Hyper-Threading Technology: Impact on Compute-Intensive Workloads

William Magro, Software Solutions Group, Intel Corporation
Paul Petersen, Software Solutions Group, Intel Corporation
Sanjiv Shah, Software Solutions Group, Intel Corporation

Index words: SMP, SMT, Hyper-Threading Technology, OpenMP, Compute Intensive, Parallel Programming, Multi-Threading

ABSTRACT

Intel's recently introduced Hyper-Threading Technology promises to increase application- and system-level performance through increased utilization of processor resources. It achieves this goal by allowing the processor to simultaneously maintain the context of multiple instruction streams and execute multiple instruction streams or *threads*. These multiple streams afford the processor added flexibility in internal scheduling, lowering the impact of external data latency, raising utilization of internal resources, and increasing overall performance.

We compare the performance of an Intel® Xeon™ processor enabled with Hyper-Threading Technology to that of a dual Xeon processor that does not have Hyper-Threading Technology on a range of compute-intensive, data-parallel applications threaded with OpenMP[1]. The applications include both real-world codes and hand-coded "kernels" that illustrate performance characteristics of Hyper-Threading Technology.

The results demonstrate that, in addition to functionally decomposed applications, the technology is effective for many data-parallel applications. Using hardware performance counters, we identify some characteristics of applications that make them especially promising candidates for high performance on threaded processors.

Finally, we explore some of the issues involved in threading codes to exploit Hyper-Threading Technology, including a brief survey of both existing and still-needed tools to support multi-threaded software development.

INTRODUCTION

While the most visible indicator of computer performance is its clock rate, overall system performance is also proportional to the number of instructions retired per clock cycle. Ever-increasing demand for processing speed has driven an impressive array of architectural innovations in processors, resulting in substantial improvements in clock rates and instructions per cycle.

One important innovation, super-scalar execution, exploits multiple execution units to allow more than one operation to be in flight simultaneously. While the performance potential of this design is enormous, keeping these units busy requires super-scalar processors to extract independent work, or instruction-level parallelism (ILP), directly from a single instruction stream.

Modern compilers are very sophisticated and do an admirable job of exposing parallelism to the processor; nonetheless, ILP is often limited, leaving some internal processor resources unused. This can occur for a number of reasons, including long latency to main memory, branch mis-prediction, or data dependences in the instruction stream itself. Achieving additional performance often requires tedious performance

---

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[1] OpenMP is an industry-standard specification for multi-threading data-intensive and other highly structured applications in C, C++, and Fortran. See www.openmp.org for more information.

analysis, experimentation with advanced compiler optimization settings, or even algorithmic changes. Feature sets, rather than performance, drive software economics. This results in most applications never undergoing performance tuning beyond default compiler optimization.

An Intel® processor with Hyper-Threading Technology offers a different approach to increasing performance. By presenting itself to the operating system as two logical processors, it is afforded the benefit of simultaneously scheduling two potentially independent instruction streams [1]. This explicit parallelism complements ILP to increase instructions retired per cycle and increase overall system utilization. This approach is known as simultaneous multi-threading, or SMT.

Because the operating system treats an SMT processor as two separate processors, Hyper-Threading Technology is able to leverage the existing base of multi-threaded applications and deliver immediate performance gains.

To assess the effectiveness of this technology, we first measure the performance of existing multi-threaded applications on systems containing the Intel® Xeon™ processor with Hyper-Threading Technology. We then examine the system's performance characteristics more closely using a selection of hand-coded application kernels. Finally, we consider the issues and challenges application developers face in creating new threaded applications, including existing and needed tools for efficient multi-threaded development.

APPLICATION SCOPE

While many existing applications can benefit from Hyper-Threading Technology, we focus our attention on single-process, numerically intensive applications. By numerically intensive, we mean applications that rarely wait on external inputs, such as remote data sources or network requests, and instead work out of main system memory. Typical examples include mechanical design analysis, multi-variate optimization, electronic design automation, genomics, photo-realistic rendering, weather forecasting, and computational chemistry.

