intel’s extensive ecosystem and compute continuum.

Debugging Tools for Intelligent Systems within Intel System Studio

Given its focus on embedded cross-development, it should not be surprising that the Intel System Studio comes as a host-side installation, but also includes a system_studio_target.tgz package—comprising those studio components intended for target deployment—to enable remote debug and data collection.

Debug tools fall into three categories:

1. Memory and code correctness checking tools
2. High level language application debug tools
3. System software debuggers

Figure 2 illustrates Intel System Studio and its focus on key elements of embedded software development across the platform software stack, including insight, configurability, reliability, and performance. These elements are central as we look at debug solutions and analyzers. Then, we’ll take an in-depth look at using the compiler and performance libraries in an embedded cross-build environment.
The latter two are focused on real-time debug of software stack components. A fourth type of debug tool is the instrumentation-based event tracing utility. This has found new life because of the complex interactions in heterogeneous multicore SoC chipsets with multiple IP blocks and time-sensitive data passing between devices. Intel System Studio offers the GNU Project Debugger for the application debugger, and the Intel® JTAG Debugger as the system debugger. Software Visible Event Nexus (SVEN), ultra-low overhead instrumentation technology for software debug, is also included.

As a whole, Intel System Studio supports a wide variety of Linux OS hosts and the following JTAG devices:

- Macraigor Systems usb2Demon device
- Intel® XTP-XDP3

The Intel JTAG Debugger (Figure 3) included with Intel System Studio additionally supports Windows* host.

**Advanced system and application-level debuggers for fast issue detection**

<table>
<thead>
<tr>
<th>Intel® JTAG Debugger System and Application Debuggers</th>
<th>In-depth CPV, SoC, and chipset debug, with full register description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Source-level debug of OS kernel software, BIOS, UEFI, firmware, and drivers</td>
</tr>
<tr>
<td></td>
<td>Ultra-fast software event tracing infrastructure</td>
</tr>
<tr>
<td></td>
<td>Enhanced GDB*-based application debugger</td>
</tr>
</tbody>
</table>

**Figure 3. Intel® JTAG Debugger**

**Debugging Firmware, Bootloader, and the OS Layer with the Intel JTAG Debugger**

Every layer of an embedded software stack requires a slightly different approach to debugging. Let's take a look at each software layer—from the firmware all the way up to the application layer—typical challenges, and debug approaches.

For early hardware configuration and board bring-up, Intel System Studio supports JTAG debugging through an Eclipse RCP-based user interface. This interface offers features, including:

- A bitfield editor for the registers that manage device control, processor status, and software status. This editor explains the function of each bit (Figure 3).
- Page table visualization, which shows how virtual addresses map to physical memory
- Access to the general descriptor table (GDT), which describes the executability and writeability of memory segments, and the local descriptor table (LDT), which reserves memory segments for specific programs
- OS memory management and configuration awareness to identify faulty data allocation

Following the boot process from firmware, through the bootloader and the OS, all the way to device drivers, let’s begin with using the Intel JTAG Debugger for the firmware (Figure 4).
Figure 4. Intel® JTAG Debugger visualizes device registers and memory mapping.
Unified Extensible Firmware Interface (UEFI) Debug

Developers may want to customize the device firmware to speed up boot time and, specifically, turn on support for only the needed peripherals. This is done with UEFI-compatible firmware solutions, such as the Intel® Firmware Support Package or the Intel® Bootloader Development Kit. To debug issues in firmware raises some challenges, as the UEFI environment uses relocatable code modules, and the addresses of these modules are usually not known to the end users. UEFI firmware can be compiled with either the Microsoft* C/C++ compiler or the Intel® C/C++ compiler. Because debugging of firmware relies on PDB symbol information being resolved and mapped to source code, it's necessary to use the Windows*-hosted version of the Intel JTAG Debugger (also part of Intel System Studio) for firmware debug.

Two general methods for locating code modules in memory are provided by the Intel JTAG Debugger:

1. Identify the specific module located at a certain memory address (for example, at the instruction pointer)
2. List all modules that are known by the EFI runtime, so that you can choose a specific module to load symbols for

The first class of methods relies only on the module of interest. The debugger can scan memory near the given address and attempt to locate the module header, which contains sufficient data to load symbols.

To load symbols for a specific address, use the EFI "LOADTHIS" command. This command identifies the module at the indicated address and loads symbols for it. If no address is provided, the current instruction pointer is used.

```
xdb> efi “loadthis”
INFO: Software debugger set to: efi64 - EFI/PI compliant BIOS (64-bit mode)
INFO: Using DRAM search semantics, align=0x00001000 range=0x00100000
INFO: Searching backwards from 0x00000000809FB6C3 to 0x00000000808FB6C3 for PE/COFF header
INFO: Found PE/COFF module at 0x00000000809FB000 - 0x00000000809FF1C0 (size: 16832 bytes)
INFO: Loading debug symbols found at:
e:\dev.efi\work\Build\MdeModule\NOOPT_VS2008x86\X64\MdeModulePkg\Application\xdbefiutil\xdbefiutil\DEBUG
\xdbefiutil.efi
```

The second class of methods involves locating an EFI data structure in memory (or flash) and enumerating the list of loaded modules described there. It is dependent on a priori knowledge of certain data structures, such as the EFI System Table Pointer or the Flash Volume.

Device Drivers

Device drivers are another major challenge for debugging because these drivers are often timing-sensitive. Thus, adding instrumentation to driver code can change its behavior. To address this challenge, Intel System Studio provides an OS- and driver-aware kernel module to be loaded on the target device for instrumentation-free debugging. At device driver load time, this kernel module exports the memory location of the driver’s initialization and destruction methods to the host via the JTAG interface. It is thus possible to load the symbol info for the device driver, step into it, and debug its execution flow without modifying or instrumenting any of its code. This avoids the risk of changing timing behavior of the driver code.

In addition, the bitfield editor can access public device registers, permitting monitoring of device configuration register entries during device driver debug.
Instruction Tracing

The ability to track errors back to their source is the essence of debugging. Intel System Studio provides advanced instruction tracing to unroll execution flow and identify the root causes of runtime issues. Specifically, the tool inspects the last branch records (LBR) and disassembles the code to recreate program flow. It then pairs the assembly instructions with the associated source code (obtained from the ELF Dwarf executable in the case of embedded Linux), and displays the resulting trace in the debugger (Figure 5).

Tracing does not impede real-time performance. Therefore, it’s a powerful tool for tracking deterministic and repeatable errors, such as stack overflow or segmentation fault. It can be used to:

- Set breakpoints in the OS signal event handler (e.g., break on segmentation fault)
- Unroll execution flow leading up to an event
- Follow execution backward to where it deviated from expectation
- Rerun to that point and analyze memory accesses

Figure 5. The instruction trace recreates program flow
Debugging Non-Deterministic and Hard-to-Replicate Issues with SVEN

Modern intelligent systems are very sensitive to precise timing, especially when they are heavily threaded or rely on message- and data-passing events between software modules. These systems often encounter non-deterministic issues that are difficult to reproduce.

Debugging these systems can be tricky. Debugging code can impact the timing of the software stack, altering application behavior and making issues disappear during a debug session, also called “Heisenbugs.” The problem is particularly severe for issues that only appear when the device is deployed in the field where the actual device may be inaccessible.

The solution to many “Heisenbug” problems is a new component unique to Intel System Studio called SVEN software development kit (SDK) technology preview. SVEN defines a data event as a small instrumentation structure complete with event ID, time stamp, event type, and payload. Any kind of variable propagation or message passing in the program execution flow can be associated with a SVEN event, and thus tracked. SVEN relies on static code instrumentation and a small DRAM buffer with less than 5 µs timing overhead, thereby minimizing opportunities for Heisenbugs. SVEN enables developers to identify timing-dependent runtime issues that defy traditional methods.

The instrumentation code can stay in production code and only impact execution when logging is active. Thus, SVEN is well-suited for offline debug of applications deployed in hard-to-access locations.

