CMP Implementation in Systems Based on the Intel® Core™ Duo Processor

Avi Mendelson, Mobility Group, Intel Corporation
Julius Mandelblat, Mobility Group, Intel Corporation
Simcha Gochman, Mobility Group, Intel Corporation
Anat Shemer, Software Solutions Group, Intel Corporation
Rajshree Chabukswar, Software Solutions Group, Intel Corporation
Erik Niemeyer, Software Solutions Group, Intel Corporation
Arun Kumar, Software Solutions Group, Intel Corporation

Index words: Intel Core Duo, low power, CMP, multi-threading, software optimizations

ABSTRACT

The Intel® Core™ Duo processor is the first mobile processor to implement Chip Multi-Processing (CMP), also known as dual core-on-die. This first implementation was carefully chosen to deliver maximum performance for a given power. The performance improvement was achieved by enhancing the single-core micro-architecture, which results in better single-threaded performance, and by implementing CMP, which improves the performance of multi-threaded applications and parallel application processing. The focus of this paper is to introduce the reader to the CMP aspects of the Intel Core Duo processor. Since the Intel Core Duo processor was designed to be a mobile processor, we examine in detail the design considerations that had to be taken into account to achieve a balance between performance improvements and power savings, and we provide recommendations on optimizing the code developed for the Intel Core Duo processor so that future applications can take full advantage of the new design.

INTRODUCTION

The Intel Core Duo processor is the first mobile core to implement Core Multi-Processor (CMP) technology on one die. The implementation was carefully chosen to maximize performance, so it can be used as a general-purpose processor, and to minimize power consumption, in order to extend the battery life and have it fit in a large variety of thermal envelopes. The performance improvement was achieved by enhancing the micro-architecture, based on Pentium® M processor-based technology, of the single core, and by combining dual cores on the same die. In order to achieve the power consumption goal, we examined each micro-architectural decision with respect to its power/performance benefit. A general overview of the processor and its unique features can be found in this special issue of the *Intel Technology Journal* [1]. This paper focuses on the multi-core design and performance aspects of the processor, but for each of the decisions we describe here, we discuss how the power and thermal aspects were taken into account as part of our decision.

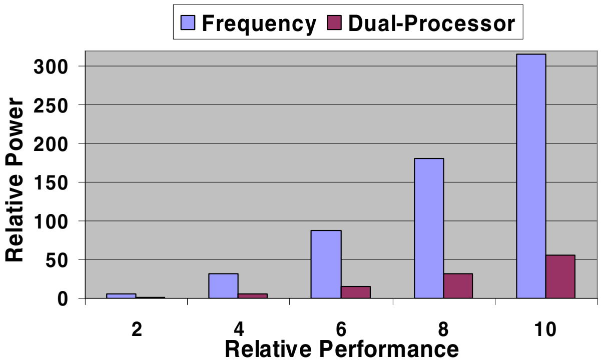

Figure 1: Theoretical power consumption for the same performance–single thread vs. dual thread

The first question one might ask is "why choose a CMP implementation for a mobile processor"? Figure 1 compares the power needed to complete the same amount of work, at the same execution time, assuming frequency scaling vs. using dual cores. In order to conduct the comparison, we assume a single-core processor that consumes 1 Watt at a given frequency and voltage, as a baseline. In order to double its performance one can either double both its frequency and voltage respectively, or he can double the number of cores (assuming perfect scaling of the software). As can be seen in the graph, it is clear that under these simple conditions a better solution will be to use parallel execution than to improve the speed of the processor to achieve the same performance. It is a known fact that the power (P) a processor consumes depends on the voltage and the frequency of the processor. In order to explain the graph of Figure 1, consider a more realistic relationship between the power the processor consumes and its voltage and frequency. The basic relationship is given by Equation 1:

Equation 1: $P = \propto CV^2F$

where P stands for power, C for capacitance, V for voltage, $\propto$ is the activity factor and F for frequency. For each frequency within the design space, there is a minimum V that can support it: we call the pair (Fi, Vi), a working point of the processor. As long as Vmin < Vi < Vmax, we can approximate that Fi is linearly dependent on Vi, and for every (Fj,Vj) such that Vj<Vmin, we set the Vj to be equal to Vmin. As a result, within the dynamic range of V, the power has a cube relation with the frequency, while below Vmin, the power has a linear dependency with the frequency. Figure 1 uses Equation 1 to estimate the power consumption of each configuration, but in order to represent more realistic scenario, we use an exponent of 2.5 rather than an exponent of 3 (cubical relation). Unfortunately, the exponential relation between the power and the frequency/voltage is only true as long as the working point is within the dynamic-scaling portion of the voltage and provided enough parallelism is available in the software being used.