A fast turnaround of results normally provides significant value to the users of these applications through better quality products delivered more quickly to market. The data-intensive nature of these codes, paired with the demand for better performance, makes them ideal candidates for multi-threaded speed-up on shared memory multi-processor (SMP) systems.

We considered a range of applications, threaded with OpenMP, that show good speed-up on SMP systems. The applications and their problem domains are listed in Table 1. Each of these applications achieves 100% processor utilization from the operating system's point of view. Despite external appearances, however, *internal* processor resources often remain underutilized. For this reason, these applications appeared to be good candidates for additional speed-up via Hyper-Threading Technology.

Table 1: Applications type

Code	Description
A1	Mechanical Design Analysis (finite element method) This application is used for metal-forming, drop testing, and crash simulation.
A2	Genetics A genetics application that correlates DNA samples from multiple animals to better understand congenital diseases.
A3	Computational Chemistry This application uses the self-consistent field method to compute chemical properties of molecules such as new pharmaceuticals.
A4	Mechanical Design Analysis This application simulates the metal-stamping process.
A5	Mesoscale Weather Modeling This application simulates and predicts mesoscale and regional-scale atmospheric circulation.
A6	Genetics This application is designed to generate Expressed Sequence Tags (EST) clusters, which are used to locate important genes.
A7	Computational Fluid Dynamics This application is used to model free-surface and confined flows.
A8	Finite Element Analysis This finite element application is specifically targeted toward geophysical engineering applications.
A9	Finite Element Analysis This explicit time-stepping application is used for crash test studies and computational fluid dynamics.

One might suspect that, for applications performing very similar operations on different data, the instruction streams might be too highly correlated to share a

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

threaded processor's resources effectively. Our results show differently.

METHODOLOGY

To assess the effectiveness of Hyper-Threading Technology for this class of applications, we measured the performance of existing multi-threaded executables, with no changes to target the threaded processor specifically.

We measured the elapsed completion time of stable, reproducible workloads using operating-system-provided timers for three configurations:

1. single-threaded execution on an single-processor SMT system
2. dual-threaded execution on a single-processor SMT system
3. dual-threaded execution on a dual-processor, non-SMT system

We then computed application speed-up as the ratio of the elapsed time of a single-threaded run to that of a multi-threaded run. Using the Intel VTune™ Performance Analyzer, we gathered the following counter data directly from the processor during a representative time interval of each application[2]:

- Clock cycles
- Instructions retired
- Micro-operations retired
- Floatingpoint instructions retired

From this raw data, we evaluated these ratios:

- Clock cycles per instruction retired (CPI)
- Clock cycles per micro-operation retired (CPu)
- Fractional floating-point instructions retired (FP%)

APPLICATION RESULTS

Naturally, highly scalable applications; that is, those that speed up best when run on multiple, physical processors, are the best candidates for performance improvement on a threaded processor. We expect less scalable applications to experience correspondingly smaller potential benefits.

As shown in Figure 1, this is generally the case, with all of the applications, except application A1, receiving a significant benefit from the introduction of Hyper-Threading Technology. It is important to note that the applications realized these benefits with little to no incremental system cost and no code changes.

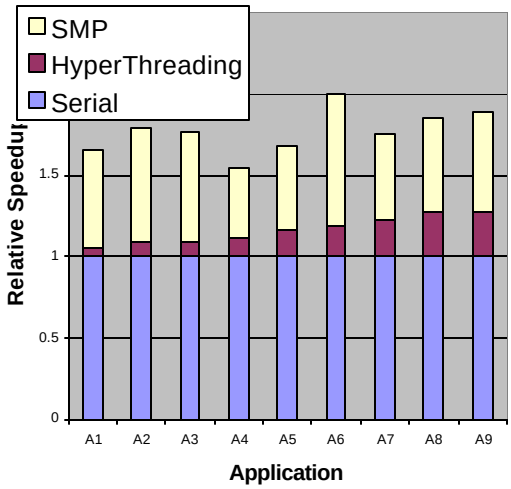

Figure 1: Application relative speed-up

Because the Intel® Xeon™ processor is capable of retiring up to three micro-operations per cycle, the best-case value of clocks per micro-op (CPu) is 1/3. Table 2 shows counter data and performance results for the application experiments. The comparatively high CPI and CPu values indicate an individual stream does not typically saturate internal processor resources. While not sufficient, high CPu is a necessary condition for good speed-up in an SMT processor.

