



Whitepaper

# Intel® 64 Architecture Processor Topology Enumeration

**Executive Summary:** Processor topology information is important for a number of processor-resource management practices, ranging from task/thread scheduling, licensing policy enforcement, affinity control/migration, etc. Topology information of the cache hierarchy can be important to optimizing software performance. This white paper covers topology enumeration algorithm for single-socket to multiple-socket platforms using Intel 64 and IA-32 processors. The topology enumeration algorithms (both processor and cache) using initial APIC ID has been extended to use x2APIC ID, the latter mechanism is required for future platforms supporting more than 256 logical processors in a coherent domain

Hardware multithreading in microprocessors has proliferated in recent years. The majority of Intel® architecture processors shipping today provide one or more forms of hardware multi-threading support (multicore and/or simultaneous multithreading (SMT), the latter introduced as HyperThreading Technology in 2002). From a processor hardware perspective, the physical package of an Intel 64 processor can support SMT and multi-core. Consequently, a physical processor is effectively a hierarchically ordered collection of logical processors with some forms of shared system resources (for example, memory, bus/system links, caches) From a platform hardware perspective, hardware multithreading that exists in a multi-processor system may consist of two or more physical processors organized in either uniform or non-uniform configuration with respect to the memory subsystem.

Application programming using hardware multithreading features must follow the programming models and software constructs provided by the underlying operating system. For example, an OS scheduler generally assigns a software task from a queue using hardware resource at the granularity of a logical processor; an OS may define its own data structure and provide services to applications that allows them to customize the assignment between task and logical processor via an affinity construct for multithreaded applications The OS and the software stack underneath an application (the BIOS, the OS loader) also play significant roles in bringing up the hardware multi-threading features and configuring the software constructs defined by the OS.

The CPUID instruction in Intel 64 architecture defines a rich set of information to assist BIOS, OS, and applications to query processor topology that are needed for efficient operation by each respective member of the software stack. Generally, the BIOS needs to gather topology information of a physical processor, determine how many physical processors are present in the system; prepare the necessary software constructs related to system topology, and pass along the system topology information to the next layer of the software stack that takes over control of the system. The OS and the application layers have a wide range of uses for topology information. This document covers several common software usages by OS and applications for using CPUID to analyze processor topology in a single-processor or multi-processor system.

The primary software usage of processor topology enumeration deals with querying and identifying the hierarchical relationship of logical processor, processor cores, and physical packages in a single-processor or multi-processor system. We'll refer to this usage as system topology enumeration. System topology enumeration may be needed by OS or certain applications to implement licensing policy based on physical processors. It is used by OS to implement efficient task-scheduling, minimize thread migration, configure application thread management interfaces, and configure memory allocation services appropriate to the processor/memory topology. Multithreaded applications need system topology information to determine optimal thread binding, manage memory allocation for optimal locality, and improve performance scaling in multi-processor systems.

Intel 64 processors fulfill system topology enumeration requirements:

1. Each logical processor in an Intel 64 or IA-32 platform supporting coherent memory is assigned a unique ID (APIC ID) within the coherent domain. A multi-node cluster installation may employ vendor-specific BIOS that preserve the APIC IDs assigned (during processor reset) within each coherent domain, extend with node IDs to form a superset of unique IDs within the clustered system. This document will only cover the CPUID interfaces providing unique IDs within a coherent domain.

2. The values of unique IDs assigned within a coherent Intel 64 or IA-32 platform conform to an algorithm based on bit-field decomposition of the APIC ID into three sub-fields. The three sets of sub-fields correspond to three hierarchical levels defined as “SMT”, “processor core” (or “core”), and “physical package” (or “package”). This allows each hierarchical level to be mapped to a sub-field (a sub ID) within the APIC ID.

Conceptually, a topology enumeration algorithm is simply to extract the sub ID corresponding to a given hierarchical level from the APIC ID, based on deriving two parameters that defines the subset of bits within an APIC ID. The relevant parameters are: (a) the width of a mask that can be used to mask off unneeded bits in the APIC ID, (b) an offset relative to bit 0 of the APIC ID.

The “SMT” level corresponds to the innermost constituent of the processor topology. So it is located in the least significant portion of the APIC ID. If the corresponding width for “SMT” is 0, it implies there is only 1 logical processor within the next outer level of the hierarchy. For example, Intel® Core™2 Duo Processors generally produce a “SMT\_Mask\_Width” of 0. If the corresponding width is 1 bit wide, there could be two logical processors within the next outer level of the hierarchy.

If the corresponding width for “core” is 0, it implies there is only 1 processor core within a physical processor. If the corresponding width for “core” is 1 bit wide, there could be two processor cores within a physical processor.

Note, the values of APIC ID that are assigned across all logical processors in the system need not be contiguous. But the subsets of bit fields corresponding to three hierarchical levels are contiguous at bit boundary. Due to this requirement, the bit offset of the mask to extract a given Sub ID can be derived from the “mask width” of the inner hierarchical levels.