Since Intel Core Duo technology is aimed at the general purpose mobile market, the design should be balanced between power consumption and performance. Thus, we used the following criteria to decide between different design alternatives:

(a) When the system runs single-threaded applications, its performance should be the same or better than previous-generation Pentium M processors (with the same cache size and at the same frequency).

(b) When the system runs multi-threaded applications, we wanted to maximize the performance of the execution and preserve power by introducing a new and efficient power and thermal control system.

On top of all the technical hurdles mentioned above, we also had to consider the complexity of different solutions, since our experience told us that complicated solutions consume much power. Thus, for any new feature, the performance improvement must be significant enough to compensate for its complexity.

The primary goal of this paper is to discuss the CMP implementation and resulting performance. We do not focus on the power-saving techniques in Intel Core Duo processors since reference [2] covers that aspect of the system. However, when we discuss our design alternatives and why we chose one solution over another, the reader will notice that power savings (both static and dynamic) were a major factor in our decisions.

The rest of this paper is organized as follows: in the next section we focus on the CMP implementation. This includes the tradeoffs we considered, why we chose the current implementation, and their power and performance impact. Next, we focus on performance measurements, and in last section we extend our discussion to cover software optimizations.

CMP IMPLEMENTATION AND DESIGN CONSIDERATIONS

The Intel Core Duo processor is a new member of the Pentium M processor family. Before discussing how CMP is implemented, let us describe the implementation of current processors in the Pentium M family.

Background – The Structure of the Pentium M Processor

All the Intel processors in the mobility family that preceded the Intel Core Duo processor were uni-processor, and therefore efficiently support only Single Threaded (ST) applications and had the same basic structure as presented in Figure 2.

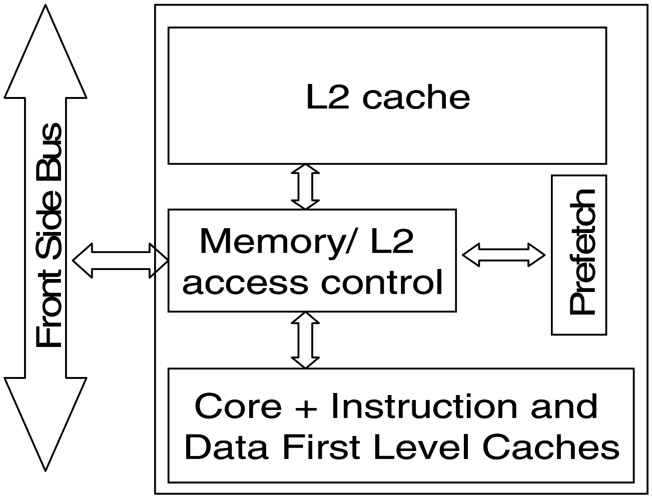
Front Side Bus
L2 cache
Memory/ L2 access control
Prefetch
Core + Instruction and Data First Level Caches

Figure 2: Structure of the memory cluster in the Intel Pentium M processor

Here, all the accesses to the L2 cache, as well as the accesses to the main memory and IO space, were under the supervision of a single control unit, shown in Figure 2 as *Memory/L2 access control* units (also called super-queue). Using this structure, cacheable requests from the core first looked for the data in the L2 cache and only if not found there (L2 miss), were they forwarded to the main memory via the front side bus (FSB). Uncacheable accesses could be directly sent to the main memory. The *Memory/L2 access control* unit also served as a central point for maintaining coherency within the core and with the external world. Pentium M processors support the MESI [3] coherence protocol that marks each cache line as Modifid, Exclusive, Shared, or Invalid.

In a nutshell, the MESI protocol attaches for each cache line a state that can be M-modified, E-exclusive, S-shared, or I-invalid. A line that is fetched, receives E, or S state depending on whether it exists in other processors in the system. A cache line gets the M state when a processor writes to it; if the line is not in E or M-state prior to writing it, the cache sends a Read-For-Ownership (RFO) request that ensures that the line exists in the L1 cache and is in the I state in all other processors on the bus (if any).

The *Memory/L2 access control* unit manipulates the coherency of each level of the caches independently. It contains a snoop control unit that receives snoop requests from the bus and performs the required operations on each cache (and internal buffers) in parallel. It also handles RFO requests and ensures the operation continues only after it guarantees that no other version on the cache line exists in any other cache in the system.

