



# Intel<sup>®</sup> Technology Journal

Multi-Core Software

**Parallelization Made Easier with  
Intel<sup>®</sup> Performance-Tuning Utility**

# Parallelization Made Easier with Intel<sup>®</sup> Performance-Tuning Utility

Alexei Alexandrov, Software and Solutions Group, Intel Corporation  
Stanislav Bratanov, Software and Solutions Group, Intel Corporation  
Julia Fedorova, Software and Solutions Group, Intel Corporation  
Dr. David Levinthal, Software and Solutions Group, Intel Corporation  
Igor Lopatin, Software and Solutions Group, Intel Corporation  
Dmitry Ryabtsev, Software and Solutions Group, Intel Corporation

Index words: performance analysis, multi-core, parallelization, multi-threading, stack sampling, data access analysis

## ABSTRACT

While multi-core processors are all around us, their effective use is made much easier with performance analysis tools that enable the developer to identify parallel execution opportunities and parallel execution bottlenecks. In this paper we introduce the new profiling capabilities available in the Intel<sup>®</sup> Performance Tuning Utility. These include statistical call tree analysis based on stack sampling, profile-guided loop detection, and event-based sampling data access profiling. The coordinated use of these features allows the developer to achieve better multi-core application performance.

## INTRODUCTION

Parallel processing has been in common use for decades, but it's only recently that it became available on virtually every computer with the advent of multi-core processors. Historically, mass performance analysis tools [1, 2, 3, 4] have not generally had features designed to help identify parallel execution opportunities nor many of the common parallel execution bottlenecks. The Intel Performance Tuning Utility (Intel PTU), externally available at [5], has many of these features available in a single tool on Intel<sup>®</sup> Architecture.

Building on the experience of the Intel VTune<sup>™</sup> Performance Analyzer, Intel PTU was designed to significantly improve on the data collection and display features available and add capabilities needed for enabling and analysis of parallel execution. Initially supported instrumentation-based control flow analysis (Exact Call Graph) suffers from excessive overhead and the resulting data distortion. This was replaced with a statistical

approach to data collection based on call stack sampling in Intel PTU. The new statistical call stack sampling is supplemented with a precise call count data collection that can be used when required. Binary analysis was added to improve the disassembly displays through the use of basic blocks as the underlying execution units and to generate a control flow graph for the disassembly to simplify its interpretation. The binary analysis also enables the identification of loops, which, coupled with the performance data, allow for the identification of parallel execution opportunities. The full use of the Precise Event Based Sampling (PEBS) mechanism, only available on Intel<sup>®</sup> processors, enables simultaneous profiling by both Instruction Pointer (IP) and by data address, and a graphical filtering interface facilitates the analysis and identification of performance bottlenecks due to data access and layout issues.

All Intel PTU features are thread and CPU aware and can display data specific to either. Intel PTU works on a wide range of Windows\* and Linux\* operating system flavors and provides the same look-and-feel on all of them. It can be used from the command-line or from a GUI, which integrates into the Eclipse\* IDE.

In this paper, we first describe the new features of Intel PTU in detail, as well as the analysis models facilitated by those features. We then illustrate the process of parallel software analysis and parallel execution discovery using Intel PTU on real program examples. We continue with an outline of areas for further development such as the quality of analysis and data representation, and finally we look at modern hardware performance monitoring capabilities.

Reading this paper requires some experience in parallel program design, as well as a certain knowledge of parallel performance monitoring and analysis. The sections below should not be viewed as providing a final recipe of efficient parallel software development or as describing methods of automated parallelization. Our goal, rather, is to illustrate the information that may be of use when dealing with parallel software and how that information may be collected, presented, and best interpreted with Intel PTU in order to ease the task of exploiting parallelization opportunities and parallel performance tuning.

## NEW PERFORMANCE ANALYSIS MODELS

Performance tuning is like debugging: you'd like to avoid it but you cannot. And similar to the debugging process, you cannot do anything effectively unless you have a reliable tool that can save you a lot of time and effort. The importance of good debuggers and performance analyzers becomes critical as we move into the all-parallel world of microprocessing.

Intel PTU is meant to become such a time-saving tool. We do not pretend though that the tool can fully automate the tuning process. We simply believe that it is more important to put the burden of routine work on the tool, and let engineers think about their performance problems rather than about the tool itself.

To accomplish this, we focus on the following:

- Provide an easy way to perform repetitive data collection and analysis tasks.
- Provide effective and reliable methods of data collection and analysis that are relevant to both sequential and parallel programs.

The rest of this section describes in detail how the above goals are addressed by Intel PTU. We explicitly indicate product features that are especially valuable in the case of parallel analysis.

### Projects, Configurations, and Experiments

To be effective in repetitive performance tuning tasks Intel PTU introduces the concepts of *project* and *profile configurations*.

Project contains information about the application, working directory, input arguments, maximum data collection time, etc. — in other words, it specifies *what* should be analyzed.

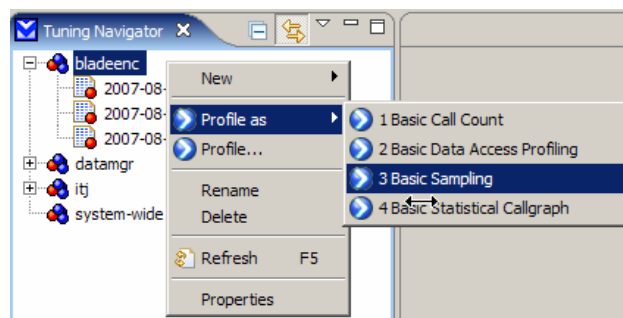
Profile configurations are a means of organizing collection methods into convenient and reusable shortcuts that can be reused for any project. Configurations can be

predefined or user-defined. The predefined configurations for Intel PTU are as follows:

- Basic Statistical Call Graph
- Basic Sampling
- Basic Call Count
- Basic Data Access Profiling

A single profile configuration can be defined for multiple Intel processors, thereby generalizing its use. For example, the predefined Basic Sampling configuration is defined to collect two performance events corresponding to the “number of cycles” and “instructions retired” events mapped to different hardware events on different processors.