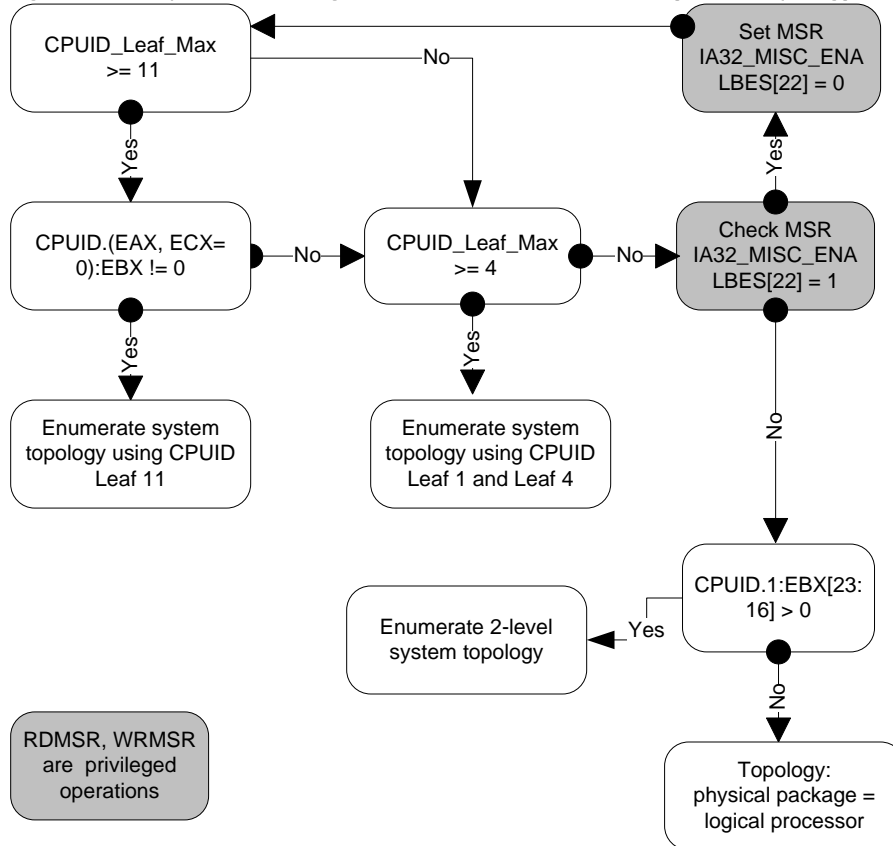
---

## ***Unique APIC ID in a Multi-Processor System***

Although legacy IA-32 multi-processor systems assigns unique APIC IDs for each logical processors in the system, the programming interfaces have evolved several times in the past. For Intel Pentium Pro processors and Pentium III Xeon processors, APIC IDs are accessible only from local APIC registers (Local APIC registers use memory-mapped IO interfaces and are managed by OS). In the first generation of Intel Pentium 4 and Intel Xeon processor processors (2000, 2001), CPUID instruction provided information on the initial APIC ID that is assigned during processor reset. The CPUID instruction in the first generation of Intel Xeon MP processor and Intel Pentium 4 processor supporting Hyper-Threading Technology (2002) provided additional information that allows software to decompose initial APIC IDs into a two-level topology enumeration. With the introduction of dual-core Intel 64 processors in 2005, system topology enumeration using CPUID evolved into a three-level algorithm on the 8-bit wide initial APIC IDs. Future Intel 64 platforms may be capable of supporting a large number of logical processors that exceed the capacity of the 8-bit initial APIC ID field. The x2APIC extension in Intel 64 architecture defines a 32-bit x2APIC ID, the CPUID instruction in future Intel 64 processors will allow software to enumerate system topology using x2APIC IDs. The extended topology enumeration leaf of CPUID (leaf 11) is the preferred interface for system topology enumeration for future Intel 64 processor.

The CPUID instruction in future Intel 64 processors may support leaf 11 independent of x2APIC hardware. For many future Intel 64 platforms, system topology enumeration may be performed using either CPUID leaf 11 or legacy initial APIC ID (via CPUID leaf 1 and leaf 4). Figure 1 shows an example of how software can choose which CPUID leaf information to use for system topology enumeration.

**Figure 1 Example of Choosing CPUID Leaf Information for System Topology Enumeration**



The maximum value of supported CPUID leaf can be determined by setting EAX = 0, execute CPUID and examine the returned value in EAX, i.e. CPUID.0:EAX. If CPUID.0:EAX >= 11, software can determine whether CPUID leaf 11 exists by setting EAX=11, ECX=0, execute CPUID to examine the non-zero value returned in EBX, i.e. CPUID. (EAX=11, ECX=0):EBX != 0.

Fully functional hardware multithreading requires full-reporting of CPUID leaves.

If software observes that CPUID.0:EAX < 4 on a newer Intel 64 or IA-32 processor (newer than 2004), it should examine the MSR IA32\_MISC\_ENABLES[bit 22].

If IA32\_MISC\_ENABLES[bit 22] was set to 1 (by BIOS or other means), the user can restore CPUID leaf function full reporting by having IA32\_MISC\_ENABLES[bit 22] set to '0' (Modify BIOS CMOS setting or use WRMSR).

For older IA-32 processors that support only two-level topology, the three-level system topology enumeration algorithm (using CPUID leaf 1 and leaf 4) is fully compatible with older processors supporting two-level topology (SMT and physical package). For processors that report CPUID.1:EBX[23:16] as reserved (i.e. 0), the processor supports only one level of topology.

Table A-1 shows a code example of dealing with CPUID leaf functions across three categories of processor hardware.