The CMP Implementation

At the early stages of the project, we considered three alternatives for CMP implementation, as illustrated with two structural alternatives in Figure 3. The first option (Figure 3a) was to put two single-core Pentium M processors, side by side, split the L2 cache among them, and communicate between the cores via the FSB or another fast interconnect.

Both other options called for a shared L2 (Figure 3b) with a different implementation of the coherence protocol; one option called for the same basic MESI table as in a single core but "adjusting it" to the new structures, while the second option called for a simple version of a directory-based protocol to improve the performance of the proposed structure.

The “simple” shared L2 implementation called for us to take advantage of the fact that the latency of the access to the L2 cache is significantly longer than the L1 access latency. This difference in latency enables us to check/update the status of the cache line in first level caches in parallel with L2 access. Therefore, this option increases the active power consumption (with respect to a single core) for snoop activities, but keeps the static power (leakage) the same as the single core, since no additional tables are used.

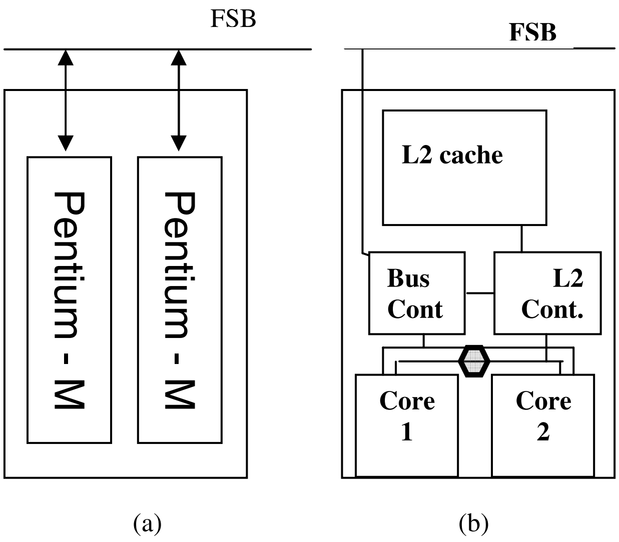

Figure 3: Implementation alternatives

The directory-based solution calls for extending the MESI information, as part of the L2 structure, and keeping information regarding the ownership on L2 cache lines. Here we assume that snoops are sent to the other core by the L2 controller, and only when needed. Thus, when a core accesses the line in the L2 cache, the cache controller knows if the line is shared with the other cache, and based on this information the cache control unit can optimize the number of snoops sent to the other L1. This technique reduces the active power due to reduced snoop activity, but increases the design complexity and the static power due to larger tag arrays.

Using the three criteria described in the introduction, we analyzed the performance and power and firstly eliminated the first option (3a). The reason for this was that it would reduce the performance of ST applications, since it provides only half of the cache size for each core. We also observed that the use of a split L2 cache could cause performance degradation when running multi-threaded (MT) applications with shared data, preventing effective data sharing between the threads, and requiring long latencies when moving data from one core to another. On top of that, it may reduce the performance of MT and parallel application processing since it could not dynamically partition the L2 cache

Deciding between the two implementations of the shared L2 cache was a tough task. The performance of the two options was very close and so we had to make our decision based on power efficiency. We decided to implement the simple solution and not the directory-based architecture due to its complexity. The directory-based solution was found to be less favorable since battery life mainly depends on static power consumption and less on dynamic power.

The general structure of the Intel Core Duo CMP implementation is given in Figure 3b. Comparing it with Figure 2 shows few structural changes: (1) The core and first-level caches structure is duplicated; (2) the traditional *memory and L2 control* unit (super-queue) is partitioned into two logical units: the *L2 controller* that handles all the requests to the L2 from the core and from the external bus (snoop requests) and a *bus control unit* that handles all the data and IO requests to and from the external bus; (3) in order to balance the requests to the L2 and memory, we added a new logical unit (represented by the hexagon) that aims to guarantee the fairness between the requests coming from different cores; and (4) we extended the prefetching unit to handle separately hardware prefetching by each core.

The new structure of the shared area allows us to enhance the performance while reducing power consumption. The new partitioned structure of the super-queue allows us to implement new power and performance optimizations, since the *L2-control unit* was designed to be relatively small, simple, and fast in order to reduce the latency to the L2 cache without increasing the power consumption. The *Bus Control Unit* was designed to be larger and more complicated, but since it was found to be more relaxed in timing, we could design it to have less leakage and even reduce its active power.