Originally developed for Intel® Atom™ processor CExxxx-based platforms, SVEN is a field-proven technology that is now available across the full line of Intel® architecture. It traces asynchronous message and data event propagation throughout the chipset, is highly configurable, and can be used to instrument system software, as well as applications. Developers simply need a reliable clock signal to correlate event timing and they can get started (Figure 6).

Intel System Studio provides the SVEN framework in the form of an open source SDK, as well as a graphical trace viewer for easy navigation of events and timing for quick identification of irregularities. Furthermore, JTAG support introduces data breakpoints that allow system-level debugging. This capability allows triggering on any SVEN event, and stepping and debugging from a suspicious event. Example breakpoints include:

- Break on any event from the USB driver
- Break on any Debug String that starts with "ERROR"
- Break if register X is accessed
- Break if specific bits of register X have a particular value

![Figure 6. SVEN visualizes events across the software stack](image)

**The Parallel Universe**
Application Debugging with GDB

Intel System Studio comes with its own GNU Project Debugger (GDB) that adds data race detection capabilities for Intel architecture-based platforms, as well as branch trace storage (BTS)-based process-specific instruction flow unrolling for Intel Atom processor-based environments. To use this enhanced GDB, instead of the default debugger provided by your Linux distribution, simply source the following debugger environment setup script:

```
<install-dir>/system_studio_2013.0.xxx/debugger/gdb/bin/debuggervars.sh
```

Just like the standard GDB, it can be integrated into the Eclipse IDE and supports the remote system explorer (RSE) plugins (http://download.eclipse.org/tm/downloads/).

To support gdbserver-based cross-debug from within Eclipse:

1. Copy the gdbserver provided by the product installation
   `<install-dir>/system_studio_2013.0.xxx/debugger/gdb/<arch>/python/bin/` to the target system and add it to the execution PATH environment variable on the target.
2. Configure Eclipse to point to the correct GDB installation:
   a. Inside the Eclipse IDE click on Window>Preferences from the pull-down menu.
   b. Once the preferences dialogue appears, select C++>Debug>GDB from the tree view on the left.
   c. The GDB executable can be chosen by editing the "GDB debugger" text box.

Point to `<install-dir>/system_studio_2013.0.xxx/debugger/gdb/<arch>/python/bin/`, where `<arch>` is ia32 or intel64 and `<python>` is py24, py26, or py27, depending on architecture and Python* installation.

Using GDB to Debug Applications on Embedded Devices

Please refer to GDB: The GNU* Project Debugger (http://www.gnu.org/software/gdb/) for more details. GDB comes with a remote debug agent called gdbserver for cross-development. This debug agent can be installed on the debug target to launch and attach to a debugger remotely from the development host. The agent can be useful in situations where the program needs to be run on a target host that is different from the host used for development, particularly when the target has a limited amount of resources (either CPU and/or memory). Here is how you would use the debugger in that scenario:

- Start your program using gdbserver on the target machine.
- gdbserver then automatically suspends the execution of your program at its entry point, waiting for a debugger to connect to it.
- The following commands start an application and tell gdbserver to wait for a connection with the debugger on localhost port 2000.

```
$ gdbserver localhost:2000 program
Process program created; pid = 5685
Listening on port 2000
```

Once gdbserver has started listening, we can tell the debugger to establish a connection with this gdbserver, and then start the same debugging session as if the program was being debugged on the same host, directly under the control of GDB.

```
$ gdb program
(gdb) target remote targethost:4444
Remote debugging using targethost:4444
0x0000000000000000 in ?? ()
from /lib64/ld-linux-x86-64.so.
(gdb) b foo.adb:3
Breakpoint 1 at 0x401f0c:
file foo.adb, line 3.
(gdb) continue
Continuing.
```

“Intel System Studio provides deep hardware and software insights to speed development, testing, and optimization.”
It is also possible to use gdbserver to attach to a program that is already running, in which case the execution of that program is simply suspended until the connection between the debugger and gdbserver is established. The syntax would be:

```bash
$ gdbserver localhost:2000 --attach 5685
to tell gdbserver to wait for GDB to attempt a debug connection to the running process with process ID 5685
```

### Data Race Detection

The Intel System Studio-enhanced build of GDB can break on data sharing events and possible data races in real time, as soon as they are encountered. The only prerequisite is that the code is compiled with the option `--debug parallel` using the Intel® C++ Compiler. Smart detection filters allow the user to focus on areas where threading is suspected, to introduce a data race, so they can pin down the exact location for this error.

A data race occurs when multiple threads access overlapping memory without synchronization. Although data races may be harmless or even part of the design in some cases, a data race typically indicates a bug. GDB may be used as a front end for the parallel debug extension (PDBX) data race detector that is part of the Intel® compiler. The PDBX data race detector consists of the compiler instrumentation and a runtime support library. Both are provided by the Intel compiler. The PDBX runtime library provides a debugger interface for communicating detected data races, as well as for configuring the analysis, again by compiling with the option `--debug parallel`.

The PDBX data race detector logs all memory accesses and synchronization for all threads. It checks for data races on each memory access. As expected, this is very expensive, both in terms of performance and memory consumption. To mitigate the expense, you can fine-tune the data race analysis. While you cannot control logging of synchronization events, you can control logging of memory accesses. Memory accesses are logged so long as the data race detector is enabled, and because the data race detector always logs synchronization, it may be disabled and re-enabled at any time. Of course, the detector is only able to detect data races on known memory accesses. In addition to selective enabling, you can get the data race detector to discard the memory logs it has collected, which may be useful when debugging data races in separate parts of the program under memory constraints.

You can configure the data race detector to ignore parts of the application that may be useful under a variety of different use cases, such as:

- **Ignore false positives**: The data race detector may incorrectly report correctly synchronized accesses as data races. This occurrence is typically caused by a synchronization construct that is not known to the data race detector. It may also be caused by partially instrumented applications.
- **Ignore intended or harmless data races**: In some cases, data races may be harmless or even intended. Examples are data races where all threads are guaranteed to write either the same or an equivalent value. Accepting such a harmless data race is typically preferable to synchronizing the threads.
- **Ignore currently irrelevant data races**: The data race detector may correctly report data races that are not related to the bug currently under observation. Such distracting data races may be ignored until a later time.
- **Improve data race analysis performance**: Ignoring parts of your program may have a significant impact on the analysis performance overhead. It is typically preferable to repeatedly analyze a small portion of your program, than to analyze the whole program.
- **Improve data race analysis memory consumption**: Ignoring parts of your program may have a significant impact on the analysis memory consumption. This allows data race detection to be used in scenarios where a whole program analysis would not be feasible.

### Branch Instruction Tracing

Intel architecture on the Intel Atom processor offers a feature called Branch Trace Store (BTS), which stores a log of branches into an OS-provided ring buffer. The GNU/Linux operating system supports this feature (since version 2.6.32) as part of the perf_event interface.

