

White Paper
Stephen Blair-Chappell
Technical Consulting
Engineer
Intel Corporation

7 Tips To Help Getting Started With Multicore

January 2009



Executive Summary

Multicore processors are everywhere. In desktop computing, it is almost impossible to buy a computer that does not have a multicore CPU inside. Multicore technology is also impacting the embedded space, where increased performance per watt presents a compelling case for migration.

Multicore technology is also impacting the embedded space, where increased performance per watt presents a compelling case for migration

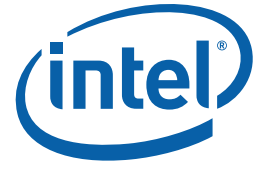
Developers are increasingly turning to multicore because they either want to improve the processing power of their product, or they want to take advantage of some other technology that is 'bundled' within with the multicore package.

This article offers seven tips to help making that first step to using such devices.



Contents

Tip 1. Don't Fix What Works	4
Tip 2. Use Multicore to Take Advantage of the Low Thermal Footprint.....	4
Tip 3. Utilize the Advanced Architectural Extensions.....	5
Auto-vectorization and SIMD	5
Tip 4. Know Your Program Before You Parallelize	6
Tip 5. Consider the Kind of Parallelism You Want to Introduce	8
Virtualization	9
Thread-based Parallelization	10
Tip 6. Implement Your Parallelism Using High-level Constructs	10
OpenMP*	11
C++ Language extensions.....	11
Graphical Programming.....	12
Tip 7. Use Tools That Are Capable of Debugging and Tuning Parallel Applications	13



Tip 1. Don't Fix What Works

It's not unnatural to want to use the latest technology in our favorite embedded design. It is tempting to make a design to be a technological showcase, using all the latest bells and whistles. However, it is worth reminding ourselves that what is fashion today will be 'old hat' within a relatively short period.

If you have an application that works just fine, and is likely to keep performing adequately within the lifetime of the product, then maybe there is no point in upgrading.

Tip 2. Use Multicore to Take Advantage of the Low Thermal Footprint

One of the benefits of the recent processor design trends has been the focus on power efficiency. Prior to the introduction of multicore, new performance barriers were reached by providing silicon that could run with faster and faster clock speeds. An unfortunate by-product of this speed race was that the heat dissipated from such devices made them unsuitable for many embedded applications.

As clock speeds increased, the limits of the transistor technology physics were moving ever closer. Researchers looked for new ways to increase performance without further increasing power consumption. It was discovered that by turning down the clock speeds and then adding additional cores to a processor, it was possible to get a much improved performance per watt measurement.

The introduction of multicore, along with of new gate technology, redesign of power-hungry parts of CPU designs has led to CPUs that use significantly less power, and yet are capable of more raw processing than their antecedents.

An example is the Intel® Atom™ processor, a low power Intel® architecture processor which uses 45nm Hi-K transistor gates. By implementing an in-order pipeline, adding additional deep sleep states, supporting SIMD (Single Instruction Multiple Data) instructions and using efficient instruction decoding and scheduling, Intel has produced a powerful but not power-hungry piece of silicon.

Taking advantage of the lower power envelope could in itself be a valid reason for using multicore devices in an embedded design – even if the target application is still single-threaded.



Tip 3. Utilize the Advanced Architectural Extensions

All the latest generation CPUs have various architectural extensions that are 'free' and should be taken advantage of. One very effective, but often underused extension is support for SIMD – that is conducting several calculations in one instruction.

The Intel® Atom™ processor, for example, has dedicated SIMD execution units, as can be seen in [Figure 1](#).

Often developers ignore these advanced operations because of the perceived effort of adding such instructions to application code. While it is possible to use these instructions by adding macros, inline assembler or dedicated library functions to the application code, a favorite of many developers is to rely on the compiler to automatically insert such instruction in the generated code.