The power and performance results were measured on Intel Core Duo silicon and justified the CMP architecture we choose. We discuss this later in the paper.

THE PROTOCOL

From the external observer, the behavior of a CMP system should be looked at as the behavior of a dual package (DP) system. For that purpose, Intel Core Duo processor implements the same MESI protocol as in all other Pentium M processors.

In order to improve performance, we optimized the protocol for faster communication between the cores, particularly when the data exist in the L2 cache. A noticeable example of such a modification was done in order to allow the system to distinguish between a situation in which data are shared by the two CMP cores, but not with the rest of the world, and a situation in which the data are shared by one or more caches on the die as well as by an agent on the external bus (can be another processor). When a core issues an RFO, if the line is shared only by the other cache within the CMP die, we can resolve the RFO internally very fast, without going to the external bus at all. Only if the line is shared with another agent on the external bus do we need to issue the RFO externally.

For most Intel Core Duo systems, when only one package exists, this is a very important optimization. In the case of a multi-package system, the number of coherence messages over the external bus is smaller than in similar DP or MP systems, since much of the communication is being resolved internally. The number of required coherency messages is also much smaller than in the case of using a split cache (Figure 3a) which requires all the communication between the cores and split L2 caches to be done over the external bus.

PERFORMANCE MEASUREMENTS

This section describes the different measurements we did on an Intel Core Duo processor-based system. We start with basic measurements and then discuss the impact of programming models and optimizations on the overall power and performance of the system.

Basic Measurements

Two of the basic requirements we had from the system were (1) to keep, or improve the performance of ST applications, using the same frequency and L2 cache sizes, and (2) to take full advantage of parallel execution, when parallelism is available.

Figure 4 compares the performance of Pentium M processor and Intel Core Duo processors, using the same platform, running at the same frequency, and executing all the programs out of the SpecINT benchmark suite. As can be seen, on average, the performance of the Intel Core Duo and the Pentium M are the same.

Figure 5 compares the execution of all the programs out of the SpecFP performance benchmark. Here, some of the programs show a significant improvement over the Pentium M execution on the same platform. The main reason is the use of SSE3 new instructions by the compiler and a few other performance improvements described in [2].

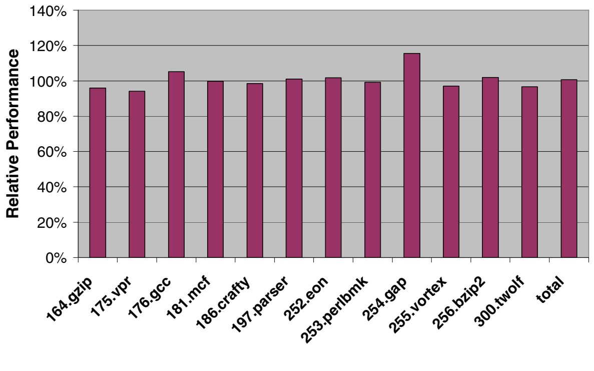

Figure 4: Single-threaded performance–SpecINT Core Duo vs. Pentium M (same cache, same platform)

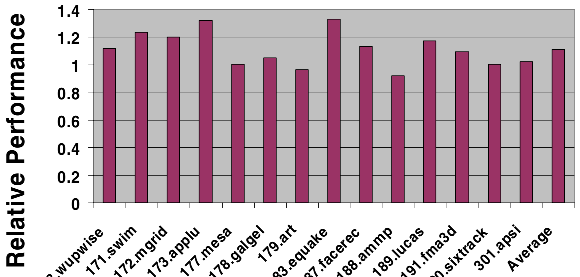

Figure 5: Single-threaded performance–SpecFP Core Duo vs. Pentium M (same cache, same platform)

After achieving the first goal of keeping the performance of an ST application at the same level (or better) as when run on a Pentium M processor, Figure 6 shows the speedup numbers that various MT applications can achieve.