Creating a project is the first thing a user does. Once it is created, the predefined configurations list can be invoked by a simple right-click on the project in the navigator, floating the mouse over the “Profile As” option and selecting one of the profiling options (Figure 1).



**Figure 1: Launching a predefined data collection**

Alternatively, right clicking on the project and selecting the “Profile...” option will invoke the configuration editor allowing users to select one of their own existing configurations or to create and invoke a new one.

After a profile configuration is applied to a project, the data are collected into an *experiment*. The basic visualization of the experiment data in Intel PTU is a tabular spreadsheet. The rows correspond to the currently chosen aggregation unit: module, function, basic block, or single address. The columns display the metrics for that region. The granularity of the aggregation unit can be selected through pull down menus (Figure 2).

Function	Module	Branch Instructions Executed	Branch Instructions Retired
ftol	msvcrt.dll	45,164	45,510
count_bits_long	bladenc.exe	10,641	10,586
fft	bladenc.exe	4,917	5,045
mdct	bladenc.exe	4,562	4,555
HalpPmTimerStallExecProc	hal.dll	3,384	0
double_huffman	bladenc.exe	3,318	3,295
psycho_anal	bladenc.exe	2,885	3,053
calc_noise	bladenc.exe	2,470	2,487
windowFilterSubband	bladenc.exe	2,125	2,204
<b>Total selected:</b>		<b>60,722</b>	<b>61,141</b>

Figure 2: Intel® PTU tabular data view

For parallel programs, Intel PTU includes the current thread identifier (TID) and the CPU identifier in the program state information so that for each collection point it is clear which thread was executing on which processor at that moment. It is possible to filter the data for a specific thread, process, or CPU by using pull-down filter menus. Specifically this is useful for analysis of thread- or CPU-balancing.

Now, let's discuss the predefined analysis methods Intel PTU suggests.

### Statistical Call Graph Analysis

Statistical Call Graph (a.k.a. Stack Sampling) collects its data by interrupting the program execution periodically (100 times per second) and capturing the current instruction pointer (IP) and the call stack. Using the IP value, it calculates how many samples occurred in a given function. This number is called Self Samples, because it corresponds to the number of samples that occurred in the function itself, not in the functions called by this function (callees). Places where a significant portion of samples occur are called *hotspots*.

The sample data can be aggregated by different units: function, module, basic block, or address. In the Intel PTU GUI the aggregation unit concept is exposed as "granularity." The function granularity is still the most popular, so it was made the default one in Hotspot view. By default, functions (rows) are sorted by the number of Self Samples, so the most active functions are displayed at the top.

A second metric for each function, Total Samples, can be defined as the number of samples in the function plus all the samples that have the function in the call stack. Thus Total Samples measures the time in the function and everything the function calls.

To illustrate this we used a simple program:

```

int main() {
    f1();
}

int f1() {
    loop 30;
    f2();
    f3();
}

void f2() {
    loop 20;
}

int f3() {
    loop 40;
    foo();
}

void foo() {
    f4();
}

void f4() {
    loop 10;
}
    
```

The time spent in each function is proportional to the iteration count of the loops. The loops in f1, f2, f3 and f4 are defined to split the total execution time of the application and thus the expected numbers of samples, in a known manner. 30% in f1, 20% in f2, 40% in f3, and 10% in f4, while main and foo are negligible.

Function	Hint	Module	Samples(self)
f3		call_chain.exe	1,644
f1		call_chain.exe	1,230
f2		call_chain.exe	829
f1		call_chain.exe	829
main		call_chain.exe	829
f4		call_chain.exe	421
			<b>4,124</b>

Function	Hint	Module	Samples(self)	Samples(total)
main		call_chain.exe	1,230	3,703
f1		call_chain.exe	1,230	3,703
f3		call_chain.exe	1,644	1,644
f2		call_chain.exe	829	829

Figure 3: Statistical Call Graph results

The top hotspot display shows that the self samples are in the ratio of 4:3:2:1 (Figure 3). The call stack expanded from f2 shows f1 and main as its callers. The four hotspots are all highlighted and the total sample count for them is shown below as 4124 samples. An important thing to note is that 829 samples for main here does not mean that we had 829 samples in the main() itself. It is all about samples in f2: we had 829 samples in f2, and for all those samples we had f1 and main as callers.

Note the pull down menus for process, thread, and module filtering of the data displayed in the hotspot view. This greatly simplifies use of the view. This technique is common to hotspot displays for all the collection modes.

The four functions were highlighted, one by one, by clicking the left mouse button and holding down the CTRL key. As the last function selected was f1, it is displayed in the Caller/Callee view. The call chain is expanded in both directions around f1 with callers of f1 shown above it and its callees shown below it. The total number of samples for f1 is equal to its self samples plus

the total number of samples for the functions it calls, f2 and f3. So we get 3703 total samples for f1 as 1230 plus 1644 plus 829. The total for f3 is equal to the total number of self samples for f3 and f4. As main is the caller of f1 it inherits the self and total times associated with f1.

Functions foo and main are not visible in the hotspot view because no samples occurred within their code ranges. Statistical Call Graph doesn't capture every call the program made. There is another collection technique called Exact Call Graph that instruments all functions in the program and can collect information about each call. However, this method has a much higher overhead. It is also very intrusive and distorts the execution of parallel programs making it impossible to map the results of this analysis to the behavior of the original program. Hence, it has a limited scope of applicability and is not always relevant for parallelization tasks. Exact Call Graph has the advantage of providing function call counts; to provide this useful information Intel PTU has a special Basic Call Count configuration.

### Profile-Guided Loop Analysis

An important step in program parallelization is deciding which parts of the program should be parallelized. Since loops are often good candidates for parallelization, Intel PTU treats loops in a very special way.

Loops are identified by analysis of the binary. This information is then used to generate entries in the hint column. The hints tell you if there is a hot loop in the function and if a hot function was called from a loop. Hot functions called from a loop can be considered for parallelization.