The gdb extension for branch tracing is based on the hardware BTS feature, making it very useful to debug problems that do not immediately result in a crash. It is particularly useful for bugs that make other debugger features fail; for example, a corrupted stack that breaks unwinding. You can use the `gdb branch tracing commands to record program control flow and view the recorded branch trace as a:

- List of blocks of sequential execution (list view)
- Disassembly of one of the listed blocks

Branch tracing may be less powerful when compared to reverse debugging, but it is considerably faster. In addition, the list view provides a quick overview of where you are, and is therefore comparable with the backtrace command, thus providing a really good alternative to backtrace when the program callstack is corrupted due to optimization artifacts or the OS signal handler truncating it.
Example: Your program has crashed and backtrace is not much help:

```
(gdb) run
Starting program: ../gdb/trace/examples/function_pointer/stack64  Program received signal SIGSEGV, Segmentation fault. 0x000000000000002a in ?? ()
(gdb) bt
#0  0x000000000000002a in ?? ()
#1  0x0000000000000017 in ?? ()
#2  0x0000000000000000 in fun_B (arg=0x4005be) at src/stack.c:32
#3  0x0000000000000000 in ?? ()
```

Look at the branch trace.
Here is the list of blocks, starting from the most recent block (ending at the current pc) and continuing toward older blocks such that control flows from block \( n+1 \) to block \( n \).

```
(gdb) btrace list 1-7
in ?? ()
in fun_B () at src/stack.c:36-37
in fun_B () at src/stack.c:32-34
in main () at src/stack.c:57
in fun_A () at src/stack.c:22-25
in fun_A () at src/stack.c:18-20
in main () at src/stack.c:51-56

from main(), we called first fun_A() and then fun_B(). The call to fun_A() returned, and we crashed somewhere in fun_B().
```

Look at the disassembly of the last three blocks in the original control flow (i.e., reverse trace) order, starting from the call to fun_B() from main().

```
/m interleaves source info
(gdb) btrace /m 1-3
src/stack.c:32  static long fun_B(void* arg) {
  0x000000000040050e <fun_B+1>: mov  %rsp,%rbp
  0x0000000000400511 <fun_B+4>: mov  %rdi,-0x18(%rbp)
src/stack.c:33  struct B_arg* myarg = arg;
  0x0000000000400515 <fun_B+8>: mov  -0x18(%rbp),%rax
  0x0000000000400519 <fun_B+12>: cmpq $0x0,-0x8(%rbp)
src/stack.c:34  if (!myarg) return -1;
  0x000000000040051d <fun_B+16>: cmpq %eax,-0x8(%rbp)
  0x0000000000400522 <fun_B+21>: jne  0x40052d <fun_B+32>
src/stack.c:36  return myarg->arg1 + myarg->arg2;
  0x0000000000400526 <fun_B+32>: mov  -0x8(%rbp),%rax
  0x0000000000400531 <fun_B+36>: mov  (%rax),%rdx
  0x0000000000400534 <fun_B+39>: mov  -0x8(%rbp),%rax
  0x0000000000400538 <fun_B+43>: mov  0x8(%rax),%rax
  0x000000000040053c <fun_B+47>: lea  (%rdx,%rax,1),%rax
src/stack.c:37  }
  0x0000000000400540 <fun_B+51>: leaveq
  0x0000000000400541 <fun_B+52>: retq
0x000000000000002a: Cannot access memory at address 0x2a
```

fun_B() is executed and returns to an invalid address suggesting a corrupted stack. fun_B() leaves, but there was no corresponding push on entry to fun_B() => the function pointer comp that was called in main() was corrupted.
Intel System Studio Analyzers

The aforementioned debug tools represent one approach to ensure reliability and maturity of an embedded platform. Code analysis through the Intel® Inspector for Systems provides an additional set of methodologies for identifying memory issues and concurrency issues. Using the Intel® VTune™ Amplifier for Systems addresses not just the need for reliability, but also for responsiveness, performance, and power consumption control on the embedded platform. Both tools are designed to correspond to Intel System Studio’s methodology of actively supporting the embedded cross-development paradigm and being aware of possibly limited target resources.

<table>
<thead>
<tr>
<th>Intel® Inspector for Systems Memory and Threading Analyzer</th>
<th>Quick finds memory leaks, invalid access, plus data races and deadlocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Executes fast and effective static and heap growth analysis to expose critical defects</td>
</tr>
<tr>
<td></td>
<td>Supports remote data collection, debugger breakpoints, and break on selected errors</td>
</tr>
</tbody>
</table>

Intel Inspector for Systems includes an additional command-line only version in the system_studio_target.tgz package to be used on user interface and memory-restricted platforms. Data analysis can be either done via command line on the development target, or by copying the sampling results data over to the development host and analyzing it using the full Intel® Inspector graphical user interface (GUI).

To perform a level 2 memory checking analysis on an embedded target device or virtualized platform, one would run the command line Intel Inspector using the following set of commands:

```
$ source /home/root/inspector_for_systems/inspxe-vars.sh
$ inspxe-cl --no-auto-finalize --collect mi2 ../<binary-name>
```

Those commands will create a sampling results directory named ../r000mi2. To get source level information about the location of a memory leak or stack overflow, it's required to compile your optimized code with symbol info enabled (-g).

To view these results in Intel® Inspector 2013 for Systems on your Linux host (Figure 7):

a. Copy the results directory from the target device to the development host.

b. Source /opt/intel/system_studio_2013.0.xxx/inspector_2013_for_systems/inspxe-vars.sh

c. Run

```
$ inspxe-gui ~/<results-location>/test/r000mi2
```

![Figure 7. Intel® Inspector for Systems identifying memory allocation issue](image-url)
**Intel® VTune™ Amplifier for Systems**

Intel VTune Amplifier for Systems enables power usage analysis, as well as performance analysis for the software components running on your embedded device. Developers can identify code hotspots where optimization efforts should be spent, as well as architectural resource constraints causing cache misses or excessive memory accesses. This tool also allows identifying those functions and code lines in system or application code that cause wake-up events or processor frequency changes due to frequent IO device pings or similar activities. In the system_studio_target.tgz package, one finds two target components for data collection. For “architectural events only” sampling, the Intel® VTune™ Amplifier Sampling Enabling Product (SEP) is provided. For remote data collection, the Intel® VTune™ Amplifier Command Line Data Collector amplxe-runss.py is provided. **Figure 8** illustrates the basic architecture of the amplxe-runss remote data collection model using SSH protocol.

<table>
<thead>
<tr>
<th><strong>Power Optimization and Efficiency</strong></th>
<th>Intel® VTune™ Amplifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides in-depth SoC wide visibility of events and analysis of CPU and GPU activities</td>
<td></td>
</tr>
<tr>
<td>Identifies wake-up causes, timers triggered by application, and interrupts mapped to hardware interrupt level</td>
<td></td>
</tr>
<tr>
<td>Displays CPU core frequencies and events that wake up the processor at the source code level</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Performance Optimization</strong></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th><strong>Power and Performance Profiling Tool</strong></th>
<th>Analyzes system and SoC events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displays hardware events with call stacks, provides lower overhead sampling, and finds hotspots in small functions</td>
<td></td>
</tr>
<tr>
<td>Provides statistical call counts for better data for in-lining and parallelization decisions</td>
<td></td>
</tr>
<tr>
<td>View results in source or assembly; no instrumentation required</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 8.** Intel® VTune™ Amplifier for systems host and target relationship
Collecting performance data using this remote data collection model follows the steps outlined below after installation is complete:

Set up your target to not require a password:

1. Make sure your root directory has a .ssh directory and only the owner had read/write/execute access on it.
2. cat ~/.ssh/id_dsa.pub | ssh root@ip_target “ cat >> /home/root/.ssh/authorized_keys”
3. On the target chmod 600 /home/root/.ssh/authorized_keys
4. Verify you can now log in without password
   a. ssh root@target_ip

Run the Intel VTune Amplifier 2013 for Systems remote collector on your application:

1. Set up some target environment variables
   a. export AMPLXE_TARGET_PRODUCT_DIR=/home/root/linux32
      i. From the location you copied the remote collection binaries
   b. export AMPLXE_TARGET_TMP_DIR=/tmp
2. Run collection
3. amplxe-runss.py --target=user@target -r result@@@ --duration 10 --no-pmu-stack
   --event-config CPU_CLK_UNHALTED.CORE:,CPU_CLK_UNHALTED.REF

To collect wake-up event and frequency change data you would run this command instead:

```
amplxe-runss.py --target=root@ip_target --pwr-config=sleep,frequency --result-dir r@@@ -- /tmp/hello
```

This command will run a remote power collection on your target, and it will run your target application /tmp/hello. It will also create a result directory r000 that you can open on your host.

On your Linux host you can view the Intel VTune Amplifier 2013 for Systems results (Figure 9) using:

```
(amplxe-gui r000)
```

Figure 9. Intel® VTune™ Amplifier for systems power state analysis
Compiler and Libraries within Intel System Studio

The Intel C++ Compiler supports cross-compilation and integration with Poky-Linux* or OpenEmbedded*-compatible GNU cross-toolchains as used for Wind River* Linux*, Yocto Project* and many other custom GNU cross-build toolchains. It comes with predefined compiler environment files that make using the Intel C++ Compiler for cross-development as easy as applying the –platform=y13 or –platform=wrl50 option switch in the case of Yocto Project* 1.3 targets using the application development toolkit of Wind River Linux 5.0 targets.