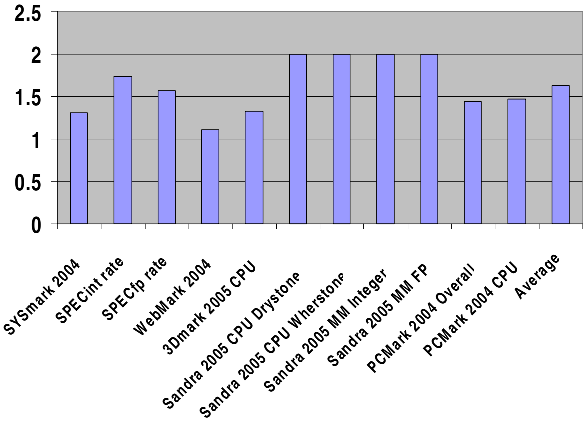

Figure 6: MT speedups

The speedup numbers presented here range from 1.2 to 2, which is the theoretical maximum that can be achieved by two cores. A closer look at the applications that reveal relatively low scalability shows that the main reason for that is lack of parallelism within the application. A few applications, such as SpecFP rate, suffer from high utilization of the bus. In these cases doing the same experiment but with a faster bus yields a better scalability.

Threading Models

When multi-threading an application, the choice of a threading model plays a key role in achieving maximum performance scaling. In this section, we discuss the effect of "data domain decomposition" and "functional domain decomposition" on the performance of an application.

Data domain decomposition usually results in a balanced threading model and is likely to produce a better scalable threading behavior when running the application on platforms with a higher number of processors. Functional domain decomposition is susceptible to imbalanced threads due to thread specific performance characteristics, and hence load-balancing issues need to be considered. A functional domain decomposed model is also likely to limit the scalability by any number of processors. One very important consideration with imbalanced threading behavior in applications is the operating system (OS) scheduling of threads on a CMP system (we illustrate this with an example in the sections below).

Applications With Balanced Threading Models

Applications studied here are CPU-intensive, consuming 95-100% of the CPU with the threads performing equal work and consuming equal processing resources. Here, we discuss the performance of these applications when they run in ST and MT modes. The performance data are measured in seconds.

The graph in Figure 7 indicates performance data for running ST and MT versions of the applications. Cryptography and Video Encoding applications have two MT implementations, and hence, results are indicated as MT1 and MT2. MT1 is implemented using a data domain decomposition methodology, and MT2 is implemented using functional domain decomposition.

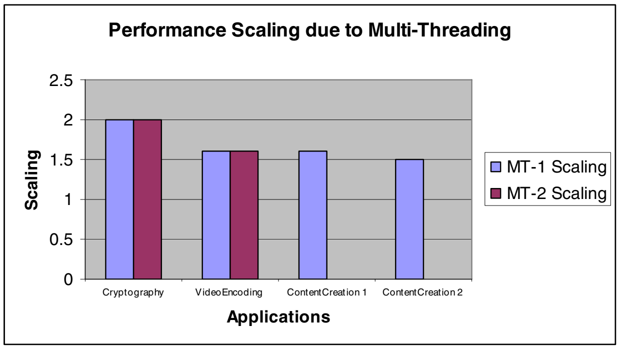

Figure 7: Balanced threading performance

As indicated in Figure 7, MT applications clearly demonstrate significant performance improvement over ST applications. Some of the applications have two different multi-threaded implementations. For example, MT-1, MT-2 versions of the Cryptography workload demonstrate a 2x performance improvement as compared to the ST version.

Applications with Imbalanced Threading Model

In this section, we examine the performance implications on an application with imbalanced threading models. For this study, a sample game physics engine was created (using Microsoft DirectX*). The sample application has two parts: 1) Physics Computation (collision detection and resolution for graphics objects), 2) Rendering (updated positions are drawn onto screen). The application was deliberately designed such that balanced and imbalanced threading could be studied for a CMP processor:

a. *Balanced*: For this implementation, graphical objects (and background imagery) were divided into two parts and each thread took care of the collision detection and resolution of its own set of objects.

b. *Imbalanced:* In this implementation, one thread was tasked with performing collision detection and resolution for the colliding objects while the other thread calculated the updated positions. The result was the desired goal of the first thread being more CPU intensive than the second thread.

With the two implementations, performance data in different power schemes, MaxPerf and Adaptive, are as shown below. Adaptive mode here refers to the power-saving scheme where the OS optimizes overall power consumption, by dynamically changing CPU frequency on demand, using Intel SpeedStep® technology (the GV3 technology). The MaxPerf mode refers to the power scheme where the processor is always running at the highest clock speed.

Let us discuss the first two data sets in Figure 8 for now.