™ VTune is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[2] The counters and their significance are described in the Appendix.

® Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

™ Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Application	Cycles/instruction	Cycles/uop	FP%	SMT speedup	SMP Speedup
A1	2.04	1.47	29	1.05	1.65
A2	1.11	0.89	4.6	1.09	1.79
A3	1.69	0.91	16	1.09	1.77
A4	1.82	1.29	20	1.11	1.54
A5	2.48	1.45	36	1.17	1.68
A6	2.54	1.60	0.1	1.19	2.00
A7	2.80	2.05	10	1.23	1.75
A8	1.69	1.27	19	1.28	1.85
A9	2.26	1.76	20	1.28	1.89

Table 2: Counter data and performance results

Exactly which resources lie idle, however, is not clear. The fraction of floating-point instructions (FP%) gives one indication of the per-stream instruction mix. For the chosen applications, the FP% ranges from zero, for application A6, to the range of 4.6% to 35.5% for the remaining applications. It may seem unusual that the FP% of these numerically intensive applications is so low; however, even in numerically intensive code, many other instructions are used to index into arrays, manage data structures, load/store to memory, and perform flow control. The result can be a surprisingly balanced instruction mix.

Even though the instruction mix *within* a stream may be varied, a data parallel application typically presents pairs of similar or even identical instruction streams that could compete for processor resources at each given moment. The performance results, however, show that Hyper-Threading Technology is able to overlap execution of even highly correlated instruction streams effectively. To understand how this can occur, consider two threads consisting of identical instruction streams. As these threads execute, spatial correlation exists only with particular temporal alignments; a slight shift in the timing of the streams can eliminate the correlation, allowing a more effective interleaving of the streams and their resource demands. The net result is that two identical but time-shifted instruction streams can effectively share a pool of resources.

By reducing the impact of memory latency, branch mis-prediction penalties, and stalls due to insufficient ILP, Hyper-Threading Technology allows the Xeonprocessor to more effectively utilize its internal resources and increase system throughput.

TEST KERNEL RESULTS

To examine these effects more closely, we developed four test kernels. The first two kernels (int_mem and dbl_mem) illustrate the effects of latency hiding in the memory hierarchy, while the third kernel (int_dbl) attempts to avoid stalls due to low ILP. The fourth kernel (matmul) and a corresponding, tuned library function illustrate the interplay between high ILP and SMT speed-up. The performance results of all the kernels are shown in Table 3. The int_mem kernel, shown in Figure 2, attempts to overlap cache misses with integer operations. It first creates a randomized access pattern into an array of cache-line-sized objects, then indexes into the objects via the randomized index vector and performs a series of addition operations on the cache line.

```
#pragma omp for
    for (i = 0; i < buf_len; ++i) {
        j = index[ i ];
        for (k = 0; k < load; ++k) {
            buffer[ j ][ 0 ] += input;
            buffer[ j ][ 1 ] += input;
            buffer[ j ][ 2 ] += input;
            buffer[ j ][ 3 ] += input;
        }
    }
```

Figure 2: The int_mem benchmark

Table 3: Kernel performance results

Code	Benchmark	CPI	CPuops	FP%	SMT Speed-up
M1	int_mem (load=32)	1.99	0.94	0.0%	1.08
M2	int_mem (load=4)	6.91	3.61	0.0%	1.36
M3	dbl_mem (load=32)	1.81	1.47	23.2%	1.90
M4	int_dbl	3.72	1.63	9.8%	1.76
M5	matmul	2.17	1.60	34.5%	1.64
M6	dgemm	1.63	1.58	58.0%	1.00

We tested two variants (M1 and M2). In the first, we assigned a value of 32 to the parameter "load"; in the second test, "load" was 4. The larger value of "load" allows the processor to work repeatedly with the same data. Cache hit rates are consequently high, as is integer unit utilization. Smaller values of "load" cause the code to access second-level cache and main memory more often, leading to higher latencies and increased demand on the memory subsystem. Provided these additional accesses do not saturate the memory bus bandwidth, the processor can overlap the two threads' operations and effectively hide the memory latency. This point is demonstrated by the inverse relationship between clocks per instruction and speed-up.