For compiler integration into the Eclipse IDE, Intel System Studio comes with an automated Eclipse IDE integration script: eclipse_integration.sh. Another idea Intel System Studio expands upon is easy access to the key features of the Intel C++ Compiler and other components through an IDE commonly used throughout the industry. Running the integration script creates a new Intel Tools menu entry, with access to the Guided Auto-Parallelism, Intel C++ Compiler selection and an Intel® C++ Compiler environment file editor. This environment file editor (Figure 10) is a unique feature of Intel System Studio, allowing customization of the integration of the Intel C++ Compiler into a GNU cross-build toolchain, not just via command line editor, but also from within an IDE.

Industry-leading high-performance C++ Compiler and libraries enables the performance and scalability benefits of Intel processors

<table>
<thead>
<tr>
<th>Intel® C++ Compiler 13.0 and Libraries</th>
<th>Generates faster code through outstanding speed optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enables shortest execution times for developing low-power applications</td>
</tr>
<tr>
<td></td>
<td>Supports GNU* cross-build, integration into Eclipse CDT*, and Yocto Project* Application Development Toolkit</td>
</tr>
<tr>
<td>Intel® Cilk™ Plus to easily utilize performance boosting multicore capabilities with three simple keywords</td>
<td></td>
</tr>
<tr>
<td>Intel® Math Kernel Library (Intel® MKL) provides highly optimized threaded math routines for Intel® Core™ and Intel® Xeon® processors</td>
<td></td>
</tr>
<tr>
<td>Intel® Integrated Performance Primitives (Intel® IPP) is an extensive library of highly optimized software building blocks for the most demanding signal, data, and multimedia applications</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. Compiler Eclipse IDE* integration and environment file editor.
Intel Cilk Plus (included with the compiler) enables forward-scaling task parallelism for C and C++ with a runtime-dynamic scheduler that maps an arbitrary number of tasks to a limited set of workers (pool of threads). This feature allows for composable designs where multicore parallelism can be added without supervising call chains (nested parallelism) and oversubscribing resources. Intel Cilk Plus guarantees to enable parallelism according to the number of workers in the pool instead of an unbound resource usage according to the number of scheduled tasks. This guarantee is especially useful when one has long-term scalable parallelism implementations and resource-conscious embedded designs (Figure 11).

Intel® Math Kernel Library (Intel® MKL) and Intel® Integrated Performance Primitives (Intel® IPP) included in Intel System Studio provide unique approaches to highly optimized data- and signal processing on intelligent systems. Intel IPP is a C Library with ready-to-use functions for a wide variety of domains covering a broad range of data types. Intel IPP focuses on in-core optimizations, such as cache-blocking, and continuously adopts new instruction set extensions such as Intel® Streaming SIMD Extensions (Intel® SSE). Prominent capabilities include upcoming support for Intel® Advanced Vector Instructions 2 (Intel® AVX 2).

```c
for (size_t i = 0; i <= nsteps && (0 < stage[1].n || 0 == i); ++i) {
    cilk_spawn read_signal   (size, x + stage[0].i, y + stage[0].i, stage[0].n);
    cilk_spawn process_signal(stage[1].n, x + stage[1].i, y + stage[1].i,
                                x + stage[2].i, y + stage[2].i);
    print_signal  (stage[3].n, x + stage[3].i, y + stage[3].i, std::cout);
    cilk_sync;
    stage[2].n = stage[1].n;
    std::rotate(stage, stage + 4 - 1, stage + 4);  // quad-buffering
}
```

Figure 11. Example of Intel® Cilk™ Plus syntax.

Optimized functions tailored for all Intel® Embedded processors

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Intel® MKL provides highly optimized threaded math routines for Intel® Core™ and Intel® Xeon® processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel® IPP is an extensive library of highly optimized software building blocks for the most demanding signal, data, and multimedia applications</td>
</tr>
</tbody>
</table>

**Signal Processing (1d)**
- Transforms (e.g., Wavelet)
- Convolution/Correlation
- Filtering (e.g., IIR, FIR)
- Statistics

**Image Processing (2d)**
- Transforms (e.g., rotation)
- (Non-)lin. Filter (e.g., noise)
- FFT, DFT, DCT
- Statistics

**Color Conversion**
- Color space conversion
- Pattern (e.g., Bayer)
- Brightness/Contrast
- Resampling

**Vector/Matrix**
- Logical, shift, conversion
- Trigonometric functions
- Decomposition, Eigenvalues
- Transpose

**Integrit Compression/Cryptography**
- Error correction, Reed-Solomon
- Compression (entropy, dct.)
- MD5, T(DES), RSA, DSA
- Random number generators

**More Domains**
- Video, picture coding
- Audio (e.g., speech coding)
- String processing
- Utilities

Figure 12. Various domains covered by Intel® IPP.
Intel IPP is supplied as a sequential library (library name postfix is ".l" for "linear") due to higher efficiency for smaller data sets, latency constraint applications, or for better control via application-level threading. In order to support multi-threading, Intel IPP is fully thread-safe. The main sample collection of Intel IPP (extra download) contains an “advanced usage” category with a sample driver, along with an application requesting this driver’s service. Such an example may help to develop code that runs in kernel mode. Intel IPP provides libraries that are not position-independent (“nonpic”) in order to support code that runs in kernel mode (ring 0).

The function names in Intel IPP follow a scheme that consist of a prefix indicating the library domain (e.g., “ipps” for signal processing), as well as a postfix that indicates the data types involved (e.g., “f32” in case of single-precision floating point). C-language call-convention and linkage (undecorated functions names) allow for calling Intel IPP functions from almost all programming languages and compilers.

For floating point data types, Intel MKL accompanies the set of Intel IPP functions and allows exploiting the full potential of Intel Xeon and Intel Core processors. Intel MKL is available as a single or multithreaded implementation.

Intel MKL includes sophisticated optimizations along with threading. Massive workloads up to the 64-bit integer index space are supported (“ip64”). Intel MKL focuses on high throughput; however, application-level threading can be served by sequential implementations similar to Intel IPP. Note that calling multithreaded Intel MKL functions from multiple threads is thread-safe.

```c
int main(int argc, char* argv[]) {
    ippInit();
    // Start using Intel IPP!
}
```

**Figure 13.** Intel IPP requires calling ippInit prior to any other function of Intel IPP in order to pick the best available code path. A dynamic link library (*.so, *.dll) allows for an initialization step by capturing the "load event" of the library. In contrast, static linkage takes code (according to a call) out of a static library (*.a, *.lib). However, in order to be flexible, ippInit should be called regardless of using dynamic or static linkage.

```c
void process_signal_ipp(size_t size, const float xin[], const float yin[], float xout[], float yout[]) {
    std::copy(xin, xin + size, xout);
    ippsAutoCorr_32f(yin, static_cast<int>(size), yout, static_cast<int>(size));
}

void process_signal_ipp(size_t size, const double xin[], const double yin[],
                        double xout[], double yout[]) {
    std::copy(xin, xin + size, xout);
    ippsAutoCorr_64f(yin, static_cast<int>(size), yout, static_cast<int>(size));
}
```

**Figure 14.** Intel® IPP is a C-function library that easily binds to various programming languages. The C++ code above overloads process_signal_ipp in order to perform auto-correlation in a type-agnostic manner.