## System Topology Enumeration Using CPUID Extended Topology Leaf

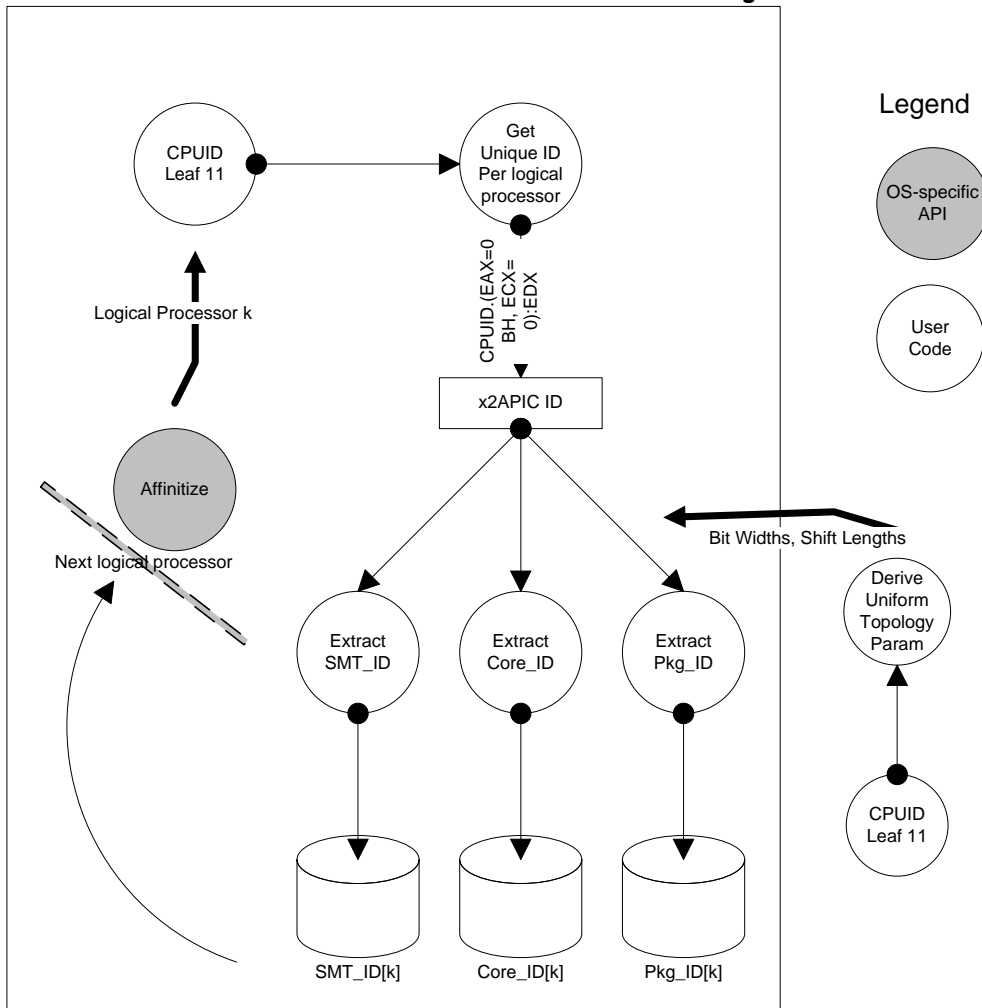
The algorithm of system topology enumeration can be summarized as three phase of operation:

- Derive "mask width" constants that will be used to extract each Sub IDs.
- Gather the unique APIC IDs of each logical processor in the system, and extract/decompose each APIC ID into three sets of Sub IDs.
- Analyze the relationship of hierarchical Sub IDs to establish mapping tables between OS's thread management services according to three hierarchical levels of processor topology.

Table A-2 shows an example of the basic structure of the three phases of system wide topology as applied to processor topology and cache topology.

Figure 2 outlines the procedures of querying CPUID leaf 11 for the x2APIC ID and extracting sub IDs corresponding to the "SMT", "Core", "physical package" levels of the hierarchy.

Figure 2 Procedures to Extract Sub IDs from the x2APIC ID of Each Logical Processor



Table

A-3 lists a data structure that holds the APIC ID, various sub IDs, and a hierarchical set of ordinal numbering scheme to enumerate each entity in the processor topology and/or cache topology of a system.

System topology enumeration at the application level using CUID involves executing CUID instruction on each logical processor in the system. This implies context switching using services provided by an OS. On-demand context switching by user code generally relies on a thread affinity management API provided by the OS. The capability and limitation of thread affinity API by different OS vary. For example, in some OS, the thread affinity API has a limit of 32 or 64 logical processors. It is expected that enhancement to thread affinity API to manage larger number of logical processor will be available in future versions.

### System Topology Enumeration Using CUID Leaf 1 and Leaf 4

Figure 3 outlines the procedures of querying initial APIC ID via CUID leaf 1 and extracting sub IDs corresponding to the SMT, Core, physical package levels of the hierarchy using CUID leaf 1 and leaf 4. Extraction of sub ID from initial APIC ID is based on querying CUID leaf 1 and 4 to derive the bit widths of three select masks (SMT mask, Core mask, Pkg mask) that make up the 8-bit initial APIC ID field. The select masks allow software to extract the sub IDs corresponding to "SMT", "core", "package" from the initial APIC ID of each logical processor.