### Event-based Sampling

Intel processors have a powerful performance monitoring unit (PMU) that can count and interrupt execution for sampling on a wide variety of performance critical signals (e.g., CPU cycles, instructions retired, last-level cache misses, etc.). Intel PTU has made it easier to use the hundreds of performance events by displaying the sampling data in a logical and convenient manner. The event based sampling hotspot view shows an ordered spreadsheet of all functions, in all modules and processes by default. The spreadsheet can be sorted by any of the collected events. The granularity can be set to module, function (default), basic block, or instruction. A histogram of samples vs. IP can be viewed for any event with a right click option.

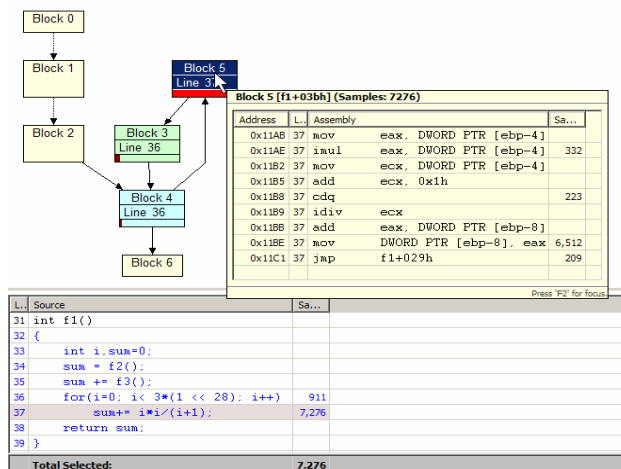


Figure 4: Source view in Intel® PTU

Double clicking on a row will open a source view display that includes a source view spreadsheet and a disassembly spreadsheet organized into units of basic blocks and a control flow graph for the basic blocks (Figure 4). The disassembly spreadsheet can be sorted by the sample totals for the basic blocks to ease the identification of hotspots. The disassembly view can be collapsed to only show the basic block data summary rows for analysis of large complex functions.

There is also the ability to compare two event-based sampling experiments. This is particularly useful for identifying the performance differences from two binaries that have been compiled differently.

As there are hundreds of events, their use must be organized into a methodology. An introduction to the use of the Intel® Core™2 processor PMU is discussed in [6]. A detailed discussion of the cycle accounting methodology on that processor is offered in [7]. The same Web site [8] also contains a number of articles about the use of the Itanium® processor PMU. There are a variety of performance issues associated with parallel execution. Their identification with the Intel Core 2 processor PMU is discussed in [9].

### Data Access Analysis

Data access tends to dominate application performance, even in single-threaded execution. Parallel execution only exacerbates this, as the number of execution units available has increased faster than the memory access capability. The actions of the processor, in response to data access requests, can be monitored with performance events counting last-level cache misses, bus traffic, and the like. What has not been generally available is the ability to analyze the application memory access behavior in terms of the data address patterns.

To collect and present performance metrics for accessed data addresses Intel PTU uses advanced features of Intel processors. Intel Itanium processor CPU supports capturing the data access address and access latency directly. On Intel Core 2, Xeon® and Pentium® 4 processors the tool uses *precise* performance events that allow the capture of the values of all the registers at a known value of IP. When coupled with the disassembly of the function, load and store operations can have their target addresses reconstructed. This feature is unique for mass-market CPUs, and for end users, the aspects of the collection mechanism are abstracted by a predefined data access configuration that is used the same way on any of the supported processors. It is also possible to define custom data access configurations using any combination of memory-related events.

Some of the more obvious objectives of address profiling would be identifying the following:

1. Cachelines that are only partially consumed, increasing memory bandwidth and wasting cache space for no benefit.
2. Cachelines shared by multiple threads unnecessarily (false sharing).
3. Variables (and cachelines) that are being thrashed during synchronization.
4. Arrays of structures that are not organized by usage, resulting in 1 above.

5. Cachelines and variable access resulting in disproportionate access latency.

Today data access analysis provides good help in pinpointing items 2 and 5, while easy identification of the rest of the items is still dependent on future development of the technology.

For data access analysis, Intel PTU provides two hotspot views in both IP and data address (Figure 5). The IP hotspot view is similar to the other hotspot views but has columns associated with data access metrics (average and total latency, reference count, page access count, etc.). The address hotspot view uses a granularity of 64 byte aligned address ranges for IA-32 and Intel® 64 Architecture-based processors and 128 byte aligned ranges on Itanium processors. These correspond to cachelines even though we use virtual rather than physical addresses. Similarly “pages” are usually defined as 4KB aligned ranges and 8KB ranges per the architecture.

The address hotspots can be expanded to show which offsets into the lines were accessed, and which threads and functions accessed the offset. This easily identifies lines that are falsely shared by multiple threads. As a result of the automatic analysis, the tool highlights such lines in pink (note those pink lines in the address hotspot view in Figure 5). However, for now the false-positives are possible, although we hope to minimize their number in the future.

Function	Module	Data Refs (%Total)	LLC Misses (%Total)	Avg. Latency	Total Latency (%Total)	Cachelines #	Pages # (%Total)
compute_rhs	sp.A	10,872,000,000 (28.5%)	118,300,000 (28.2%)	6	71,701,200,000 (30.0%)	13,379	6,006 (45.8%)
z_solve	sp.A	7,906,000,000 (20.8%)	90,700,000 (21.6%)	6	48,643,000,000 (20.4%)	7,036	4,723 (36.0%)
x_solve	sp.A	5,634,000,000 (14.8%)	46,800,000 (11.2%)	5	31,512,200,000 (13.2%)	5,617	3,987 (30.4%)
y_solve	sp.A	5,500,000,000 (14.4%)	92,300,000 (22.0%)	7	41,806,200,000 (17.5%)	5,849	4,145 (31.6%)
lhsx	sp.A	2,528,000,000 (6.6%)	7,900,000 (1.9%)	4	10,481,000,000 (4.4%)	2,051	1,532 (11.7%)
__kmp_wait_sleep	libguid...	1,680,000,000 (4.4%)	200,000 (0.0%)	3	5,099,200,000 (2.1%)	51	15 (0.1%)
lhsz	sp.A	1,396,000,000 (3.7%)	8,800,000 (2.1%)	4	6,844,600,000 (2.9%)	1,124	957 (7.3%)
<b>Total Selected:</b>		<b>5,634,000,000 (14.8%)</b>	<b>46,800,000 (11.2%)</b>	<b>5</b>	<b>31,512,200,000 (13.2%)</b>	<b>5,617</b>	<b>3,987 (30.4%)</b>