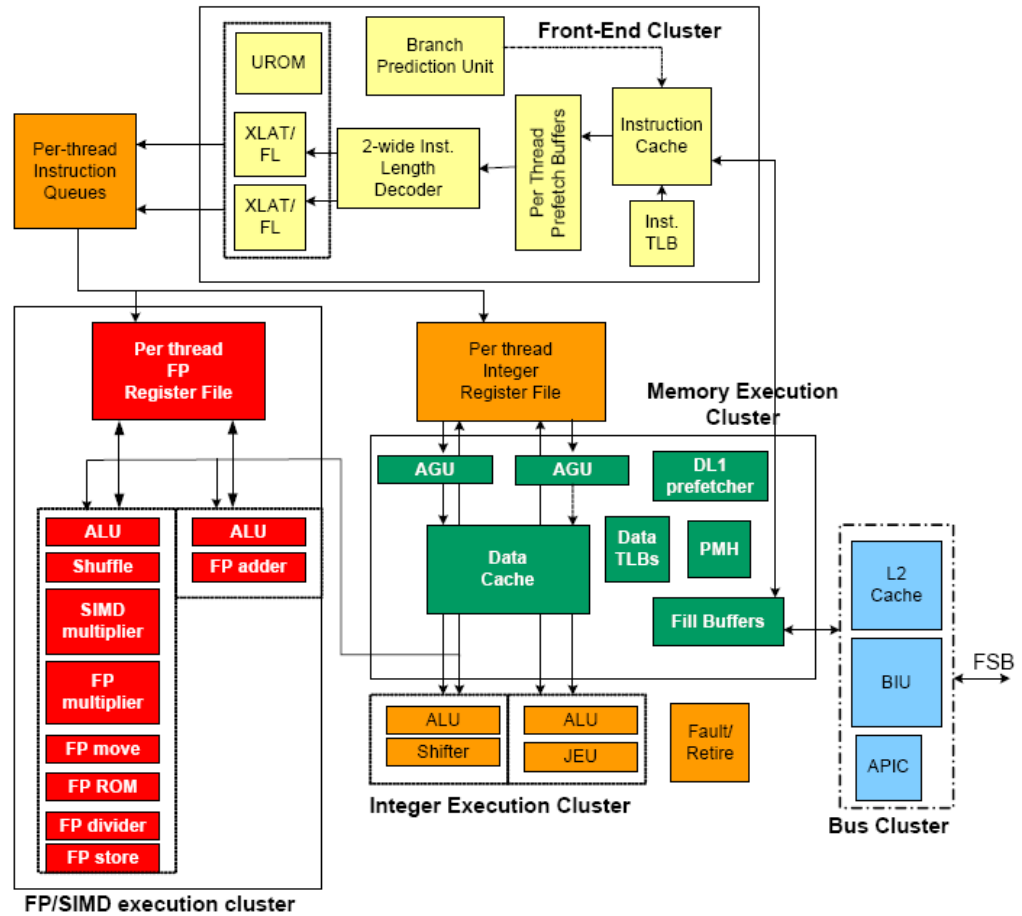
Auto-vectorization and SIMD

One technique known as 'auto-vectorization' can lead to significant performance boost of an application. In this technique the compiler looks for calculations that are performed in a loop. By replacing such calculations with say Streaming SIMD Extension (SSE) instructions, the compiler effectively reduces the number of loop iterations required. Some developers have seen their applications run twice as fast by turning on auto-vectorization in the compiler.

The potential performance boost by this technique cannot be overstated. Some developers have reported speedup of their application by a factor of 2 or more. In one recent case, a speedup of over 10 was achieved – all by the flick of a compiler switch. Intel® architecture yields the best benefit.

Like the power gains discussed previously, using these architectural extensions may be a valid reason in itself for using a multicore processor even if you are not developing threaded code.

Figure 1. The Internals of the Low Power Intel® Architecture



Tip 4. Know Your Program Before You Parallelize

Not all programs are good candidates for parallelism. Even if your program seems to need a 'parallel facelift' it does not necessarily follow that going multicore will help.

For example say your product is an application that runs simulations of weather patterns in real-time based on data collected from a number of remote sensors.

The measurements of wind speed, direction, temperature and humidity used to calculate what the weather pattern will occur over the following 30



minutes. Imagine that the application always produces its calculation results too late, and the longer the application runs the worse the timeliness of the simulation is.

One could assume that the poor performance is because the CPU is not powerful enough to do the calculations in time. Going parallel might be the right solution – but how do we prove this? Of course, it could be that the real bottleneck is an I/O problem, the reason for the poor application performance being the implementation of the remote data collection and not excessive CPU load.