Figure 3 Procedures to Extract Sub IDs from the Initial APIC ID of Each Logical Processor

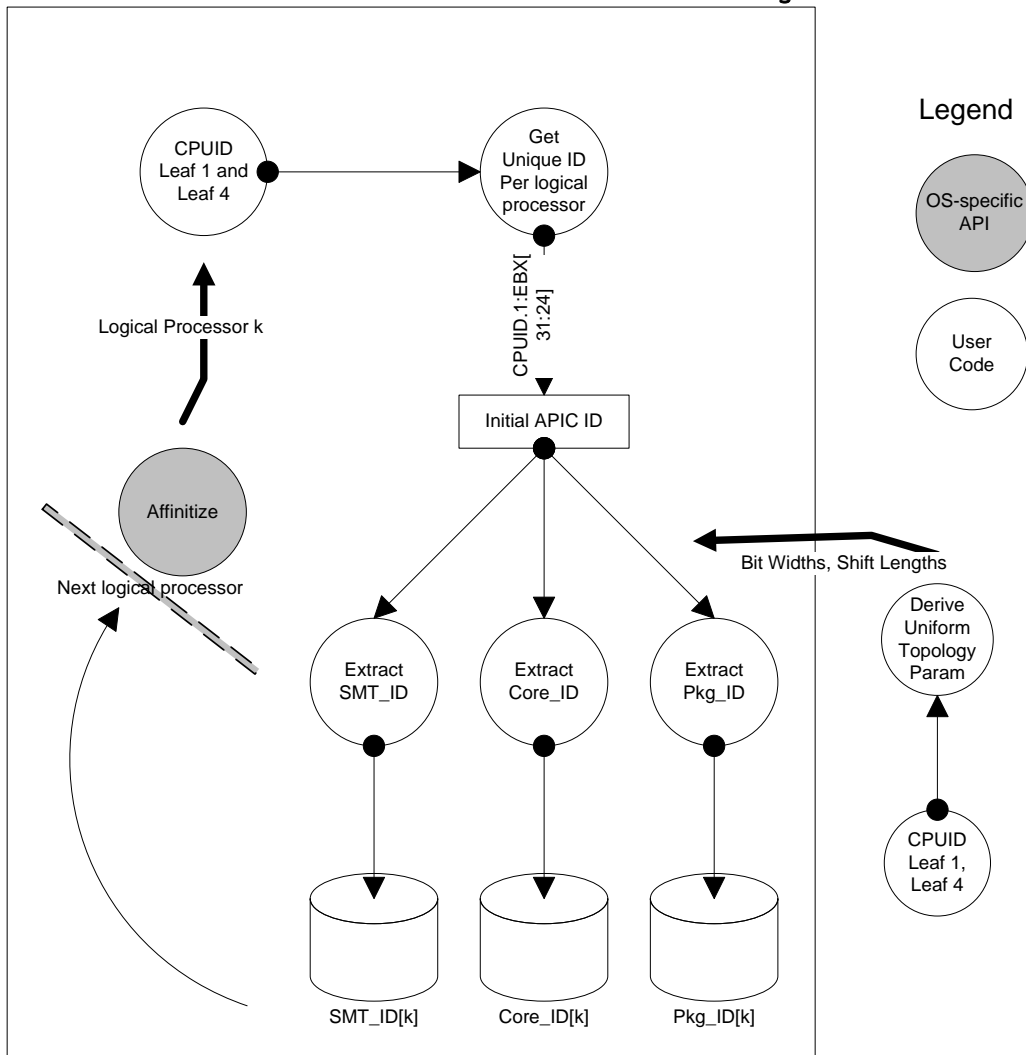


Table A-4 shows an example of querying the APID ID for each logical processor in the system and parsing each APIC ID into respective sub IDs for later analysis of the topological makeup.