Granularity	Function	Process	sp.A	Thread	All	Module	All	Filter by selection	
-------------	----------	---------	------	--------	-----	--------	-----	---------------------	--

Cacheline Address / Offset / Thread / Function	Refs (%Total)	Avg. Latency	Total Latency (%Total)	Contributors
▼ 0x0000000017cbe40	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2
▼ Offset:0x00(0)	100,000 (0.0%)	250	25,000,000 (0.0%)	Threads: 1
▼ Thread:00004a64(0010)	100,000 (0.0%)	250	25,000,000 (0.0%)	Functions: 1
z_solve	100,000 (0.0%)	250	25,000,000 (0.0%)	
▼ Offset:0x20(32)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Threads: 1
▼ Thread:00004a63(0014)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Functions: 1
x_solve	2,000,000 (0.0%)	3	6,000,000 (0.0%)	
▶ 0x000000000297e040	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 1 Threads: 2
▶ 0x000000000158e840	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2
▼ 0x00000000017021c0	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 1

Figure 5: Data access analysis views

The two hotspot views (IP and data address) are coupled and a selection in one can be used to filter the display of the other (with control buttons indicated in Figure 5). Thus the user can select a single function, identify which lines it accesses heavily, select a set of those lines, and

then see which other functions also access those same lines. This filtering extends down to the source views.

We went through the most important features of Intel PTU. We learned the important concepts provided by the tool (project, profile configuration, and experiment). We

found out which collection and analysis capabilities are supported and identified which of them are specifically applicable for parallelization tasks. Now it's time to discuss how what we learned can be applied to solving real-world parallelization problems.

## TUNING FOR DATA-LEVEL PARALLELISM

In this section we provide a real tuning example to highlight the capabilities of Intel PTU in real-world software analysis. We start with discovering parallel execution opportunities, and then we analyze the efficiency of parallelization by locating thread interaction and data layout issues. In the course of our analysis we consider the data-level parallelism wherein different data ranges are processed in parallel on a shared-memory multi-processor.

### SP Application and Environment

For our example we took the SP code from the NAS 2.3 Benchmark Suite (NPB2.3) [10]. We started by profiling the serial version of SP, then took the OpenMP C implementation, made by the OMNI compiler team [11, 12], analyzed, and tuned it.

SP is a simulated computational fluid dynamics application. The finite difference solution is based on an approximate factorization that decouples the x, y, and z dimensions [13]. The data set we use is class A, for which a problem size is equal to 64. The simulation is done in 400 high-level iterations over time. The main loop contains the following calls:

```
for (it=1; it<=niter; it++) {
    compute_rhs();
    txinrv();
    x_solve();
    y_solve();
    z_solve();
    add();
}
```

Where `x_solve` calls `lhsx` and `ninvr`; `y_solve` – `lhsy` and `pinvr`; `z_solve` – `lhsz` and `tzetar`.

At each iteration, SP re-calculates a number of three- (64x64x64) and four-dimensional (5x64x64x64) arrays consisting of double precision floating-point numbers and consuming ~76 Mb of the memory space in total.

Our environment was Red Hat Linux\* 3.0 Update 8 running on a 2.66 GHz Quad-Core Intel® Xeon® processor 5300<sup>1</sup> series system. This system had eight cores configured in four paired CPUs, with two such pairs per package. Each CPU had a 4Mb L2 cache. The application was compiled using the Intel® Compiler 10.0 with options “-O3 -openmp -g.”

### Identifying Synchronization Overhead Using Statistical Call Graph

We started with the Basic Statistical Call Graph and Loop Analysis to understand the behavior of the serial version of SP. As can be seen in Figure 6, the tool identified a number of hotspot functions (most of the samples reside in `compute_rhs`). These hotspots have significant numbers of samples that fall in loops inside the hotspots (circled arrow icons). In addition every hotspot is called from within a loop (exclamation mark icons).

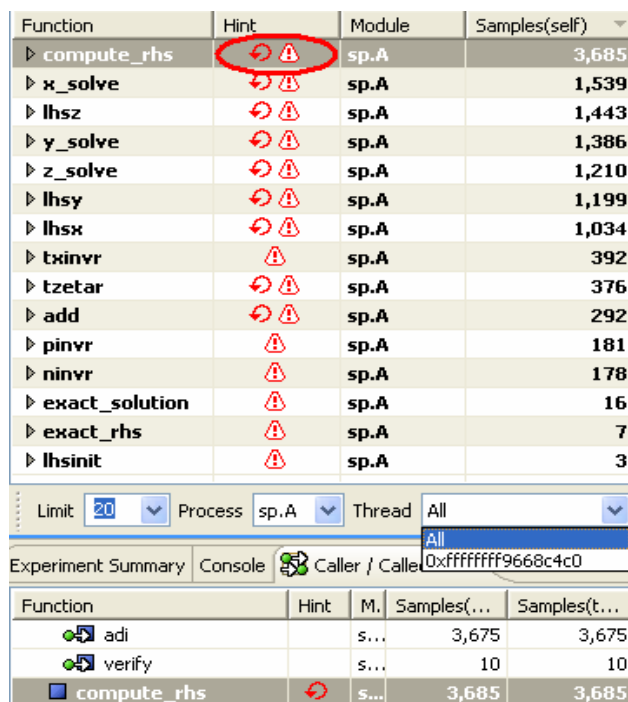


Figure 6: SP serial version hotspots in Statistical Call Graph display

Inspection of the source for the hotspot functions suggests that we cannot parallelize the program in question

<sup>1</sup> Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See [www.intel.com/products/processor\\_number](http://www.intel.com/products/processor_number) for details.

assigning a different thread to every time iteration (that is, trying to multi-thread the loop surrounding the hotspot calls), because every new time iteration depends on the arrays produced by the previous iterations. Instead, we can employ data decomposition and assign multiple threads to different iterations of the loops inside the hotspots. For such an approach, OpenMP [11] is the obvious choice.