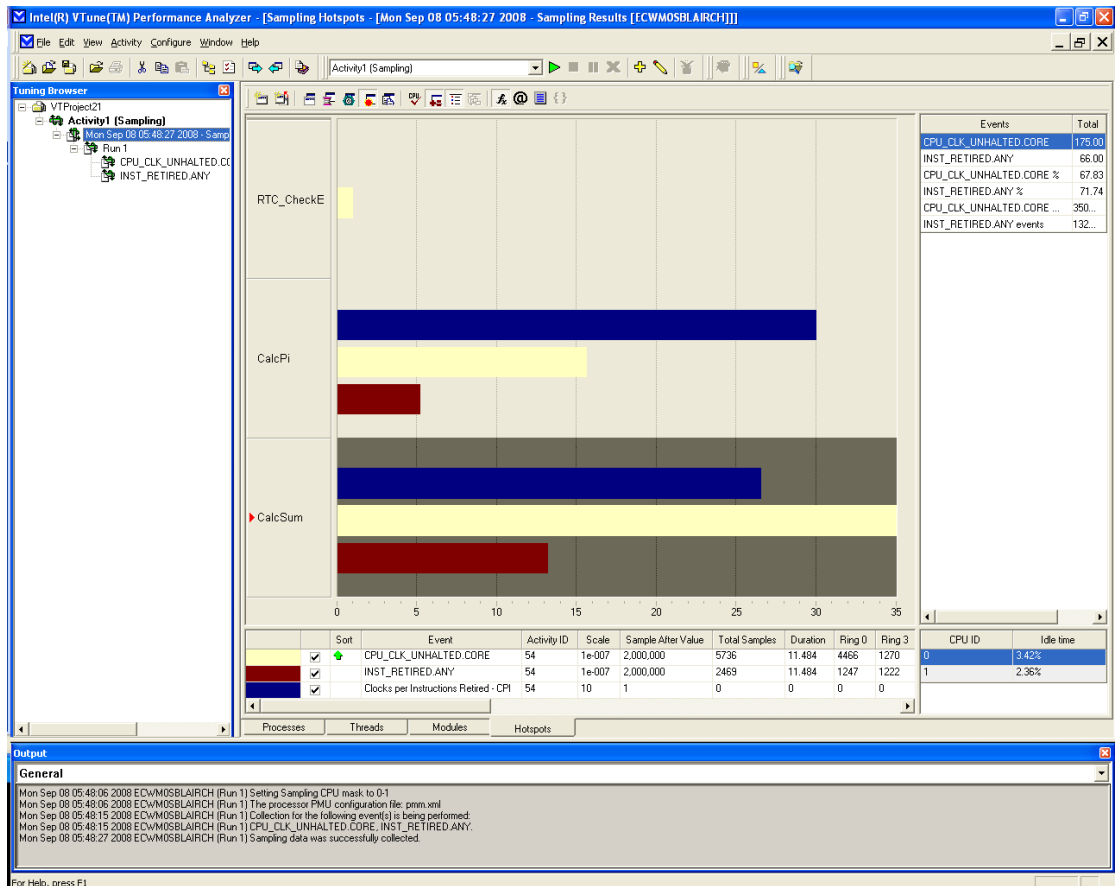
There are a number of profiling tools available that will help you get a correct picture of the running program. The Intel® VTune™ Performance Analyzer is an example of such a tool, and can be seen in [Figure 2](#)

Such analyzers typically rely on runtime architectural events that are generated by the CPU.

Before migrating your application to multicore, it would be worth analyzing the application with such a tool, using the information in the decision making process.



Figure 2. Using the Intel® VTune™ Performance Analyzer to Observe Runtime Behavior



Tip 5. Consider the Kind of Parallelism You Want to Introduce

There are several techniques open to the developer who wants to migrate to multicore. Techniques vary depending on whether you simply want to run multiple instances of the same application, or whether you want to optimize a single application so that it takes advantage of the extra cores. Running multiple applications can be achieved by either relying on the operating system to schedule parallel runs, or by relying on virtualization techniques, where multiple copies of the operating system run on different cores.