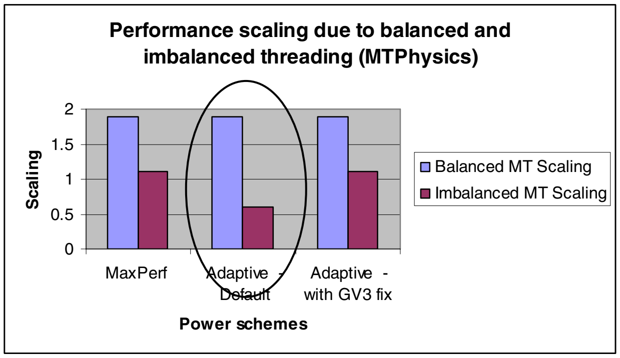

Figure 8: Imbalanced threading performance

The Imbalanced MT (Imbalanced-MT) implementation demonstrates a 2x performance degradation (0.6 scaling) when running in the Adaptive power scheme as compared to MaxPerf (indicated with the circle in Figure 8). In the Imbalanced-MT case, since one of the threads is doing a large amount of the work as compared to the other thread, the thread performing more work keeps migrating between the cores, making effective CPU utilization on the cores at ~50%. On systems running in "Adaptive" (portable/laptop) power mode, this thread migration causes the Windows* kernel power manager to incorrectly calculate the optimal target performance state for the processor. This reduces the operating frequency of both cores even when one of the cores is fully utilized in Adaptive mode and hence causes degradation in performance for the Imbalanced-MT case. Note that this issue may occur while running single-threaded workload as well. To address this issue, Microsoft provided a hot-fix (KB896256) to change the kernel power manager to track CPU utilization across the entire package, rather than the individual cores and hence calculate the optimum frequency for applications.

The third set in Figure 8 indicates data with the kernel hot-fix. In this case, the Imbalanced-MT implementation in Adaptive mode shows expected performance scaling as of MaxPerf mode. With this fix, both cores run at optimum frequency, not causing any degradation in Adaptive (PL) mode.

COMPARING SPLIT CACHE WITH SHARED CACHE

Recently, different architectures use a split last-level cache in order to achieve a fast time-to-market of a dual-core system. Clear downsides of this solution are as follows:

1. Cache coherent-related events that need to be served over the FSB, such as RFO or invalidation signals, greatly impact performance and power.

2. An ST application cannot take full advantage of the entire cache.

The hard partitioned cache may have one significant benefit over the unified cache; that is, it may prevent one application from significantly reducing the amount of cache memory available to an application running on the other core. Thus, in this section we compare two systems: one uses a split L2 cache and the other uses a unified model. In order to make the comparison fair, we present speedup numbers and not absolute numbers.

A sample physics engine game is created (using Microsoft DirectX) to perform this study. The application is MT using data domain decomposition. The threads are synchronized before rendering the updates on the screen. Since the dependency among the threads is very minimal, we expected to achieve ~2.0x performance improvement with the MT version as compared to the ST version.

The split L2 cache indicated approximately a 1.68x performance improvement due to MT. Running the same application on the Intel Core Duo processor-based system demonstrates ~1.90x scaling as per our expectations.

The root cause of the difference in the scaling is due to the shared L2 cache on the Intel Core Duo system. The sample application under study is designed in a way that both threads work on data from a shared data structure. Hence, on the system with the split L2 cache, to get access to the data modified by one processor, the second processor needs to go to main memory, which results in many L2 cache misses. Since the Intel Core Duo system has a unified L2 cache, a penalty of cache miss and access to the main memory is avoided, as the data modified by one core can be made available to the other core immediately.

OPTIMIZATION OPPORTUNITIES FOR INTEL® CORE™ DUO PROCESSOR

Like any parallel system, the performance and the power of the Intel Core Duo processor may be sensitive to the memory access patterns. In this section we review three optimizations that are very important for getting the best out of the system.

Efficient Use of the Shared L2 Cache

Sharing data between two threads on the Intel Core Duo processor is fastest when done through the L2 cache. This section examines several scenarios for sharing.

One scenario is when one thread brings the data from memory, and the other thread later uses this data directly from the L2 cache. If the single-threaded workload needs to bring the same data several times from memory but the multi-threaded version is carefully designed to use the same data by the two threads simultaneously, the MT version gains performance by bringing the data less times from the memory to the cache hierarchy. Such a design can help applications with a larger than L2 cache data set and even achieve higher than 2x performance improvement.