The timing of the parallel version of SP from the OMNI package [12] when running from two to eight threads is shown in Table 1. The two-threaded version's execution time decreased from 130 seconds to 91 seconds.

**Table 1: SP OpenMP execution time and relative `__kmpc_barrier` contribution for different numbers of threads**

#of threads	1	2	4	8
SP OpenMP execution time (sec)	130	91	92	94
Total <code>__kmpc_barrier</code> (%)	N/A	16	22	27

However, running the code with four threads or more shows no additional performance gain. This clearly indicates that there are some problems with SP OpenMP implementation.

The Statistical Call Graph profile for the 4-thread execution (Figure 7) shows that one of the main hotspots, namely the `__kmp_wait_sleep` function, belongs to the `libguide` library. Another substantial hotspot that belongs to `libguide` is `__kmp_x86_pause`. These hotspots all have the `__kmpc_barrier` function on their stacks. `__kmpc_barrier` in turn is called from many SP functions. This can be seen either from the expanded stack of the `__kmp_wait_sleep` hotspot (Figure 7) or, in an aggregated form, in the caller-callee view (Figure 8). `__kmpc_barrier` is dominantly called from the `lhsx`, `lhsy` and `lhsz` functions as the total number of samples for these three functions clearly account for the majority of the time (Figure 8).

Function	Module	Samples(self)
▶ <code>compute_rhs</code>	<code>sp.A</code>	8,235
▶ <code>z_solve</code>	<code>sp.A</code>	3,879
▼ <code>__kmp_wait_sleep</code>	<code>libguide.so</code>	3,817
▼ <code>__kmp_end_split_barrier</code>	<code>libguide.so</code>	2,901
▼ <code>__kmp_barrier</code>	<code>libguide.so</code>	2,901
▼ <code>__kmpc_barrier</code>	<code>libguide.so</code>	2,901
▶ <code>lhsz</code>	<code>sp.A</code>	614
▶ <code>lhsy</code>	<code>sp.A</code>	598
▶ <code>lhsx</code>	<code>sp.A</code>	523
▶ <code>compute_rhs</code>	<code>sp.A</code>	385
▶ <code>x_solve</code>	<code>sp.A</code>	281
▶ <code>y_solve</code>	<code>sp.A</code>	196
▶ <code>z_solve</code>	<code>sp.A</code>	178
▶ <code>main</code>	<code>sp.A</code>	126
▶ <code>__kmp_barrier</code>	<code>libguide.so</code>	888
▶ <code>__kmp_fork_barrier</code>	<code>libguide.so</code>	28
▶ <code>x_solve</code>	<code>sp.A</code>	3,477
▶ <code>lhsz</code>	<code>sp.A</code>	3,226
▶ <code>y_solve</code>	<code>sp.A</code>	3,186
▶ <code>lhsy</code>	<code>sp.A</code>	2,507
▶ <code>lhsx</code>	<code>sp.A</code>	2,142
▶ <code>__kmp_x86_pause</code>	<code>libguide.so</code>	1,552
▶ <code>main</code>	<code>sp.A</code>	1,489

**Figure 7: Hotspots for SP OpenMP. Number of threads = 4. The partial stack for the `__kmp_wait_sleep` hotspot is shown.**

Function	Samples(self)	Samples(total)
▶ <code>lhsz</code>	95	1,949
▶ <code>lhsy</code>	39	1,715
▶ <code>lhsx</code>	27	1,575
▶ <code>compute_rhs</code>	0	845
▶ <code>x_solve</code>	0	648
▶ <code>y_solve</code>	2	469
▶ <code>z_solve</code>	2	433
▶ <code>main</code>	0	288
▶ <code>__kmpc_barrier</code>	199	7,956
▶ <code>__kmp_barrier</code>	293	7,746
▶ <code>__kmp_end_split_barrier</code>	11	11

**Figure 8: Caller/callee view for SP OpenMP with `__kmpc_barrier` as a target function. This view is useful in evaluating an aggregated target contribution and the relative contributions of its callers.**

The data for the `__kmpc_barrier` itself and its callees contribution are summarized in Table 1. The tables show that the number of total samples for `__kmpc_barrier` grows up to one quarter of the application's total number of samples when running with more than four threads. By total samples we mean the self sample count plus the self counts of all the functions down the call chain (callees).

The conclusion from the profiling session is that the initial SP OpenMP implementation doesn't scale because of a significant synchronization overhead exposed as a substantial number of total samples associated with the `__kmpc_barrier` function.

Note that Intel PTU significantly simplifies the identification of the total contribution of a function by automatically synchronizing views for the focus function. Thus, the Caller/Callee view displays aggregated total samples for a function selected in the Hotspot view.

To find the cause of the synchronization overhead, we look at the lhs[\*] functions code (Figure 9) and find where the OpenMP “omp for” pragmas are applied. Instead of being applied to the outer loops, they are applied to the inner loop, decomposing the leading dimensions of the multi-dimensional arrays and causing a considerable overhead at an implicit barrier.

```
// cv and rhoq are thread-shared
// as declared in global scope as static

for (i = 1; i <= grid_points[0]-2; i++) {
  for (k = 1; k <= grid_points[2]-2; k++) {
    #pragma omp for
    for (j = 0; j <= grid_points[1]-1; j++) {
      ru1 = c3c4*rho_i[i][j][k];
      cv[j] = vs[i][j][k];
      rhoq[j] = max(dy3 + con43 * ru1,
        max(dy5 + c1c5*ru1, max(dymax + ru1, dy1)));
    }

    #pragma omp for
    for (j = 1; j <= grid_points[1]-2; j++) {
      lhs[0][i][j][k] = 0.0;
      lhs[1][i][j][k] = -dtt2 * cv[j-1] - dtt1 * rhoq[j-1];
      lhs[2][i][j][k] = 1.0 + c2dtt1 * rhoq[j];
      lhs[3][i][j][k] = dtt2 * cv[j+1] - dtt1 * rhoq[j+1];
      lhs[4][i][j][k] = 0.0;
    }
  }
}
```