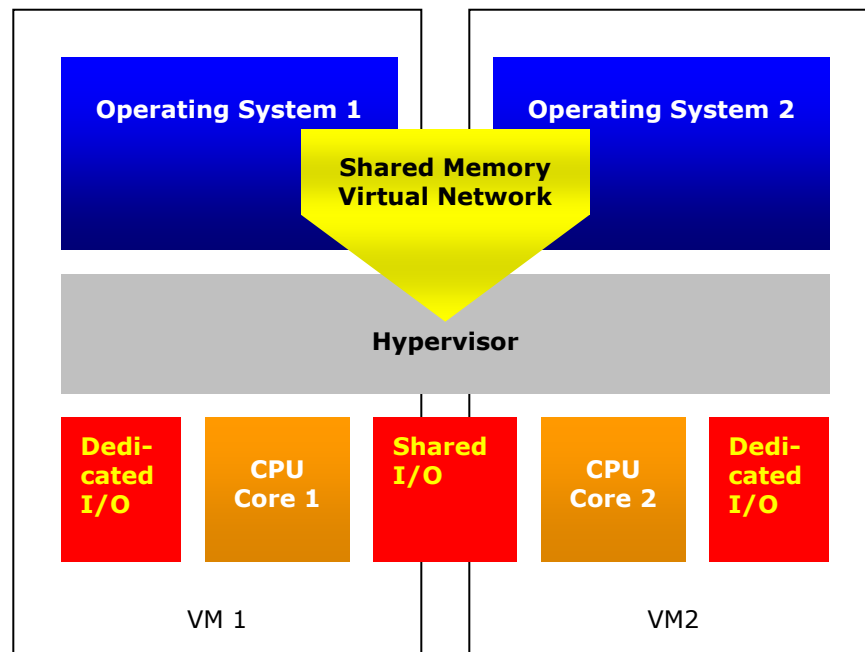
Virtualization

There are a number of schemes that can be called *virtualization*. This paper focuses on any scheme that allows us to run our application and OS on multiple cores in a deterministic manner.

Virtualization, that is, the ability to run more than one OS on a single machine is not new - what is new is the implementation of virtualization support in the CPU. An example is the Intel® Virtualization Technology for IA-32 Intel® architecture in the latest generation of multicore processors.

Most virtualization implementations rely on a Hypervisor or Virtual Machine Manager that interfaces between the CPU and the virtual operating systems – see [Figure 3](#).

Figure 3. An Embedded Virtualization Scheme



In an embedded system it is important that deterministic and timely response can be guaranteed from the underlying hardware. Hypervisors that are suitable for embedded applications will have the facility to assign use of underlying hardware to specific Virtual Machine.

Virtualization can be a quick and effective way to get your embedded application running on all cores on a multicore platform – as long as the results of the independent runs can be used.



For further information on virtualization a good resource is Max Domeika's book "Software Development for embedded Multicore systems" ISBN 978-0-7506-8539-9.

Thread-based Parallelization

One can introduce thread-based parallelism into the high-level design of a program in different ways. Three common strategies available are *functional parallelism*, *data parallelism* and *software pipe-lining*.

In *functional parallelism*, each task or thread is allocated a distinct job, for example one thread might read a temperature transducer, while another thread is carries out a series of CPU intensive calculations.

In *data parallelism*, each task or thread carries out the same type of activity. For example, a large matrix multiplication can be shared between say 4 cores, thus reducing the time taken to perform that calculation by a factor of 4.

A *software pipeline* is akin to a production line, where a series of workers carry out a specific duty before passing the work onto the next worker in the production line. In a multicore environment, each worker or pipeline is assigned to a different core.

In traditional parallel programming, much emphasis is laid on the *scalability* of an application. Good scalability implies that a program running on a dual-core processor would run twice as fast on a quad-core.

In embedded systems computing scalability is less important because the execution of the end product tends not to be changed; the shelf life of the end product usually being measured in years rather than months. When moving to multicore, the embedded engineer should not be over-sensitive to the scalability of the design, but rather use a combination of data and functional parallelism that gets best performance.