### Conclusion

Intel System Studio is a new integrated software development suite that provides deep hardware and software insights to speed development, testing, and optimization of Intel®-based embedded devices and intelligent systems running embedded Linux, including Wind River Linux and Yocto Project.
**Linear Algebra**

- BLAS, Sparse BLAS
- LAPACK solvers
- Sparse Solvers (DSS, PARADISO)
- Iterative solver (RCI)
- ScalAPACK, PBLAS

**Fast Fourier Transforms**

- Multidimensional
- FFTW interfaces
- Cluster FFT
- Trig. Transforms
- Poisson solver
- Convolution via VSL

**Vector Math**

- Trigonometric
- Hyperbolic
- Exponential, Logarithmic
- Power / Root

**Random Number Generators**

- Congruential
- Wichmann-Hill
- Mersenne Twister
- Sobol
- Neiderreiter
- Non-deterministic

**Summary Statistics**

- Kurtosis
- Variation coefficient
- Quantiles
- Order statistics
- Min/max
- Variance-covariance

**Data Fitting**

- Spline-based
- Interpolation
- Cell search

---

**Figure 15.** Linear algebra is one of the main domains of Intel® MKL. The FFTs in Intel MKL support’s sophisticated descriptors (hence forward/backward functions have a slightly simpler interface compared to Intel® IPP).

```cpp
class process_signal_mkl(size_t size, const float xin[], const float yin[],
                        float xout[], float yout[])
{
    static MKL_INT mkl_size = static_cast<MKL_INT>(size);
    static VSLCorrTaskPtr task = 0;

    if (mkl_size != size || 0 == task) {
        vslCorrDeleteTask(&task);
        mkl_size = static_cast<MKL_INT>(size);
        vslsCorrNewTask1D(&task, VSL_CORR_MODE_AUTO, mkl_size, mkl_size, mkl_size);
    }

    std::copy(xin, xin + size, xout);
    vslsCorrExec1D(task, yin, 1, yin, 1, yout, 1);
}
```

**Figure 16.** The interface for correlation (and convolution) is unlike the interface in Intel® IPP. A “flattened object-oriented interface” allows creating and editing a handle (“correlation task”).

---

**Learn More**

**Buy or Renew:** [http://software.intel.com/en-us/articles/buy-or-renew/](http://software.intel.com/en-us/articles/buy-or-renew/)


**Support:** All product updates and Intel® Premier Support services are included for one year. Intel Premier Support gives you secure, web-based, engineer-to-engineer support. Learn more about this tool: [http://software.intel.com/en-us/articles/intel-system-studio-articles](http://software.intel.com/en-us/articles/intel-system-studio-articles)


**Content Library:** Provides access to various technical information

---

1. Measured using an instrumentation example on an Intel® Atom™ Processor E660 running at 1.6GHz and Intel® Core™ i7
2. The remote system explorer (RSE) is part of the Eclipse* Target Management Project. For details on how to use it for embedded cross-development, please refer to the following tutorial at eclipse.org: [http://www.eclipse.org/rtutorial/index.php](http://www.eclipse.org/rtutorial/index.php)
3. There are also dynamic link libraries available that provide internal multithreading.
CloverLeaf Meets the Intel® Xeon Phi™ Coprocessor

By Victor Gamayunov and Stephen Blair-Chappell
Intel's Victor Gamayunov and Stephen Blair-Chappell recently spent some time with the author of the CloverLeaf benchmark helping to make it "Intel® Xeon Phi™ coprocessor-ready."

In this article, Victor and Stephen discuss the steps they took and the results achieved.
What is CloverLeaf?
CloverLeaf is a small benchmark that solves two-dimensional, compressible Euler equations on a staggered Cartesian grid. The code consists of a Fortran framework and a number of Fortran and ANSI-C low level building blocks that do all the real work.

The techniques used in CloverLeaf are similar to what might be used to simulate the effect of a meteoroid impacting the earth's atmosphere—where it is useful to predict how far any resultant shockwave would travel, along with the bulk flow of material behind the shockwave.

Figure 1 shows four snapshots of a CloverLeaf simulation. The results have been visualized using the U.S. Department of Energy graphical analysis tool VisIt. At the start of the simulation, a region of high-energy, high-density gas is introduced into a region of low-energy, low-density gas.

As the simulation progresses, you can see how the shockwave moves. The last picture shows the shockwave reflecting off the boundary—the walls of the square box—and producing a more complex pattern.

What steps did you take to become “Intel Xeon Phi coprocessor ready”?
At first, we had no access to any Intel® Many Integrated Core (Intel® MIC) architecture hardware, so all our development was carried out on a regular eight-core workstation PC. We knew that to make best use of the Intel® Xeon Phi™ coprocessor, our code should be both well parallelized and well vectorized. To perform well, CloverLeaf would need to take advantage of all the coprocessor cores and the 512-bit wide SIMD registers.

The benchmark was already parallelized using MPI, so we put most of our early efforts into vectorizing the code. Our approach was first to find the hotspots in the code, and then to make sure that these hotpots were auto-vectorized by the compiler.

On a regular Intel® Xeon® processor, we found that auto-vectorization improved the performance of the benchmark by 20 percent. On the Intel Xeon Phi coprocessor, vectorization produced a performance boost of 80 percent.

As time went on, we added OpenMP to the MPI code.

Did you find programming for more than 240 threads a challenge?
No, from a programming point of view, it was very easy. In CloverLeaf the simulation space is divided into regions and run using MPI on multiple cores. Figure 2 shows the simulation mesh divided into 16 regions. For the Intel Xeon Phi coprocessor, we simply increased the number of regions to 60, running 60 MPI tasks, with each task using four OpenMP threads.
Is there anything that went particularly well in the project?

We were surprised at how easy it was to get the first version of CloverLeaf running on the Intel Xeon Phi co-processor. We simply recompiled the existing code using the -mmic compiler option, and ran the executable natively on the co-processor.

What was the most difficult hurdle, and how did you overcome it?

At the start of the project, the code was written without much consideration about how well it would vectorize. By using the reporting features of the Intel® compiler, we were able to get reports that told us why some code did not automatically vectorize.

Because of multiple vectorization opportunities within one line of code, we sometimes found that looking at the compiler messages alone was not enough to verify whether code had been vectorized. In these cases, we resorted to looking at the assembly code and searching for the existence of packed instructions.

Based on your experience on this project, what advice would you give to anyone porting a program to an Intel Xeon Phi co-processor?

1. Don’t wait for access to an Intel Xeon Phi co-processor-based system; start your optimization work now!
   Any optimization work you do on a traditional PC or workstation will nearly always result in that same code running better on an Intel Xeon Phi co-processor. In fact, we found that the optimizing work we did had a positive effect even when we ran the code on non-Intel devices.

2. Make sure your code is both highly parallel and well vectorized.
   We found that once our code was parallel and vectorized it ran really well on the Intel Xeon Phi co-processor.

3. Find some means of measuring the parallelism of your program.
   We used Intel® VTune™ Amplifier to confirm how concurrent the code was, and then a combination of the compiler vectorization reports and Intel VTune to see how well the code was vectorized.

4. Refactor your algorithms so they are parallel friendly.
   Getting rid of loop dependencies was especially important in CloverLeaf.

5. Reorganize your data access so it is vector and cache friendly.
   Minimize reading and writing from disjointed memory locations. We found modifying our code so it writes to consecutive addresses made a huge difference to the performance.

6. Where possible, avoid synchronization.
   Don’t use synchronization primitives inside loops, as this will serialize your code. Consider reducing the scope of variables. If possible, avoid the use of shared variables.

7. Avoid dynamic memory allocations.
   Try to avoid using traditional memory allocation functions and other system libraries in your threaded code. Most of these functions will have locks that will introduce serialization. In CloverLeaf, we replaced fine-grained dynamic memory allocation by doing a one-off memory allocation at the start of the program.

What results did you achieve?

Figure 3 shows the results of running CloverLeaf on a single-socket, eight-core CPU and an Intel Xeon Phi co-processor. When the mesh size is small, the co-processor can only manage one-third the performance of the single processor. As the mesh size increases, the Intel Xeon Phi co-processor delivers more than three times the performance than the single processor.