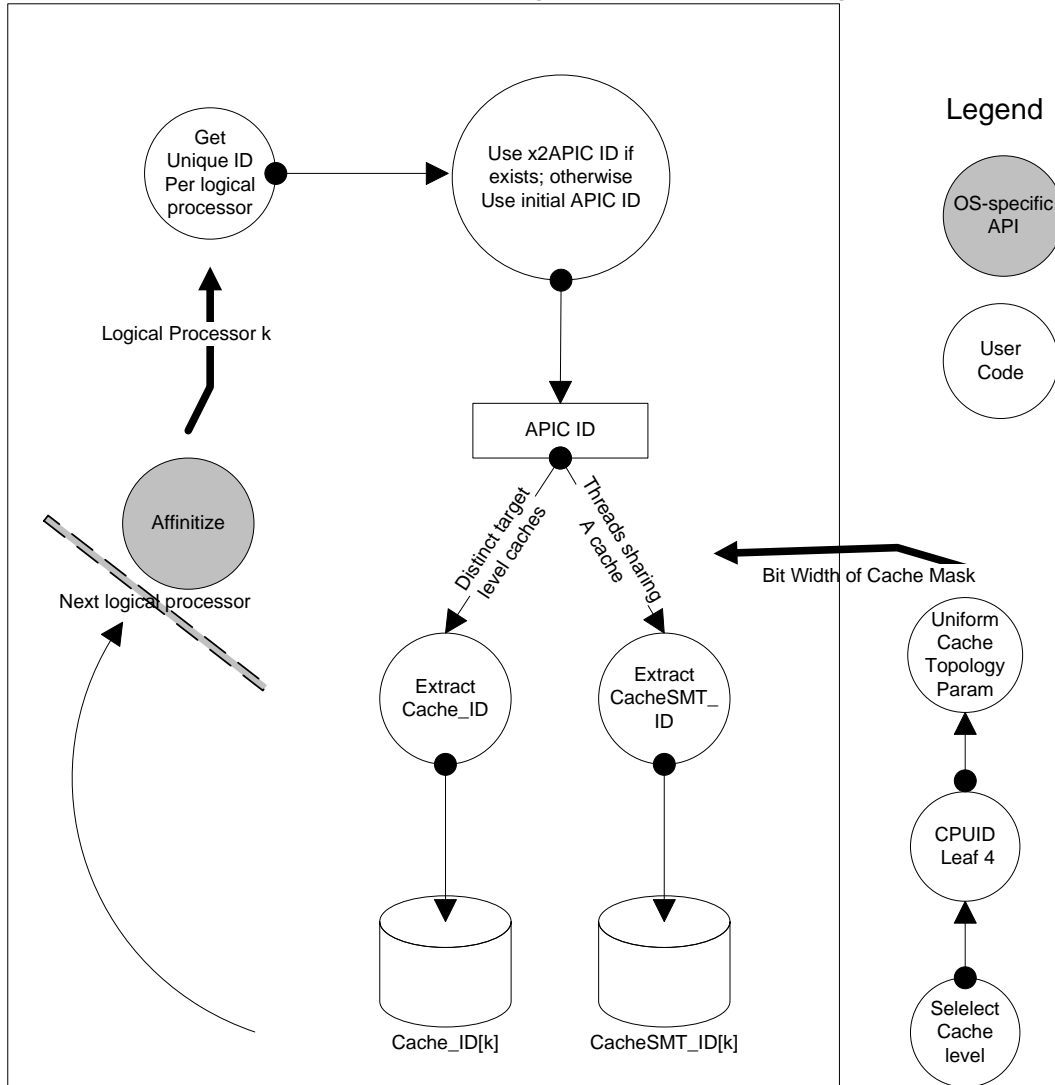
Table A-5 lists support routines to extract various sub IDs from each APIC ID of the logical processor that we have bound the current execution context.

Table A-6 shows OS-specific wrapper functions.

## Cache Topology Enumeration

The physical package of an Intel 64 processor has a hierarchy of cache. A given level of the cache hierarchy may be shared by one or more logical processors. Some software may wish to optimize performance by taking advantage of the shared cache of a particular level of the cache hierarchy. Performance tuning using cache topology can be accomplished by combining the system topology information with the addition of cache topology information. Figure 4 outlines the procedures of decomposition of sub IDs to enumerate logical processors sharing the target cache level and enumerating the target level caches visible in the system. The Cache\_ID can be extracted from the x2APIC ID for processors that reports 32-bit x2APIC ID or from the initial APIC ID for processors that do not report x2APIC ID. The array of "Cache\_ID" can be used to enumerate different caches in conjunction with other sub ID derived from the processor topology to implement code-tuning techniques.

Figure 4 Procedures to Extract Cache\_ID of the Target Cache Level of Each Logical Processor



The three-level sub IDs, SMT\_ID[k], Core\_ID[k], Pkg\_ID[k], k = 0, .., N-1 can be used by software in a number of application-specific ways. Some of the more common usages include:

1. Determine the number of physical processors to implement a per-package licensing policy. Each unique value in the Pkg\_ID[] array represents a physical processor.
2. A thread-binding strategy may choose to favor binding each new task to a separate core in the system. This may require the software to know the relationships between the affinity mask of each logical processor relative to each distinct processor core.
3. An MP-scaling optimization strategy may wish to partition its data working set according to the size of the large last-level cache and allow multiple threads to process the data tile residing in each last level cache. This will require software to manage the affinity masks and thread binding relative to each Cache\_ID and APIC ID in the system.

## Data Processing of Sub IDs of the Topology

Each hierarchy of the sub IDs represents a subset of the APIC ID (either x2APIC ID or initial APIC ID). It allows software to address each distinct entity within the parent hierarchical level. For processor topology enumeration:

- SMT\_ID: each unique SMT\_ID allows software to distinguish different logical processors within a processor core,
- Core\_ID: each unique Core\_ID allows software to distinguish different processor cores within a physical package,
- Pkg\_ID: each unique Pkg\_ID allows software to distinguish different physical packages in a multi-processor system.

For cache topology enumeration:

- CacheSMT\_ID: each unique CacheSMT\_ID allows software to distinguish different logical processors sharing the same target cache level.
- Cache\_ID: each unique Cache\_ID allows software to distinguish different target level cache in the system.

The extraction of sub IDs from the APIC ID makes use of constant parameters that are derived from CPUID instruction. From platform hardware perspective, Intel 64 and IA-32 multi-processor system require each physical processor support the same hardware multi-threading capabilities. Therefore, system topology enumeration can execute the relevant CPUID leaf functions on one logical processor to derive system-wide sub-ID extraction parameters. But the APIC IDs must be queried by executing CPUID instruction on each logical processor in the system.