Tip 6. Implement Your Parallelism Using High-level Constructs

Threading is not a new discipline, and most operating systems have an API that allows the programmer to create and manage threads. Using the APIs directly in the code is complicated, so the recommendation is to use a higher level abstraction.

One way of implementing threading is to use various high-level constructs or extensions to the programming language.



OpenMP*

OpenMP* is a pragma-based language extension for C\C++ and FORTRAN that allows the programmer to easily introduce parallelism into an existing program. The standard has been adopted by a number of compiler vendors including GNU, Intel, and Microsoft*. A full description of the standard can be found at www.openmp.org

With OpenMP it is easy to incrementally add parallelism to a program. Because the programming is pragma based, code can still be built on compilers that do not support OpenMP – the compiler in this case would just issue a warning that it found an unsupported pragma.

Figure 4a shows a loop based parallelism. The number of loops is split up between the cores. It is important the work carried out within the loop not have iteration dependencies – for example, results in loop n , must not be used in loop $n + 1$.

Figure 4. Examples of Parallelism Using OpenMP

Example 4a

```
int results[100];
.
.

#pragma openmp parallel for
for (i=0; i< 100;i++)
{
    results[i]=i;
}
```

Example 4b

```
#pragma openmp parallel
sections
{
    #pragma openmp section
    myfunc1();

    #pragma openmp section
    myfunc2();
}
```

As stated earlier, functional parallelism is potentially more interesting than data parallelism in embedded application development. Figure 4b is an example of how to achieve functional parallelism using the *sections* directive. Each *section* within the *sections* (notice the extra 's') are run simultaneously on a different core.

C++ Language extensions

An alternative to using OpenMP* is to use one of the newly emerging language extensions which supply a similar functionality. Below are examples of the use of such extensions.



Figure 5: Using Experimental Language Extensions to Implement Parallelism

```
void f_sum ( int length, int *a, int *b, int *c )
{
    int i;
    __par for (i=0; i<length; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

It is expected that eventually such language extensions will be adopted by an appropriate standards committee. An experimental compiler that contains such extensions can be found at <http://software.intel.com/en-us/whatif>.

Graphical Programming

An alternative approach to traditional programming languages is to use a graphical development environment. There are a number of 'program by drawing' development tools that take care of the low level threading implementation for the developer.

An example of this is National Instruments LabVIEW*, which allows the programmer to design his program diagrammatically, by connecting a number of objects together. Support for multicore is simply adding a loop block to the diagram. An example of such a diagram can be seen in [Figure 6](#).



Figure 6: National Instruments LabVIEW Graphical Programming Environment.



Tip 7. Use Tools That Are Capable of Debugging and Tuning Parallel Applications

When programs run in parallel, they can be very difficult to debug – especially if you try using tools that are not enabled for parallelism.

Identifying and debugging issues related to using shared resources and shared variables, synchronization between different threads, and dealing with deadlocks and livelocks are notoriously difficult.

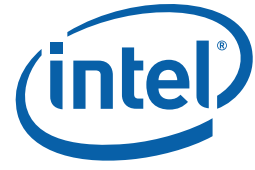
There are a number of tools from different vendors that are specifically designed to aid in the debugging and tuning of parallel applications.

The Intel® Thread Checker and Intel® Thread Profiler are examples of tools that can be used to debug and tune parallel programs.

Where no parallel debugging tools are available for the embedded target you are working on, it is a legitimate practice to use standard desktop tools,



carrying out the first set of tests on the desktop rather than the embedded target. It is a common experience that threading issues appearing on the target can be first captured by running the application code on a desktop machine.



Authors

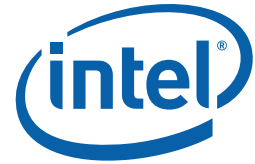
Stephen Blair-Chappell is a Technical Consulting Engineer with the Intel Compiler Labs at Intel Corporation.

Acronyms

CPU Central Processing Unit

SSE Streaming SIMD Extension

SIMD Single Instruction Multiple Data



INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

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

This paper is for informational purposes only. THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel, the Intel logo, Intel. leap ahead. and Intel. Leap ahead. logo, VTune, Intel Atom, Intel Core, are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2008 Intel Corporation. All rights reserved.