Another scenario is when one thread generates the data and the other thread consumes it. A couple of variations of this scenario are possible and are further explained in the "Producer Consumer Models," Section 5.3 of the *Intel® Core™ Duo Processor Optimization Guide*[4]. Briefly, they are the "Delay" approach and "Symmetric" approach. Below is an example of the expected speedup when the producer-consumer model is run on an Intel Core Duo processor vs. a Dual Core Intel® Xeon® processor vs. an Intel Pentium 4 processor with Hyper-Threading Technology[1] (W = Write, R = Read, xxK = buffer size).

Not only do these data show the benefit of avoiding the bus/memory latency, they also demonstrate how varying multi-processor implementations behave in both code affinity (functional) decomposition and data affinity (data) decomposition threading models. If the produced/consumed data set size is bigger than the L1 data cache size, yet smaller than the L2 cache size, data decomposition and functional decomposition yield similar performance (assuming the functional decomposition implementation is well balanced), and the best performance that can be achieved for data sharing.

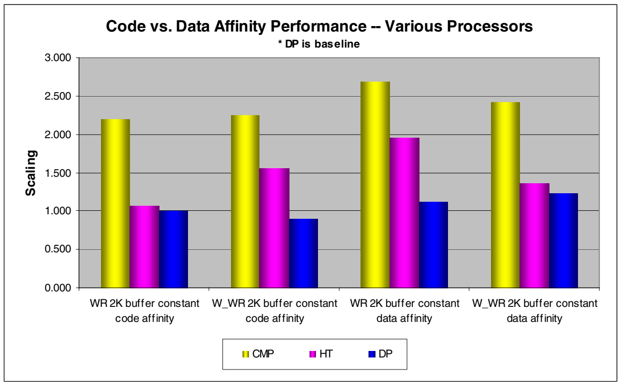

Figure 9: Code vs. data affinity performance on various processors

False Sharing Can Reduce Performance

False sharing happens when two or more threads access different address ranges on the same cache line simultaneously. This causes the cache line to be in the first level cache of the two cores.

False sharing causes a severe performance penalty if one or more of the threads writes to the shared cache line. This causes invalidation of the cache line at the first-level cache of the other core. As a result, the next time that the other core accesses the cache line in question it will have to transfer it from the core that wrote it earlier through the bus, thereby incurring a major latency penalty.

Below is an example of code that has false sharing when executed by several threads simultaneously.

```
int counter[THREAD_NUM];
int inc_counter ()
{
    counter[my_tid]++;
    return counter[my_tid];
}
```

Table 1 lists the penalties that an application can suffer if it uses false sharing intensively on an Intel Core Duo system. In order to avoid such an unnecessary overhead, the programmer needs to avoid false sharing, and in particular, needs to make sure it does not occur unintentionally in the following cases:

[1] Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a HT Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See www.intel.com/products/ht/Hyperthreading_more.htm for additional information.

- Global data variables and static data variables that are placed in the same cache line but are written by different threads.
- Objects allocated dynamically by different threads can accidentally share cache lines.

Table 1: False sharing penalties

Case	Data location	Latency (cycles/nsec)
L1 to L1 Cache	L1 Cache	14 core cycles + 5.5 bus cycles
Through L2 Cache	L2 Cache	14 core cycles
Through Memory	Main memory	14 core cycles + 5.5 bus cycles + ~40-80 nsec depending on FSB and DDR freq.

Optimize Bus Access Between the Cores to Maximize the Bus Bandwidth

Be careful when parallelizing code sections that use data sets exceeding the second-level cache and/or bus bandwidth. If only one of the threads is using the second-level cache and/or bus, then it is expected to get the maximum possible speedup when the other thread running on the other core does not interrupt its progress. However, if the two threads use the second-level cache there may be performance degradation if one of the following conditions is true:

- Their combined data set is greater than the second-level cache size.
- Their combined bus usage is greater than bus capacity.
- They both have extensive access to the same set in the second-level cache, and at least one of the threads writes to this cache line.

To avoid these, we recommend that you investigate parallelism schemes in which only one of the threads accesses the second-level cache at a time, or that the level of using the second-level cache and the bus does not exceed their limits. This concept is explained further in Section 5.3.5 of the *Intel® Core™ Duo Processor Optimization Guide*.

CONCLUSION AND REMARKS

The full performance potential of the Intel® Centrino® Duo mobile technology architecture can be realized by efficiently multi-threading applications, using the methods detailed in this paper. The use of balanced threading techniques is likely to provide optimal performance improvements on CMP. Multi-tasking scenarios, one of the common usage scenarios, provide a richer user experience on the Intel Centrino Duo mobile technology system. The shared cache structure is likely to showcase better performance scaling for MT applications when the threads work on shared data sets. Avoid using false sharing which impacts performance scaling. The optimization guide mentioned earlier provides a detailed explanation of these and other performance optimization techniques.