### Sub ID Extraction Parameters for x2APIC ID

Extraction of sub ID from x2APIC ID is based on querying the value of CPUID.(EAX=11, ECX=n):EAX[4:0] for a valid sub leaf index n to obtain the bit width parameter to derive an extraction mask while x2APIC ID is queried by CPUID.(EAX=1,ECX=0):EDX[31:0]. The extraction mask allows software to extract a subset of bits from the x2APIC ID as a sub ID for the respective level of the hierarchy. In order to enumerate the sub IDs, increase sub leaf index (ECX=n) by 1 until CPUID.(ECX=11,ECX=n).EBX[15:0] == 0

- **SMT\_ID:** CPUID.(EAX=11, ECX=0):EAX[4:0] provides the width parameter to derive a SMT select mask to extract the SMT\_IDs of logical processors within the same processor core. The sub leaf index (ECX=0) is architecturally defined and associated with the "SMT" level type (CPUID.(EAX=11, ECX=0):ECX[15:8] == 1)
  - SMT\_Mask\_Width = CPUID.(EAX=11, ECX=0):EAX[4:0] if CPUID.(EAX=11, ECX=0):ECX[15:8] is 1
  - SMT\_Select\_Mask =  $\sim((-1) \ll \text{SMT\_Mask\_Width})$
  - SMT\_ID = x2APIC\_ID & SMT\_Select\_Mask
- **Core\_ID:** The level type associated with the sub leaf index (ECX=1) may vary across processors with different hardware multithreading capabilities. If CPUID.(EAX=11, ECX=1):ECX[15:8] is 2, it is associated with "processor core" level type. Then, CPUID.(EAX=11,ECX=1):EAX[4:0] provides the width parameter to derive a select mask of all logical processors within the same physical package. The "processor core" includes "SMT" in this case, and enumerating different cores in the package can be done by zeroing out the SMT portion of the inclusive mask derived from this.

- CorePlus\_Mask\_Width = CPUID.(EAX=11,ECX=1):EAX[4:0] if CPUID.(EAX=11, ECX=1):ECX[15:8] is 2
- CoreOnly\_Select\_Mask =  $(\sim(-1) \ll \text{CorePlus\_Mask\_Width}) \wedge \text{SMT\_Select\_Mask}$ .
- Core\_ID =  $(\text{x2APIC\_ID} \& \text{CoreOnly\_Select\_Mask}) \gg \text{SMT\_Mask\_Width}$
- **Pkg\_ID**: Within a coherent domain of the three-level topology, the upper bits of the APIC\_ID (except the lower “CorePlus\_Mask\_Width” bits) can enumerate different physical packages in the system. In a clustered installation, software may need to consult vendor specific documentation to distinguish the topology of how many physical packages are organized within a given node.
  - Pkg\_Select\_Mask =  $(-1) \ll \text{CorePlus\_Mask\_Width}$
  - Pkg\_ID =  $(\text{x2APIC\_ID} \& \text{Pkg\_Select\_Mask}) \gg \text{CorePlus\_Mask\_Width}$

An example of deriving the extraction parameters for x2APIC ID can be found in the support function “CPUTopologyLeafBConstants()” in the Appendix.

Table A-7 lists the support function to derive bitmask extraction parameters from CPUID leaf 0BH to extract sub IDs from x2APIC ID.

## Sub ID Extraction Parameters for Initial APIC ID

Topological sub ID extraction from an INITIAL\_APIC\_ID (CPUID.1:EBX[31:24]) uses parameters derived from CPUID.1:EBX[23:16] and CPUID.(EAX=04H, ECX=0):EAX[31:26]. CPUID.1:EBX[23:16] represents the maximum number of addressable IDs (initial APIC ID) that can be assigned to logical processors in a physical package. The value may not be the same as the number of logical processors that are present in the hardware of a physical package. The value of  $(1 + (\text{CPUID}(\text{EAX}=4, \text{ECX}=0):\text{EAX}[31:26]))$  represents the maximum number of addressable IDs (Core\_ID) that can be used to enumerate different processor cores in a physical package. The value also can be different than the actual number of processor cores that are present in the hardware of a physical package.

- **SMT\_ID**: The equivalent “SMT\_Mask\_Width” can be derived from dividing maximum number of addressable initial APIC IDs by maximum number of addressable Core IDs
  - $\text{SMT\_Mask\_Width} = \text{Log}_2^1(\text{RoundToNearestPof2}(\text{CPUID}(\text{EAX}=4, \text{ECX}=0):\text{EAX}[31:26]) / ((\text{CPUID}(\text{EAX}=4, \text{ECX}=0):\text{EAX}[31:26]) + 1))$ , where Log<sub>2</sub> is the logarithmic based on 2 and RoundToNearestPof2() operation is to round the input integer to the nearest power-of-two integer that is not less than the input value.
  - $\text{SMT\_Select\_Mask} = \sim(-1) \ll \text{SMT\_Mask\_Width}$
  - $\text{SMT\_ID} = \text{INITIAL\_APIC\_ID} \& \text{SMT\_Select\_Mask}$
- **Core\_ID**: The value of  $(1 + (\text{CPUID}(\text{EAX}=04\text{H}, \text{ECX}=0):\text{EAX}[31:26]))$  can also be use to derive an equivalent “CoreOnly\_Mask\_Width”.
  - $\text{CoreOnly\_Mask\_Width} = \text{Log}_2(1 + (\text{CPUID}(\text{EAX}=4, \text{ECX}=0):\text{EAX}[31:26]))$
  - $\text{CoreOnly\_Select\_Mask} = \sim(-1) \ll (\text{CoreOnly\_Mask\_Width} + \text{SMT\_Mask\_Width}) \wedge \text{SMT\_Select\_Mask}$ .
  - $\text{Core\_ID} = (\text{INITIAL\_APIC\_ID} \& \text{CoreOnly\_Select\_Mask}) \gg \text{SMT\_Mask\_Width}$
- **Pkg\_ID**: Pkg\_Select\_Mask can be derived as follows;
  - $\text{CorePlus\_Mask\_Width} = \text{CoreOnly\_Mask\_Width} + \text{SMT\_Mask\_Width}$

<sup>1</sup> Evaluating the Log<sub>2</sub> value of a positive number that happens to be a power of 2 can be implemented using integer operation. For example, the BSR instruction can be used to obtain the index position of the most significant bit that is not zero.