Figure 3. The CloverLeaf results

References

2. https://wci.llnl.gov/codes/visit/home.html

**Single-socket eight-core Intel® Xeon® E5-2687W processor, 3.1GHz, 32GB DDR3 (1333Mhz) memory, with both turbo boost and hyperthreading enabled.**

**Intel® Xeon Phi™ coprocessor with 61-cores, running at 1.09GHz, with 8GB of GDDR5 (5.5 GT/s) memory.**
The Intel® Xeon Phi™ coprocessor can be used to handle offloaded, highly parallel code running on the Intel® Xeon® processor. However, the Intel Xeon Phi coprocessor can also be considered an independent compute node. This means that a distributed memory programming model using the message passing interface (MPI)² is a natural approach for this platform, too.

Starting with version 4.1, the Intel® MPI Library for Linux® supports both Intel Xeon processor and Intel Xeon Phi coprocessor platforms. It provides an easy way to port existing high performance computing (HPC) applications to Intel Xeon Phi coprocessor-equipped platforms.

In this article, we demonstrate how new programming models enabled by the Intel MPI Library, combined with the simplicity of the user interface, facilitate this process. (Look for specific, real-life application-porting efforts in upcoming articles.)

Intel® MIC Architecture as Another MPI Platform

Based on Intel® Many Integrated Core (Intel® MIC) architecture, the Intel Xeon Phi coprocessor has been introduced as a complement to the Intel® Xeon® processor family to enable dramatic performance boosts of highly parallel HPC applications. The Intel Xeon Phi coprocessor features 61 computational cores equipped with a 512-bit vector unit. All the cores have fully coherent 512KB L2 cache, support four threads, run up to 1.05 GHz, and share up to 8GB of GDDR5—providing the memory for the Intel® Manycore Platform Software Stack (MPSS), user processes, and file system storage.

The software stack includes an embedded Linux kernel and provides full set of services, including:

- IP addressing
- Ethernet bridging
- Complete Secure Shell® (SSH) access
- InfiniBand Host Channel Adapter® (HCA) access thru CCL-direct (Coprocessor Communication Layer)
- Network File System® (NFS)

All these features provide high-level standard interfaces that allow us to treat the Intel Xeon Phi coprocessor as an independent network node. Thus, MPI allows for a very natural approach to Intel Xeon Phi coprocessor programming, expanding the breadth of programming models supported by this architecture.

Programming Models

There are two high-level programming models applicable to Intel MIC, namely, the offload and the MPI models (see Figure 1).

In the offload mode, either the Intel Xeon Phi coprocessor or the Intel Xeon processor is considered an accelerator. Common to both offload models is the use of the offload capabilities of the products like the Intel® C Compiler, Intel® C++ Compiler, and Intel® Math Kernel Library (Intel® MKL). The key feature of this mode is that an MPI call cannot be placed inside the offloaded part of the code because MPI library “knows nothing” about offload features of the application.

<table>
<thead>
<tr>
<th>Offload</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® MIC Architecture or host CPU as an accelerator</td>
<td>MPI ranks on several coprocessors and/or host nodes</td>
</tr>
<tr>
<td>Host CPU as an accelerator</td>
<td>Messages to/from any core</td>
</tr>
</tbody>
</table>

**MIC Offload (direct acceleration)**
- MPI ranks on the host CPU only
- Messages into/out of the host CPU
- Intel® MIC Architecture as an accelerator

**Host Offload (reverse acceleration)**
- MPI ranks on the MIC CPU only
- Messages into/out of the MIC CPU
- Host CPU as an accelerator

**Symmetric**
- MPI ranks on the MIC and host CPUs
- Messages into/out of the MIC and host CPUs
- Threading possible

**Native**
- MPI ranks on the MIC CPU only
- Messages into/out of the MIC c/o host CPU
- Threading possible

*Figure 1. Programming models*
The direct acceleration model is already supported by earlier versions of Intel MPI, as well as the other MPI implementations. The reverse acceleration model is supported by Intel® MPI, version 4.1, depending on the respective offload capabilities in the aforementioned related products. In the MPI mode, the host CPUs (typically, the Intel Xeon processor) and Intel Xeon Phi coprocessors are considered as peer nodes. Here, the MPI processes may reside on both or either host CPUs and Intel Xeon Phi coprocessors in any combination. There are two major use cases:

- **Symmetric model:** MPI processes reside on both the host and the Intel Xeon Phi coprocessor. This is the most general MPI view of an essentially heterogeneous cluster.
- **Native model:** All MPI processes reside only on the Intel Xeon Phi coprocessor. This is a specific case of the symmetric model. Also, this model has a certain affinity to the reverse acceleration model, because the host CPUs may, in principle, be used for offload.

### User Interface

For the Intel Xeon Phi coprocessor, Intel MPI provides the Hydra* mpiexec.hydra process manager (PM) and the convenience script mpiexec.hydra built on top of the Hydra. They are provided for the Intel Xeon processor as well, so this PM is intended to be common for all aforementioned programming models.

Each node and each coprocessor are identified using their unique symbolic or IP addresses. This is a fundamental requirement that all Intel MPI support for Intel MIC is built upon. The same is true of related services provided by third-party products, such as job management, resource control, cluster monitoring, and so on.

The intended hosts and coprocessors can be specified using their symbolic names or IP addresses in the command line of the mpiexec.hydra through the long (i.e., colon) notation, for example:

```
(host)$ mpiexec.hydra -n 64 --hostfile myhosts ./hello
```

To use the more convenient short notation, create a hostfile or a machinefile where the intended nodes and coprocessors are specified by their symbolic names or IP addresses. We will use this method below in application to the test program hello.c provided with Intel MPI.

The building and launching process for the offload model looks exactly the same as for an Intel Xeon processor cluster, namely:

1. Set up the Intel Composer XE and Intel MPI environments.
   ```
   (host)$ source <composer_install-dir>/bin/compilervars.sh intel64
   (host)$ <installdir>/intel64/bin/mpivars.sh
   ```
2. Compile an application, for example:
   ```
   (host)$ mpiicc --o hello hello.c
   ```
3. Launch the executable file, for example using a hostfile to define the intended nodes:
   ```
   (host)$ mpirun -n 64 --hostfile myhosts ./hello
   ```
   Where myhosts lists the names of the intended nodes:
   ```
   host1
   host2
   ```

To build an application in the native mode, set up the Intel Composer XE and Intel MPI environment on the host as indicated above. Then, add the -mmic option to the Intel® compiler invocation string to build an executable binary for the Intel MIC architecture.

```
(host)$ mpiicc -mmic -o hello.mic hello.c
```

To run the application on the coprocessors, set the environment variable I_MPI_MIC to 1, and put the coprocessors’ names or IP addresses into the hostfile.

```
(host)$ mpirun -n 64 --hostfile <hostfile> ./hello.mic
```

The coprocessors pointed out in the hostfile can be attached to one or several host CPUs on one or different cluster nodes. MPI rank distribution over the coprocessors following the same rules as in a regular, host-only MPI application run.

Intel MPI Library normally targets so-called “thin” binary executable files. For example, all binary executable files are specifically targeted to either the host CPU or the Intel MIC architecture at the time of compilation. Due to the incompatibility of the respective instruction sets and machine types of binary executables, a host-targeted binary executable file cannot run on Intel MIC architecture, and vice versa. This is why, in the Symmetric model, you’ll want to build the application twice: once for the Intel Xeon processor, and once for the Intel Xeon Phi coprocessor platform. For example:

```
(host)$ mpiicc -o hello hello.c
(host)$ mpiicc -mmic -o hello.mic hello.c
```
In order to select a proper executable for the platform, Intel MPI uses the following convention:

1. The name of the executable mentioned in the `mpiexec.hydra` command line is assumed to be the name for the host.
2. The name of the Intel MIC executable is produced on the basis of the host executable name by prepending the contents of the `I_MPI_MIC_PREFIX` environment variable, and appending the contents of the `I_MPI_MIC_POSTFIX` environment variable. Both variables are empty by default.

For the example above, set the `I_MPI_MIC_POSTFIX=.mic`. The heterogeneous launch may be performed now in the usual manner:

```
(host)$ mpirun -n 64 -hostfile <hostfile> ./hello
```

The executable binaries `hello` and `hello.mic` will be launched on Intel Xeon processor or Intel Xeon Phi coprocessor nodes as defined by the `hostfile`.