The dbl_mem kernel is identical to int_mem, but with the data variables changed to type "double." The results with "load" equal to 32 (M3) demonstrate the same effect, instead interleaving double-precision floating-point instructions with cache misses. In addition, the floating-point operations can overlap with the supporting integer instructions in the instruction mix to allow the concurrent use of separate functional units resulting in near-linear speed-up.

The int_dbl kernel (M4), shown in Figure 3, calculates an approximation to Pi via a simple Monte Carlo method. This method uses an integer random number generator to choose points in the x-y plane from the range [-1…1]. It then converts these values to floating point and uses each point's distance from the origin to determine if the point falls within the area of the unit radius circle. The fraction of points that lies within this circle approximates Pi/4. Like dbl_mem, this kernel achieves excellent speed-up, but for a different reason: the different functional units inside the processor are utilized simultaneously.

```
#pragma omp for reduction(+:count)
    for (i = 0; i < NPOINTS; ++i) {
        double x, y;
        //  guess  returns  a  pseudo-random
number
        x = guess(&seed, 2.0)-1.0;
        y = guess(&seed, 2.0)-1.0;
        if ( sqrt(x*x + y*y) <= 1.0 ) {
            /* The current point is
               inside the circle... */
            ++count;
        }
    }
```

Figure 3: The int_dbl benchmark

The "matmul" kernel (M5), shown in Figure 4, computes the product of two 1000 x 1000 matrices using a naï ve loop formulation written in FORTRAN. Comparing its absolute performance and speed-up to that of a functionally equivalent, but hand-tuned library routine illustrates the effect of serial optimization on the effectiveness of Hyper-Threading Technology. The naï ve loop formulation (M5) has comparatively poor absolute performance, executing in 3.4 seconds, but achieves good SMT speed-up. The hand-optimized dgemm (M6) library routine executes in a fraction of the time (0.6s), but the speed-up vanishes. The highly tuned version of the code effectively saturates the processor, leaving no units idle[3].

```
!$omp parallel do
      DO  26  J = 1,N
          DO  24  K = 1,N
              DO  22  I = 1,N
                   C(I,J) = C(I,J) + A(I,K)
* B(K,J)
   22         CONTINUE
   24     CONTINUE
   26 CONTINUE
```

[3] Note that the FP% for M6 is due to SIMD packed double precision instructions, rather than the simpler x87 instructions used by the other test codes.

Figure 4: The Matmul kernel

MEMORY HIERARCHY EFFECTS

Depending on the application characteristics, Hyper-Threading Technology's shared caches [1] have the potential to help or hinder performance. The threads in data parallel applications tend to work on distinct subsets of the memory, so we expected this to halve the effective cache size available to each logical processor. To understand the impact of reduced cache, we formulated a very simplified execution model of cache-based system.

In a threaded microprocessor with two logical processors, the goal is to execute both threads with no resource contention issues or stalls. When this occurs, two fully independent threads should be able to execute an application in half the time of a single thread. Likewise, each thread can execute up to 50% more slowly than the single-threaded case and still yield speed-up.

Figure 5 exhibits the approximate time to execute an application on a hypothetical system with a three-level memory hierarchy consisting of registers, cache, and main memory.