**Figure 9: A code fragment from the lhs function causing barrier overhead**

```
// make cv and rhoq thread-private,
// for this static declaration for them was removed

#pragma omp for private(cv, rhoq)
for (i = 1; i <= grid_points[0]-2; i++) {
  for (k = 1; k <= grid_points[2]-2; k++) {
    for (j = 0; j <= grid_points[1]-1; j++) {
      ru1 = c3c4*rho_i[i][j][k];
      cv[j] = vs[i][j][k];
      rhoq[j] = max(dy3 + con43 * ru1,
        max(dy5 + c1c5*ru1,max(dymax + ru1, dy1)));
    }

    for (j = 1; j <= grid_points[1]-2; j++) {
      lhs[0][i][j][k] = 0.0;
      lhs[1][i][j][k] = -dtt2 * cv[j-1] - dtt1 * rhoq[j-1];
      lhs[2][i][j][k] = 1.0 + c2dtt1 * rhoq[j];
      lhs[3][i][j][k] = dtt2 * cv[j+1] - dtt1 * rhoq[j+1];
      lhs[4][i][j][k] = 0.0;
    }
  }
}
```

**Figure 10: An optimized code fragment at the lhs function**

Similar problems were observed in other functions where the “omp for” pragmas were applied to the middle loop of three nested loops. We modified the initial SP OpenMP implementation by making changes to lhs\* and \*\_solve functions so that the “omp for” pragmas were properly

applied to the outermost loop. Further, we merged some separate loops under one “omp for” pragma. We also had to privatize several variables as part of the changes to ensure the correctness of the program.

The improved version of the same non-optimal code fragment (from Figure 9) is shown in Figure 10. We refer to this version of the SP code as “SP OpenMP Opt.”

## Data Layout Analysis Using Sampling and Data Access Profiling

However, while the issue of the large barrier overhead was fixed by these modifications, the overall performance and scaling did not improve much beyond two threads (see Table 2).

**Table 2: Execution time for SP OpenMP initial version and optimized version (time is in seconds)**

#of threads	1	2	4	8
SP OpenMP initial	130	91	92	94
SP OpenMP Opt	130	91	78	72

Since the SP application uses ~76 Mb of data space and our system has only 16 Mb of shared L2 cache, the memory usage approach might be the reason for the poor improvement in scaling. To prove this we launched sampling, and we collected the MEM\_LOAD\_RETIERED.L2\_LINE\_MISS event for SP OpenMP Opt running with thread numbers 1 through 8. The results (summarized in Table 3) clearly indicate that the number of L2 cache line misses grows with the increasing number of threads. Although for the code to be scalable the number of cache misses should remain the same or even decrease.

Profiling runs for four and eight threads reveal that compute\_rhs and z\_solve functions are the main hotspots, contributing ~28% and ~12% L2 cache line misses, respectively, in both runs. The other main hotspots are the x\_solve and y\_solve functions.

**Table 3: Count of the MEM\_LOAD\_RETIERED.L2\_LINE\_MISS event for the SP OpenMP Opt code**

# of threads	1	2	4	8
Event count	3.5E+08	3.3E+08	4.1E+08	6.3E+08

The Data Profiling analysis for the four- and eight-thread runs confirms the same functions as the memory access bottlenecks. The Data Access Display shows that compute\_rhs, z\_solve, y\_solve and x\_solve functions

are also hotspots in terms of Last Level Cache (LLC) misses, Total Latency, and Cachelines accessed (Figure 11).

Function	LLC Misses (%Total)	Avg. L...	Total Latency (%Total)	Cachelines #
compute_rhs	118,300,000 (28.2%)	6	71,701,200,000 (30.0%)	13,379
z_solve	90,700,000 (21.6%)	6	48,643,000,000 (20.4%)	7,036
x_solve	46,800,000 (11.2%)	5	31,512,200,000 (13.2%)	5,617
y_solve	92,300,000 (22.0%)	7	41,806,200,000 (17.5%)	5,849
lhsx	7,900,000 (1.9%)	4	10,481,000,000 (4.4%)	2,051
__kmp_wait_sleep	200,000 (0.0%)	3	5,099,200,000 (2.1%)	51
lhsz	8,800,000 (2.1%)	4	6,844,600,000 (2.9%)	1,124
lhsy	7,900,000 (1.9%)	4	6,147,400,000 (2.6%)	1,011
main	45,300,000 (10.8%)	14	15,424,800,000 (6.5%)	1,929
<b>Total Selected:</b>	<b>348,100,000 (82.9%)</b>	<b>6</b>	<b>193,662,600,000 (81.1%)</b>	<b>31,881</b>

Cacheline Address / Offset / Thread / Function	Avg. Latency	Total Latency (%Total)
0x0000000002681500	15	30,700,000 (0.0%)
0x00000000040801740	15	30,700,000 (0.0%)
0x000000000298e000	15	30,700,000 (0.0%)
0x0000000000280e8c0	15	30,700,000 (0.0%)
0x00000000041be1740	15	30,700,000 (0.0%)
0x0000000000266d040	15	30,700,000 (0.0%)
0x00000000002dabfc0	15	30,700,000 (0.0%)
0x00000000002762300	15	30,700,000 (0.0%)
0x000000000016b4c80	15	30,700,000 (0.0%)

**Figure 11: Main hotspots for the SP OpenMP Opt 8 threads run in the Data Access Display. The figure also illustrates how to filter the cachelines accessed by selected functions.**

The reason for the growing number of cache misses (refer to Table 3) for four- and eight-thread runs might be interfering data accesses. The data access profile allows us to investigate if there are access contention issues for the cachelines used in the hotspots compute\_rhs, z\_solve, y\_solve, and x\_solve, particularly those caused by the threads running on different cores but accessing the same cachelines. This will show if there are any contentious lines associated with high average latencies and accessed by several threads.

To explore contention issues we ran the SP OpenMP Opt version with eight threads, bounding each thread to a distinct core using the KMP\_AFFINITY environment variable supported by the Intel Compiler OpenMP runtime library. The execution time and hotspots do not change with respect to the non-bound run.

In data access view we select the hotspot functions and use the “Filter by Selection in Code Hotspots” button (circled in Figure 11), to display only the cachelines accessed by these functions.