**Conclusion**

The key design feature when using Intel MPI for the Intel Xeon Phi coprocessor is “same as for the Intel Xeon processor.” Intel MPI provides two programming models to enable the Intel Xeon Phi coprocessor for existing MPI applications. Porting these applications can require some contemplation and effort.

Our next step is to examine these programming models in greater depth, using real-life applications ported to Intel Xeon Phi coprocessor systems to highlight the relative benefits of the proposed approaches and provide practical recommendations as to the choice of the programming model and the most appropriate porting methodologies. Look for these upcoming articles in future issues of the magazine.

**References**

From Legacy Serialism to Parallelism:
Converting a Real-World Application Using Intel® Cilk™ Plus
by Chris Chrulski
This article walks through a high-level workflow and the results achieved when converting a real-world, legacy serial application into a parallel application using Intel® Cilk™ Plus technology. The workflow uses Intel® VTune™ Amplifier XE to analyze the original serial application for hotspots; applies a parallel design pattern to convert the serial algorithm into a parallelizable algorithm; implements the parallelism using Intel Cilk Plus functionality; and analyzes the correctness and performance of the modified application using Intel® Inspector XE 2013 and Intel VTune Amplifier XE. Using a combination of serial optimizations and parallelism from Intel Cilk Plus, a 25X improvement in runtime was achieved on an important workload for the application when running on an 8-core system without a significant recoding effort of the legacy serial application.
Application Background

The application optimized was the profmerge utility which is a component within the Intel® Composer Xe product. Profmerge is used for merging the intermediate data files produced by the Profile Guided Optimization (PGO) instrumentation. These files are merged into a form that can be efficiently used to enable higher levels of optimizations during the profile-guided feedback compilation step, as well as for generating reports with the Intel® Code Coverage tool.

When applications are built with PGO instrumentation, each run of the instrumented application produces one or more dynamic data result (.dyn) files, which contain information about the frequency of code paths and frequent values. Within each data file, a data record exists for each instrumented routine executed within the application. Generally, an application will be run multiple times with different data sets to cover different code paths for optimization or to analyze the test coverage.

The profmerge application was originally written in C in the mid-90’s, and has been extended over the years to support additional functionality. The basic application structure uses top-level global data structures and variables to hold the current state for merge; counters for all of the routines executed; frequencies of branch paths; common targets for indirect function calls; and statistics about the merge process itself.

Selecting a Workload

To prepare for optimizing the application, a representative workload was chosen consisting of over 1,000 .dyn files produced from training runs of the Intel Composer Xe product. This workload was selected for its importance to the product and reproducible behavior.

Analyzing the Serial Application

Profmerge was taking approximately 10 minutes to process the workload on a dual-core Intel® Xeon® X5365 system, running @ 3.0 GHz with 16 GB of memory under Windows 7 Enterprise. Intel® VTune™ Amplifier XE 2013 was used to identify hotspots in the application using a release build configuration with debug information enabled. A subset of 25 .dyn files was used to limit the amount of time and data collected to a manageable amount.

Analysis of the results (Figure 1) showed two hot functions that were extremely hot, as seen by blue bars representing the percentage of application time spent directly within those routines. Normally, these hot routines could be considered candidates for speedups from parallelization. However, in this case, the routines were involved with the loading of the data files, so these were not areas that could be readily parallelized. Instead, some initial optimization of the serial code was done to improve these routines to make sure that we had a suitable program to parallelize; otherwise these routines could limit the performance of the parallel version. After optimizing the serial routines, hotspot analysis was repeated. This did not reveal a need for more optimization of the serial version.

The next step was to investigate methods to use parallelism to simultaneously take advantage of the multiple cores.

Additional information and tutorials on using Intel VTune Amplifier XE for analyzing applications can be found in "Parallel Programming with Intel® Parallel Studio Xe," by Stephen Blair-Chappell and Andrew Stokes.

![Figure 1. Serial bottom-up hotspots analysis](image-url)
Applying a Design for Parallelization

The first step in converting our legacy serial application to a parallel application was identifying where to introduce parallelism to have good granularity for splitting the work, while minimizing the communication overhead between the workers.

Examining the Top-Down Tree within Intel VTune Amplifier XE showed that the majority of the time was spent within three main portions of the application: reading and merging the .dyn files; post-processing of the data; and writing the results. In Figure 2 the blue bars represent the amount of time taken from the time a routine starts until it completes. The first two tasks under the routine Process_Files were chosen to be parallelized, as they were consuming 80 percent of the runtime.

In the original implementation, files were processed sequentially—loading records from the beginning to the end of the file—performing some initial processing on the records to prepare for updating the aggregated data. The time spent per record is small and, because of the sequential traversal through the file, there was not much opportunity to get scalable results from multiple workers within a single file. Instead, using separate worker threads operating on different files was needed.

An examination of common design patterns used for parallel applications was completed, and the Fork-Join design pattern described in chapter 8 of “Structured Parallel Programming,” by Michael McCool, et al., was chosen. This pattern can be implemented using a “divide and conquer” algorithm for processing the set of files by splitting the set in half, and then performing a merge on each half to produce a partial result. This is applied recursively, until the set is small enough to be more efficiently processed serially. Upon completion of the sub-sets, the partial results are merged with one another as the recursive calls complete.

But, the use of the global variables for the merge state presented challenges for directly converting the existing application into a parallel application. Each worker thread could modify the global state by adding new records or updating the state of an existing record. Locks could be used to enable thread safety, but would limit performance gains. Modifications were made to encapsulate the global variables within a data structure to allow separate threads to have their own data structure and copy of the variables. This encapsulating object would contain a set of records containing all the data for a set of files.

The modifications to encapsulate the state into an object required some effort. It was necessary to go through all the locations that accessed or modified the global variables to have them operate on a state object. Also, new code needed to be written to support the merging together of separate object instances. However, the majority of the application logic and code remained largely untouched during these changes. In some applications, Intel Cilk Plus hyper-objects could be used instead of implementing manual encapsulation.
// Merge a subset of .dyn files. After this function completes the ‘merger’
// object will have processed the files of ‘node_vect’ in the range
// [lower_bound..upper_bound). This routine may recursively call itself with
// non-overlapping subsets of the range, until some minimum range is reached.
//
// Params:
// node_vect – array of files to be merged
// lower_bound – index of first file element to process
// upper_bound – index one past the last element to be merged.
// merger – the data structure to store the results of the merge into.
static void merge_nodes(
    FileNodeVect       *node_vect,
    int                 lower_bound,
    int                 upper_bound,
    DpiMerger          *merger)
{
    // If the range to be processed is small enough, just process all the
    // files serially rather than splitting it for other threads.
    if (upper_bound - lower_bound <= file_count_threshold) {
        for (int i = lower_bound; i < upper_bound; i++) {
            merger->merge_file(*node_vect[i]);
        }
    }
    else {
        // Split the range in half, and process each half independently.
        int mid_point = lower_bound + (upper_bound - lower_bound) / 2;

        // Create a new merge object for the upper half of the files so that
        // it can be operated upon concurrently with the incoming merger
        // object via the Intel® Cilk™ Plus continuation thread.
        DpiMerger *high_merge = pgopti_create_dpi_merger();

        // merge [lower_bound..mid_point)
        cilk_spawn merge_nodes(node_vect, lower_bound, mid_point, merger);

        // merge [mid_point..upper_bound)
        merge_nodes(node_vect, mid_point, upper_bound, high_merge);
        // Explicitly sync here so that results from both halves are complete
        // and can be merged into the callers DpiMerger object.
        cilk_sync;
        merge_mergers(merger, high_merge);
        pgopti_delete_dpi_merger(high_merge);
    }
}

Figure 3. Structure of cilk_spawn code

// Process each record to populate the execution counts for each of the
// blocks based on the edge counts.
void compute_counts_from_edges(DpiNodeVec *nodes, size_t count)
{
    size_t i;
    cilk_for (i = 0; i < count; i++) {
        dpi_node_compute_counts_from_edges(nodes[i]);
    }
}

Figure 4. Structure of cilk_for code
Implementing the Parallelism with Intel Cilk Plus

Figure 3 shows the implementation of the divide and conquer algorithm. The routine, `merge_nodes`, requires an array of files to be processed, the indices for the range of elements to process as inputs, and a pointer to the data structure that was to be updated with the results. A check is performed to determine whether to process the range as a single set, or to split the range into subsets. If the range is to be split, the midpoint of the range is found, and two recursive function calls are made. In order for these two calls to be processed independently when using Intel Cilk Plus, a new instance of the encapsulated state object is constructed to be used within one of these calls.