Given:

N = Number of instructions executed
F_{memory} = Fraction of N that access memory
G_{hit} = Fraction of loads that hit the cache
T_{proc} = #cycles to process an instruction
T_{cache} = #cycles to process a hit
T_{memory} = #cycles to process a miss

T_{exe} = Execution time

Then:

$$T_{exe}/N = (1 - F_{memory})\, T_{proc} + F_{memory}\, [G_{hit}\, T_{cache} + (1 - G_{hit})\, T_{memory}]$$

Figure 5: Simple performance model for a single-level cache system

While cache hit rates, G_{hit}, cannot be easily estimated for the shared cache, we can explore the performance impact of a range of possible hit rates. We assume F_{memory}=20%, T_{proc}=2, T_{cache}=3, and T_{memory}=100. For a given cache hit rate in the original, single-threaded execution, Figure 6 illustrates the effective miss rate, T'_{miss}, which would cause the thread to run twice as slowly as in serial. Thus, any hit rate that falls in the shaded region between the curves should result in overall speed-up when two threads are active.

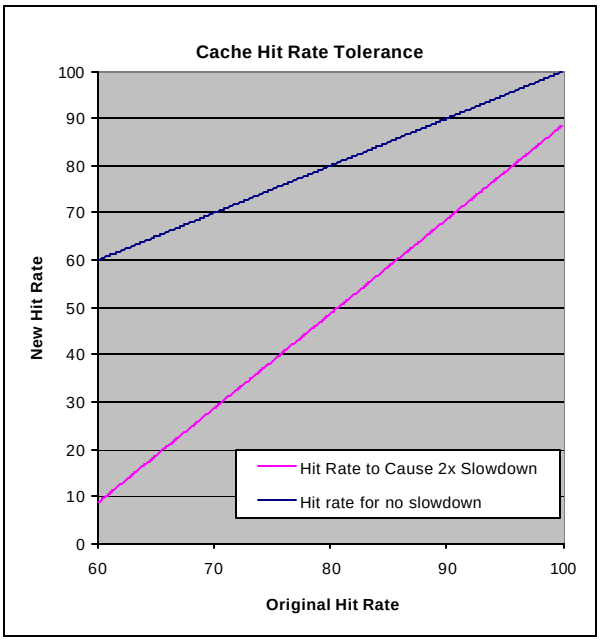

Figure 6: Hit rate tolerance for 2x slowdown in performance

The shaded region narrows dramatically as the original cache hit rate approaches 100%, indicating that applications with excellent cache affinity will be the least tolerant of reduced effective cache size. For example, when a single-threaded run achieves a 60% hit rate, the dual-threaded run's hit rate can be as low as 10% and still offer overall speed-up. On the other hand, an application with a 99% hit rate must maintain an 88% hit rate in the smaller cache to avoid slowdown.

TOOLS FOR MULTI-THREADING

It is easy to see that the existence of many multi-threaded applications increases the utility of Hyper-Threading Technology. In fact, every multi-threaded application can potentially benefit from SMT without modification. On the other hand, if no applications were multi-threaded, the only obvious benefits from SMT would be throughput benefits from multi-process parallelism. Shared memory parallel computers have existed for more than a decade, and much of the performance benefits of multi-threading have been available, yet few multi-threaded applications exist. What are some of the reasons for this lack of multi-threaded applications, and how might SMT technology change the situation?

First and foremost among these reasons is the difficulty of building a correct and well-performing multi-threaded application. While it is not impossible to build such applications, it tends to be significantly more difficult

than building sequential ones. Consequently, most developers avoid building multi-threading applications until their customers demand additional performance. The following constraints often drive performance requirements:

- Real-time requirements to accomplish some computing task that cannot be satisfied by a single processor, e.g., weather forecasting, where a 24-hour forecast has value only if completed and published in well under 24 hours.
- Throughput requirements, usually in interactive applications, such that users are not kept waiting for too long, e.g., background printing while editing in a word processor.
- Turnaround requirement, where job completion time materially impacts the design cycle, e.g., computational fluid dynamics used in the design of aircraft, automobiles, etc.

Most software applications do not have the above constraints and are not threaded. A good number of applications do have the throughput requirement, but that particular one is easier to satisfy without particular attention to correctness or performance.