- $\text{Pkg\_Select\_Mask} = ((-1) \ll \text{CorePlus\_Mask\_Width})$
- $\text{Pkg\_ID} = (\text{INITIAL\_APIC\_ID} \& \text{Pkg\_Select\_Mask}) \gg \text{CorePlus\_Mask\_Width}$

Table A-8 lists the support function to derive bitmask extraction parameters from CPUID leaf 01H and Leaf 04H to extract sub IDs from initial APIC ID. Table A-9 shows the support function to derive mask widths from the system-wide extraction parameters.

## Cache ID Extraction Parameters

Cache IDs are specific to a target level cache of the cache hierarchy. Software must determine, a priori, the target cache level (the sub-level index  $n$  associated with CPUID leaf 4) it wishes to optimize with respect to the processor topology. After it has chosen the sub-leaf index  $\text{ECX}=n$ , then  $\text{Log}_2(\text{RoundToNearestPow2}(1 + \text{CPUID}(\text{EAX}=4, \text{ECX}=n):\text{EAX}[25:14]))$  is the equivalent “Cache\_Mask\_Width” parameter. The “Cache\_Mask\_Width” parameter forms the basis to construct either a select mask to extract the sub IDs of logical processors sharing the target cache level, or a complementary mask to select the upper bits from APID to identify different cache entities of the specified target level in the system. To construct a mask to extract sub IDs of different logical processors sharing a cache, it is simply  $\sim((-1) \ll \text{Cache\_Mask\_Width})$ .

The derivation of bitmask extraction parameters for cache topology is analogous to those shown in Table A-8. Software may choose to focus on one specific cache level in the cache hierarchy. In the companion full source code package that is released separately with this paper, the reader can find code examples of derivation for the bitmask extraction parameters for each cache level, and corresponding cache topology sorting examples. For space consideration, the full source of the cache topology code is not listed in the Appendix.

## Analyzing Topology Enumeration Result and Customization

How can software make use of topological information (in the form of hierarchical sub IDs)? This really depends on the needs and situations specific to each application. It may need adaptation due to differences of APIs provided by different OS. For the purpose of illustration, we consider some examples of using sub IDs to establish manage affinity masks hierarchically.

The knowledge of sub IDs of each topological hierarchy may be useful in several ways, For example:

1. Count the number of entities in a given hierarchical level across the system;
2. Use OS threading management services (e.g. affinity masks) while adding topological insights (per-core, per-package, per-target-level-cache) to optimize application performance.

Affinity mask is a data structure that is defined within a specific OS, different OS may use the same concept but providing different means of application programming interface. For example, Microsoft Windows\* provides affinity mask as a data type that can be directly manipulated via bit field by applications for affinity control. Linux implements a similar data structure internally but abstracts it so application can manipulate affinity through an iterative interface that assigned zero-based numbers to each logical processor.

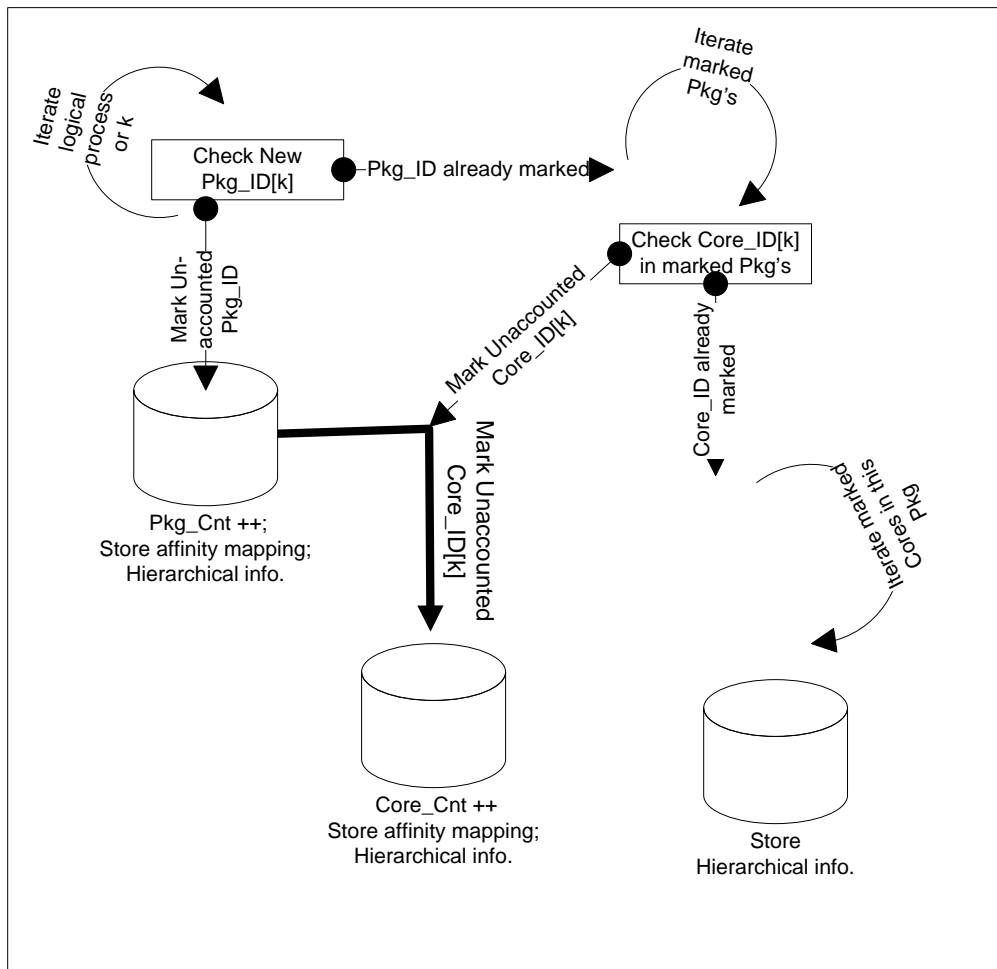
The affinity mask or the equivalent numbering scheme provided by OS does not carry attributes that can store hierarchical attributes of the system topology. We will use the “affinity mask” terminology generically in this section (as the technique can be easily generalized to the numbered interface of affinity control).