After the modified algorithm was implemented and tested, Intel Cilk Plus keywords were added to the application to enable the parallelism as shown in blue in Figure 3. With just a single `cilk_spawn` directive to the first recursive call and a `cilk_sync` directive following the second recursive call, the processing was able to be performed in parallel. The `cilk_sync` was necessary, because the application needs both ranges to be complete prior to combining them.

For the second portion of the application that was chosen for a parallel implementation, processing takes place after all the data is loaded on each record independent of all other records. A simple ‘for’ loop that walked over all the elements of the record array was converted into a parallel loop, with the introduction of the `cilk_for` keyword in the loop control statement, as shown in Figure 4.

With just these changes, and the inclusion of the `cilk.h` header file, the conversion of the legacy application into a parallel application was implemented and ready to be analyzed for errors and performance. The changes to get to this point were far simpler than manually creating threads, and managing the sharing of data among threads.

Checking for Threading Errors

Intel Inspector XE 2013 was used to check for potential problems with the parallel application using a debug build configuration. Threading error analysis using the highest level of analysis, “Locate Deadlocks and Data Races,” was run using 25 .dyn files from the workload as shown in Figure 5. When using Intel Inspector XE, we recommend using a smaller workload, due to the amount of data being collected and the overhead of the data collection.
Because the original implementation in C relied on global variables, this threading error analysis is useful for locating problems where separate threads may be modifying a common global variable. Figure 6 shows a sample of results produced by the analysis of possible data race conditions. The top pane lists a summary of the potential problem areas, and the lower pane provides more detail about the location and threads involved in the data race. In the case shown, there was a global variable being used by a routine within the parallel region, and two different threads were setting the value of this variable. Utilizing this report over several runs, code changes were made to resolve the data races.

Figure 6. Data race report

Figure 7. Hotspots by CPU usage
Performance and Scalability Analysis

To check the efficiency of processing core usage, another hotspot analysis was performed using Intel VTune Amplifier XE 2013 on the original 1,000+ input file data set.

In the summary results window of Intel VTune Amplifier XE, selecting the “Hotspots by CPU Usage” pull-down option provides information about how effectively CPU cores were used. The histogram shows the amount of time the cores were actively running the application. In Figure 7, it can be seen that, as desired, all eight cores are fully utilized the majority of the time by the application.

The application was checked for scalability by varying the number of worker threads. Data was collected for the time spent by the three main activities of the application as shown in Figure 8. The blue region indicates the time for the divide and conquer algorithm that used cilk_spawn. The red region indicates the loop that was converted to a cilk_for loop. The green region indicates the remaining serial processing. Ideally, we would like to see the performance improve as more cores get used, as in Figure 8.

However, as more cores were added, the performance gains of the parallelism diminishes once eight threads are active for this application. Experiments on a machine with more cores was done, which confirmed that without further code changes, adding more cores did not meaningfully continue scaling on this code. Investigation of the scalability limitations found they were due to the extra work needed to combine partial results by the chosen algorithm, as well as from memory getting dynamically allocated and released as each record was processed. In the conversion from a serial application to a parallel application, the calls to the C runtime for malloc and free were not changed. The sharing of a single heap for the dynamic memory requests created a bottleneck, as the system locks the heap to satisfy the memory requests. Source code changes to make use of a scalable memory allocator (such as the one provided by Intel® Threading Building Blocks [Intel® TBB]) that uses separate heaps per thread would be necessary to achieve further improvements.
Conclusion

Converting this legacy serial application into a parallel application was successful without a lot of recoding. The original application took 650 seconds to process the workload on the test system. Using Intel VTune Amplifier XE 2013, hotspots were identified and changes were implemented that reduced the runtime to 85 seconds. It took a little effort to restructure the code for encapsulating the global state into independent objects, but did not require the majority of the application’s code to be modified to make use of parallelism. Using Intel Cilk Plus, the runtime was reduced to 26 seconds on the selected workload due to parallelism. With Intel Cilk Plus, the changes needed to enable the application to take advantage of multiple CPU cores were simple—and far easier than attempting to manually create threads and coordinate data sharing and communication—and yielded a significant performance improvement for a legacy application.

References

1. Intel® C++ Compiler XE 13.0 User and Reference Guides, document number 323272-130US.

“Using Intel® Parallel Studio tools with Intel® Cilk™ Plus, a 25X improvement in runtime was achieved on an important workload—without a significant recoding effort of the legacy serial application.”
RESOURCES AND SITES OF INTEREST

Go Parallel

The mission of Go Parallel is to assist developers in their efforts toward “Translating Multicore Power into Application Performance.” Robust and full of helpful information, the site is a valuable clearinghouse of multicore-related blogs, news, videos, feature stories, and other useful resources.

“What If” Experimental Software

What if you could experiment with Intel’s advanced research and technology implementations that are still under development? And then what if your feedback helped influence a future product? It’s possible here. Test drive emerging tools, collaborate with peers, and share your thoughts via the “What If?” blogs and support forums.

Intel Developer Zone

Check out a range of resources on a wide variety of software topics for a multitude of developer communities ranging from manageability to parallel programming to virtualization and visual computing. This content-rich collection includes Intel® Software Network TV, popular blogs, videos, tools, and downloads.

Step Inside the Latest Software

See these products in use, with video overviews that provide an inside look into the latest Intel® software. You can see software features firsthand, such as memory check, thread check, hotspot analysis, locks and waits analysis, and more.

Intel® Inspector XE
Intel® VTune™ Amplifier XE

Intel® Software Evaluation Center

The Intel® Software Evaluation Center makes 30-day evaluation versions of Intel® Software Development Products available for free download. For high performance computing products, you can get free support during the evaluation period by creating an Intel® Premier Support account after requesting the evaluation license, or via Intel® Software Network Forums. For evaluating Intel® Parallel Studio, you can access free support through Intel Software Network Forums ONLY.
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
NEW RESOURCES MAY CHANGE THE WAY YOU BUILD APPLICATIONS.

Explore the latest on parallel programming from threading to HPC.

Structured Parallel Programming
by Michael McCool, Arch Robison, and James Reinders
Learn more at: www.parallelbook.com

“I’ve been dreaming for a while of a modern, accessible book that I could recommend to my threading-deprived colleagues and assorted enquirers to get them up to speed with the core concepts of multithreading, as well as something that covers all the major current interesting implementations. Finally I have that book.”

Martin Watt, Principal Engineer, DreamWorks Animation

Intel® Xeon Phi™ Coprocessor High Performance Programming
by Jim Jeffers and James Reinders
Learn more at: www.lotsofcores.com

“This book belongs on the bookshelf of every HPC professional. It takes us back to the universal fundamentals of high performance computing, including how to think and reason about the performance of algorithms mapped to modern architectures, and it puts into your hands powerful tools that will be useful for years to come.”

Robert J. Harrison, Institute for Advanced Computational Science, Stony Brook University
Accelerate embedded and mobile system development.

Deep system-level insights for optimal power, performance, and reliability.

Intel® System Studio includes tools for compiling, JTAG and application debugging, memory and thread checking, an advanced System-on-Chip (SoC) power and performance profiler, and more. Bring innovative products to market, faster with this advanced tool suite from Intel.

LEARN MORE

DOWNLOAD TRIAL SOFTWARE