Another reason for the lack of many multi-threaded applications has been the cost of systems that can effectively utilize multiple threads. Up until now, the only kinds of systems that could provide effective performance benefits from multiple threads were expensive multiple-processor systems. Hyper-Threading Technology changes the economics of producing multi-processor systems, because it eliminates much of the additional "glue" hardware that previous systems needed.

Economics alone cannot guarantee a better computing experience via the efficient utilization of Hyper-Threading Technology. Effective tools are also necessary to create multi-threaded applications. What are some of the capabilities of these tools? Do such tools already exist in research institutions?

One of the difficulties is the lack of a good programming language for multi-threading. The most popular multi-threading languages are the POSIX* threads API and the Windows* Threads API. However, these are the threading equivalent of assembly language, or C at best. All the burden of creating high-level structures is placed upon the programmer, resulting in users making the same mistakes repeatedly. Modern programming languages like Java and C# include threading as a part of the language, but again few high-level structures are available for programmers. These languages are only marginally better than the threading APIs. Languages like OpenMP [3,5] do offer higher-level constructs that address synchronous threading issues well, but they offer little for asynchronous threading. Even for synchronous threading, OpenMP [3,5] has little market penetration outside the technical computing market. If OpenMP [3,5] can successfully address synchronous threading outside the technical market, it needs to be deployed broadly to ease the effort required to create multi-threaded applications correctly. For asynchronous threading, perhaps the best model is the Java- and C#-like threading model, together with the threading APIs.

Besides threaded programming languages, help is also needed in implementing correct threaded programs. The timing dependencies among the threads in multi-threaded programs make correctness validation an immense challenge. However, race detection tools have existed in the research community for a long time, and lately some commercial tools like Visual Threads [7] and Assure [6] have appeared that address these issues. These tools are extremely good at finding bugs in threaded programs, but they suffer from long execution times and large memory-size footprints. Despite these issues, these tools are a very promising start for ensuring the correctness of multi-threaded programs and offer much hope for the future.

After building a correct multi-threaded program, a tool to help with the performance analysis of the program is also required. There are some very powerful tools today for analysis of sequential applications, like the VTune™ Performance Analyzer. However, the equivalent is missing for multi-threaded programs. Again, for OpenMP [3,5], good performance-analysis tools do exist in the research community and commercially. These tools rely heavily on the structured, synchronous OpenMP [3,5] programming model. The same tools for asynchronous threading APIs are non-existent, but seem necessary for the availability of large numbers of multi-threaded applications. Hyper-Threading Technology presents a unique challenge for performance-analysis tools, because the processors share resources and neither processor has all of the resources available at all times. In order to create a large pool of multi-threaded applications, it seems clear that effective tools are necessary. It is also clear that such tools are not yet

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

™ VTune is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

available today. To exploit Hyper-Threading Technology effectively, multi-threaded applications are necessary, and tools to create those are key.

CONCLUSION

High clock rates combined with efficient utilization of available processor resources can yield high application performance. As microprocessors have evolved from simple single-issue architectures to the more complex multiple-issue architectures, many more resources have become available to the microprocessor. The challenge now is effective utilization of the available resources. As processor clock frequencies increase relative to memory access speed, the processor spends more time waiting for memory accesses. This gap can be filled using extensions of techniques already in use, but the cost of these improvements is often greater than the relative gain. Hyper-Threading Technology uses the explicit parallel structure of a multi-threaded application to complement ILP and exploit otherwise wasted resources. Under carefully controlled conditions, such as the test kernels presented above, the speed-ups can be quite dramatic.

Real applications enjoy speed-ups that are more modest. We have shown that a range of existing, data-parallel, compute-intensive applications benefit from the presence of Hyper-Threading Technology with no source code changes. In this suite of multi-threaded applications, every application benefited from threading in the processor. Like assembly language tuning, Hyper-Threading Technology provides another tool in the application programmer's arsenal for extracting more performance from his or her computer system. We have shown that high values of clock cycles per instruction and per micro-op are indicative of opportunities for good speed-up.

While many existing multi-threaded applications can immediately benefit from this technology, the creation of additional multi-threaded applications is the key to fully realizing the value of Hyper-Threading Technology. Effective software engineering tools are necessary to lower the barriers to threading and accelerate its adoption into more applications.