REFERENCES

[1] Gochman et Al., “Introduction to Intel® Core™ Duo Processor Architecture,” in *Intel Technology Journal, Volume 10, Issue 2, 2006.*

[2] Naveh et al., “Power and Thermal Control in the Intel® Core™ Duo Processor,” in *Intel Technology Journal, Volume 10, Issue 2, 2006.*

[3] *IA-32 Intel® Architecture Software Developer’s Manual Volume 3A*: *System Programming Guide, Part 1*, in ftp://download.intel.com/design/Pentium4/manuals/25366819.pdf

[4] IA-32 Intel® Architecture Optimization Reference Manual, in http://www.intel.com/design/Pentium4/manuals/248966.htm

AUTHORS’ BIOGRAPHIES

Avi Mendelson is a principal engineer in Intel’s Mobile Platform Group in Haifa, Israel, and adjunct professor in the CS and EE departments, Technion, Israel Institute of Technology. He received his B.Sc. and M.S.c degrees from the Technion, Israel Institute of Technology and his Ph.D from the University of Massachusetts Amherst. Avi has been with Intel for 7 years. He started as senior researcher in Intel Labs, later he moved to the Microprocessor group where he serves as the CMP architect of Intel Core Duo processor. Avi’s work and research interests are in computer architecture, low-power design, parallel systems, OS related issues and virtualization. His e-mail address is avi.mendelson at intel.com.

Julius Mandelblat is a principal engineer in Intel’s Mobile Platform Group in Haifa, Israel. He received his B.Sc. and M.Sc. degrees from Transport Engineers Institute in Moscow (USSR). Julius joined Intel 16 years ago. He worked on many of the processors that have been developed by Israel Design Center during this period. Julius worked as micro-architect and senior design leader for the CMP implementation of the Core Duo processor. His e-mail address is julius.mandelblat at intel.com.

Simcha Gochman is a senior principal engineer with Intel's Mobile Platform Group in Haifa, Israel. Simcha has been with Intel for 21 years. Lately he was leading the microarchitecture development of the Pentium M processors Banias and Dothan and of the Core Duo processor code name Yonah. Earlier in Intel he led the microarchitecture definition of the Pentium Processor with MMX™ technology and was involved with the design of the 80860 processor and the 80387 numeric coprocessor. Simcha received his M.Sc. degree from the Technion, Israel Institute of Technology in 1984. His e-mail address is simcha.gochmana at intel.com.

Anat Shemer is a senior software engineer working on mobile software enabling in the Software Solutions Group. Anat has been with Intel for 16 years. She worked on performance analysis of multi-threaded workloads supporting the evaluation of Intel Core Duo micro-architecture performance. Earlier at Intel she worked on binary tools and performance simulation tools that supported Intel future micro-architectures development. Anat received here M.Sc. degree from the Technion, Israel Institute of Technology in 1989. Her e-mail address is anat.shemer at intel.com.

Rajshree Chabukswar is a software engineer working on client enabling in the Software Solutions Group that enables client platforms through software optimizations focusing on multi-threading and mobile enabling. Prior to working at Intel, she obtained a Masters degree in Computer Engineering from Syracuse University, NY. Her e-mail address is rajshree.a.chabukswar at intel.com.

Erik Niemeyer is a senior software engineer working for Intel's Software and Solutions Group. His current assignment is in New Mexico with the Mobile Enabling Client Team working on IBM-Lotus Notes* to help improve performance and reduce power consumption of the upcoming R8 release. Erik has been with Intel for 6 years. Before Intel, Erik was a systems integrator/programmer with the Federal Government for 12 years and worked on enterprise-level application performance tuning. Prior to that Erik earned his B. Sc. from the University of New Mexico. His e-mail address is erik.a.niemeyer at intel.com.

Arun Kumar is an engineering manager in Intel's Software and Solutions Group in Dupont, Washington. He received his B.Tech degree from the Indian Institute of Technology, Kanpur, and his Masters and PhD degrees from the University of Minnesota, Minneapolis, where his research was in the area of image processing and computer vision. Currently, his team works on client platforms based on Pentium-M, Pentium-D and Core Duo processor architectures with special focus on CMP, application software performance and platform power consumption. His e-mail address is arun.kumar at intel.com.