A number of filtered cachelines are marked in pink as likely suffering from false-sharing. But false-sharing (as well as true-sharing) is a particular case of thread access contention. Consequently, we sort cachelines by average latency and select the high latency lines. We then filter back either on a specific cacheline or a few of them to identify the functions that are associated with the

contention. This is done by using the “Filter by Selection in Data Hotspots” button (triangled in Figure 11).

We found a number of cases (one example is in Figure 12) where the same cachelines were accessed from different threads by the functions compute\_rhs & x\_solve, x\_solve & y\_solve, y\_solve & z\_solve. Specifically, Figure 12 demonstrates that the same highlighted cacheline was accessed by the different threads in the x\_solve and y\_solve functions. The second access (by y\_solve, as it called after x\_solve in the code) is associated with the high latency (250 cycles) equal to an L2 miss penalty.

Function	Module	Data Refs (%Total)	LLC Misses (%Total)	Avg. L...	Total Latency (%Total)
x_solve	sp.A	4,000,000 (0.0%)	0 (0.0%)	3	12,000,000 (0.0%)
y_solve	sp.A	100,000 (0.0%)	100,000 (0.0%)	250	25,000,000 (0.0%)
z_solve	sp.A	100,000 (0.0%)	100,000 (0.0%)	250	25,000,000 (0.0%)

Granularity	Function	Process	sp.A	Thread	All	Module	All
Cacheline Address / Offset / Thr...	Refs (%Total)	Av...	Total Latency (%Total)	Contributors			
0x00000000016b4c80	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2			
▼ Offset:0x10(16)	100,000 (0.0%)	250	25,000,000 (0.0%)	Threads: 1			
▼ Thread:00004a5f(0011)	100,000 (0.0%)	250	25,000,000 (0.0%)	Functions: 1			
y_solve	100,000 (0.0%)	250	25,000,000 (0.0%)				
▼ Offset:0x30(48)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Threads: 1			
▼ Thread:00004a5c(0006)	2,000,000 (0.0%)	3	6,000,000 (0.0%)	Functions: 1			
x_solve	2,000,000 (0.0%)	3	6,000,000 (0.0%)				
▶ 0x00000000002916f40	2,000,000 (0.0%)	15	30,700,000 (0.0%)	Offsets: 2 Threads: 2			

**Figure 12: The IP hotspot view filtered by the selected cacheline**

We drill down from the filtered hotspot view to the source view (now only displaying the filtered accesses) of the functions, e.g., x\_solve and y\_solve ones, to identify the source lines that generated the access contention.

The source code identified by the access counts on these lines in turn identifies a number of cache contention patterns. Figure 13 displays the typical one we discovered.

In this case the x\_solve function writes to the elements of the arrays rhs and lhs, and the y\_solve function reads from them. The “omp for” pragma is placed in such a way that the data decomposition of these arrays is different in these two function fragments. In x\_solve the decomposition, over the third index of rhs, causes thread\_1 to write into rhs[\*][\*][τ1\_range][\*], thread\_2 writes into rhs[\*][\*][τ2\_range][\*] and so on. While in y\_solve, the decomposition is over the second index, so thread\_1 here reads from rhs[\*][τ1\_range][\*][\*]. This results in multiple cores having to shuffle the cachelines between themselves as they execute x\_solve and then y\_solve. This in turn results in a large number of load-driven cache misses and the resulting execution stalls.

We didn’t go further with optimizing the SP code since our purpose was just to demonstrate how an application using data parallelism is analyzed and tuned with Intel PTU.

Possible ways for further optimization could be code transformations to make the “omp for” pragmas apply to the outermost loops, iterating over the same dimension indices. This would decrease the shuffling of the cachelines between cores and thereby improve the performance.

It would be also useful to consider decreasing some array sizes (to apply data blocking optimization), as described in [13]. This would bring an even bigger performance gain due to a more efficient cache usage.

```
void x_solve(void) {
<...>
#pragma omp for
  for (j = 1; j <= grid_points[1]-2; j++) {
    for (i = 0; i <= grid_points[0]-3; i++) {
      for (k = 1; k <= grid_points[2]-2; k++) {
        <...>
        for (m = 0; m < 3; m++) {
          rhs[m][i][j][k] = fac1*rhs[m][i][j][k];
        }
        <...>
        lhs[n+2][i][j][k] = lhs[n+2][i][j][k] <...>
        <...>
      }
    }
  }

void y_solve(void) {
<...>
#pragma omp for
  for (i = 1; i <= grid_points[0]-2; i++) {
    for (j = 0; j <= grid_points[1]-3; j++) {
      for (k = 1; k <= grid_points[2]-2; k++) {
        <...>
        for (m = 0; m < 3; m++) {
          rhs[m][i][j][k] = fac1*rhs[m][i][j][k];
        }
        <...>
        lhs[n+2][i][j][k] = lhs[n+2][i][j][k] <...>
      }
    }
  }
}
```

**Figure 13: Code fragments causing cacheline contention**

In this section we have shown that Statistical Call Graph analysis may be very helpful in the initial stages of parallel code tuning. Proceeding with the analysis requires some knowledge of the processor architecture to identify the hardware events to collect and to interpret the collected data. Advanced scalability estimations can hardly be performed without the help of data access profiling whose automatic analysis and flexible filtering interface enable pinpointing of such problems as cache contention (particularly false-sharing) and high latency loads at the source code level.

## CHALLENGES AND FUTURE DIRECTIONS

Extensibility was among the primary design concepts of the Intel PTU architecture, which may enable us to integrate more advanced profiling techniques in the future, many of which we can already define and describe.

The first step that we would like to take in the near future is to extend the Statistical Call Graph (which is now time-based in Intel PTU) to also use rich event-based sampling capabilities. The major advantages of this are expected to be as follows:

- Increased sampling granularity (as the sampling interval will no longer be limited by the operating system timer resolution and task scheduler properties).
- Higher correlation of the sampled execution paths with the architectural characteristics of a computer system.

In data profiling, a unique problem is dealing with arrays of large structures. Being able to display the access pattern in terms of the structure size granularity allows the user to split the structures by usage, reducing bandwidth and increasing cache utilization efficiency.