Hyper-Threading Technology, as it appears in today's Intel® Xeon™ processors, is just the beginning. The fundamental ideas behind the technology apply equally well to larger numbers of threads sharing additional resources. Just as the number of distinct lines in a telephone network grows slowly relative to the number of customers served, Hyper-Threading Technology has the potential to modestly increase the number of resources in the processor core and serve a large numbers of threads. This combination has the potential to hide almost any latency and utilize the functional units very effectively.

APPENDIX: PERFORMANCE METRICS

Using the VTune™ Performance Analyzer, one can collect several execution metrics *in situ* as the application runs. While the Intel® Xeon™ processor contains a host of counters, we focused on the following set of raw values and derived ratios.

Clock Cycles

The number of clock cycles used by the application is a good substitute for the CPU time required to execute the application. For a single threaded run, the total clock cycles multiplied by the clock rate gives the total running time of the application. For a multithreaded application on a Hyper-Threading Technology-enabled processor, the process level measure of clock cycles is the sum of the clocks cycles for both threads.

Instructions Retired

When a program runs, the processor executes sequences of instructions, and when the execution of each instruction is completed, the instructions are retired. This metric reports the number of instructions that are retired during the execution of the program.

Clock Cycles Per Instruction Retired

CPI is the ratio of clock cycles to instructions retired. It is one measure of the processor's internal resource utilization. A high value indicates low resource utilization.

Micro-Operations Retired

Each instruction is further broken down into micro-operations by the processor. This metric reports the number of micro-operations retired during the execution of the program. This number is always greater than the number of instructions retired.

Clock Cycles Per Micro-Operations Retired

This derived metric is the ratio of retired micro-operations to clock cycles. Like CPI, it measures the processor's internal resource utilization. This is a finer measure of utilization than CPI because the execution engine operates directly upon micro-ops rather than instructions. The Xeon processor core is capable of retiring up to three micro-ops per cycle.

Percentage of Floating-Point Instructions

This metric measures the percentage of retired instructions that involve floating-point operations. To what extent the different functional units in the processor are busy can be determined by the instruction type mix because processors typically have multiple floating-point, integer, and load/store functional units. The percentage of floating-point instructions is an important indicator of whether the program is biased toward the use of a specific resource, potentially leaving other resources idle.

REFERENCES

[1] D. Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, Q1, 2002.

[2] Intel® Pentium® 4 Processor Optimization Reference Manual.

[3] http://developer.intel.com/software/products/compilers/

[4] http://www.openmp.org/

[5] http://developer.intel.com/software/products/kappro/

[6] http://developer.intel.com/software/products/assure/

[7] http://www.compaq.com/products/software/visualthreads/

AUTHORS' BIOGRAPHIES

William Magro manages the Intel Parallel Applications Center, which works with independent software vendors and enterprise developers to optimize their applications for parallel execution on multiple processor systems. He holds a B.Eng. degree in Applied and Engineering Physics from Cornell University and M.S. and Ph.D. degrees in Physics from the University of Illinois at Urbana-Champaign. His e-mail is bill.magro@intel.com.

Paul Petersen is a Principal Engineer at Intel's KAI Software Lab. He currently works with software development tools to simplify threaded application development. He has been involved in the creation of the OpenMP parallel programming language and tools for the performance and correctness evaluation of threaded applications. He holds a B.S. degree from the University of Nebraska and M.Sc. and Ph.D. degrees from the University of Illinois at Urbana-Champaign, all in Computer Science. His e-mail is paul.petersen@intel.com

Sanjiv Shah co-manages the compiler and tools groups at Intel's KAI Software Lab. He has worked on compilers for automatic parallelization and vectorization and on tools for software engineering of parallel applications. He has been extensively involved in the creation of the OpenMP specifications and serves on the OpenMP board of directors. Sanjiv holds a B.S. degree in Computer Science with a minor in Mathematics and an M.S. degree in Computer Science from the University of Michigan. His e-mail is sanjiv.shah@intel.com.