Another important improvement and a major challenge with regard to data access analysis is the need to operate on the categories that are understandable by a programmer. This means we need to switch from raw addresses (which may mean anything) to the actual variable names, allocation blocks, and so on.

A very promising technology that we are also going to implement is the ability to handle the lowest-level operating system task context switches. This should enable the retrieval of information about thread synchronization patterns, excessive synchronization, overall processor utilization by multiple threads, thread migration between processors, thread switch overhead, and other characteristics that are vital for a detailed analysis of heavily threaded, multi-component applications.

The above described data collection challenges and improvements necessitate changes in the visualization as well. Thus, we would like to introduce a timeline view, the natural representation of thread activity over time. The timeline will reflect the state of threads, their location with respect to processors and cores, and the actual performance characteristics for each thread activity point in time. The overtime representation is supposed to facilitate intuitive understanding of the logic of a parallel program and thread state transition patterns; and it may help to determine distinct phases within the program’s operation flow. Most importantly, the timeline is designed to be fully integrated into the rest of the existing views to simplify navigation, incorporate new cross-filtering modes, and make it possible to quickly obtain aggregated characteristics for each thread execution point, state, or phase.

## CONCLUSION

The performance analysis features available in the Intel Performance Tuning Utility assist at virtually every stage of both parallel and sequential software performance tuning and may be extremely helpful at the preliminary stages of determining parallelization strategies.

We discussed data-level decomposition strategy in real program examples and illustrated how the efficiency of a parallel implementation can be estimated, and which steps should be performed to optimize a parallel program using the Intel Performance Tuning Utility.

Being easy to use and powerful at the same time, Intel PTU is growing its customer base inside Intel for solving today's problems and serving as a vehicle for exploring new features for future commercial tools.

We plan on improving Intel PTU with newer performance data-collection techniques and analysis models to keep pace with user needs and modern processor architecture developments. Intel PTU is available for an external download from [Whatif.intel.com](http://Whatif.intel.com) Web site [5].

## REFERENCES

- [1] Intel VTune™ Performance Analyzer, at <http://www3.intel.com/cd/software/products/asm-na/eng/vtune/239144.htm>
- [2] Sun Studio Performance Analyzer, at <http://developers.sun.com/sunstudio/>
- [3] Optimizing with Shark, at [http://developer.apple.com/tools/shark\\_optimize.html](http://developer.apple.com/tools/shark_optimize.html)
- [4] Gprof, at <http://www.gnu.org/software/binutils/binutils.html>
- [5] Intel® What If site, at <http://Whatif.intel.com>
- [6] D. Levinthal, "Introduction to Performance Analysis on Intel Core 2 Duo Processors," at <http://www.devx.com/go-parallel/Link/33305>
- [7] D. Levinthal, "Execution-based Cycle Accounting on Intel Core 2 Processors," at <http://www.devx.com/go-parallel/Link/33315>
- [8] "Go Parallel Web Site," at <http://www.devx.com/go-parallel/Door/32532>
- [9] D. Levinthal, "Analyzing and Resolving Multi-Core Non Scaling on Intel Core 2 Processors," at <http://www.devx.com/go-parallel/Link/34762>
- [10] "NAS Parallel Benchmarks," at <http://www.nas.nasa.gov/Resources/Software/npb.html>
- [11] "Open MP standard," at <http://www.openmp.org>

[12] "OpenMP C versions of NPB2.3," at <http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html>

[13] H. Jin, M. Frumkin, J. Yan, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," at <http://www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf>

## AUTHORS' BIOGRAPHIES

**Alexei Alexandrov** is a Senior Software Engineer in the Software and Solutions Group at Intel. His interests include building and designing modern performance analysis tools, large software development, CPU micro-architecture, and performance analysis. Alexei has a Ph.D. degree from the Saratov State Technical University. His e-mail is [alexei.alexandrov@intel.com](mailto:alexei.alexandrov@intel.com).

**Stanislav Bratanov** is a Research Engineer in the Software and Solutions Group at Intel. His research interests include multi-processor software platforms, operating system environments, software performance monitoring and analysis systems, and platform-dependent media data coding. He graduated from Nizhniy Novgorod State University, Russia. His e-mail is [stanislav.bratanov@intel.com](mailto:stanislav.bratanov@intel.com).

**Julia Fedorova** is a Senior Software Engineer in the Software and Solutions Group at Intel. Her research interests are performance analysis tools, tuning and optimization, and data access analysis. Prior to Intel she worked in the Russian Nuclear Center. Julia has an M. Sc. degree in Computational Physics from the Moscow Engineering-Physical Institute. Her e-mail is [julia.fedorova@intel.com](mailto:julia.fedorova@intel.com).

**David Levinthal** is a Senior Software Engineer in the Software and Solutions Group at Intel. His research interests include hardware performance events, computer architecture, and software optimization. He holds a Physics degrees from the University of California at Berkeley and Columbia University. He was a Professor of Physics at Florida State University. He has been awarded the DOE OJI award, the NSF PYI award, and a Sloan Foundation Fellowship. His e-mail is [david.a.levinthal@intel.com](mailto:david.a.levinthal@intel.com).

**Igor Lopatin** is a Software Engineer in the Software and Solutions Group at Intel. His research interests include software for multi-core architectures and tools based on dynamic binary instrumentation techniques. He graduated from Nizhny Novgorod State University, Russia. His e-mail is [igor.loopatin@intel.com](mailto:igor.loopatin@intel.com).

**Dmitry Ryabtsev** is a Senior Software Engineer in the Software and Solutions Group at Intel. He has worked on the VTune Performance Analyzer and currently is

focusing on DAP for Intel PTU. He received his B.S. and M.S. degrees from the Nizhny Novgorod State University, Russia. His e-mail is dimitry.ryabtsev at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

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

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS<sup>®</sup> Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from  
<http://www.intel.com>.

Additional legal notices at:  
<http://www.intel.com/sites/corporate/tradmarx.htm>.

For further information visit:

[developer.intel.com/technology/itj/index.htm](http://developer.intel.com/technology/itj/index.htm)