In the reference code example, we use the sub IDs to create an ordinal numbering scheme (zero-based) for each hierarchical level. Different entities in the system topology (packages, cores) can be referenced by applications using a set of hierarchical ordinal numbers. Using the hierarchical ordinal number scheme and a look-up table to the corresponding affinity masks, software can easily control thread binding, optimize cache usage, etc.

Figure 4 depicts a basic example of data processing of Pkg\_IDs and Core\_IDs in the system to acquire information on the number of software visible physical packages, processor cores in the system. This basic technique can also be adapted to acquire affinity mappings, hierarchical breakdowns, and asymmetry information in the system.

Table A-10 part a, b, and c lists an algorithm to analyze the sub IDs of all logical processor in the system and derive a triplet of zero-based numbering scheme to index unique entities within each topological level.

Table A-11 lists a data structure that organizes miscellaneous global variable, arrays, workspace items that are used throughout the rest of the code example. The full set of source code is provided in a separate package. The full source code can be compiled under 32-bit and 64-bit Windows and Linux operating systems. A limited set of OS and compiler tools have been tested.



## Dynamic Software Visibility of Topology Enumeration

When application software examines/uses topology information, it must keep in mind the dynamic nature of software visibility. The hardware capability present at the platform level may be presented differently through BIOS setting, through OS boot option, through OS-supported user interfaces. For example, Intel 64 and IA-32 multi-processor system require each physical processor support the same hardware multi-threading capabilities. This hardware symmetry that exists at the platform hardware level may be presented differently at application level. System topology enumeration can uncover dynamic software visible asymmetry irrespective of the cause of such asymmetry may be caused by BIOS setting, OS boot option, or UI configurations.

The appendix lists the bulk of the supporting functions that are used in enumerating processor topology of the system as visible to the current software process. A complete source code is provided separately for download. The reference code can be compiled in either 32-bit or 64-bit Windows\* environment. In 64-bit environment, the cpuid64.asm file is needed to provide an enhanced intrinsic function for querying CPUID sub-leaves. An equivalent reference code implementation for 32-bit and 64-bit Linux environment will also be available.

### Glossary

**Physical Processor:** The physical package of a microprocessor capable of executing one or more threads of software at the same time. Each physical package plugs into a physical socket. Each physical package may contain one or more processor cores, also referred to as a **physical package**.

**Processor Core:** The circuitry that provides dedicated functionalities to decode, execute instructions, and transfer data between certain sub-systems in a physical package. A processor core may contain one or more logical processors.

**Logical Processor:** The basic modularity of processor hardware resource that allow software executive (OS) to dispatch task or execute a thread context. Each logical processor can execute only one thread context at a time.

**Hyper-Threading Technology:** A feature within the IA-32 family of processors, where each processor core provides the functionality of more than one logical processor.

**SMT:** Abbreviated name for Simultaneous Multi-Threading. An efficient means in silicon to provide the functionalities of multiple logical processors within the same processor core by sharing execution resources and cache hierarchy between logical processors.

**Multi-core Processor:** A physical processor that contains more than one processor cores.

**Multi-processor Platform:** A computer system made of two or more physical sockets.

**Hardware Multi-threading:** Refers to any combination of hardware support to allow a system to run multi-threaded software. The forms of hardware support for multi-threading are: SMT, multi-core, and multi-processor.

**Processor Topology:** Hierarchical relationships processor entities (logical processors, processor cores) within a physical package relative to the sharing hierarchy of hardware resources within the physical processor.

**Cache Hierarchy:** Physical arrangement of cache levels that buffers data transport between a processor entity and the physical memory subsystem.

**Cache Topology:** Hierarchical relationships of a cache level relative to the logical processors in a physical processor.

## Appendix

Table A-1 Determination of System-wide CPU Topology Constant

```
// Derive parameters used to extract/decompose APIC ID for CPU topology
// The algorithm assumes CPUID feature symmetry across all physical packages.
// Since CPUID reporting by each logical processor in a physical package are
// identical, we only execute CPUID on one logical processor to derive these
// system-wide parameters
// return 0 if successful, non-zero if error occurred
static int CPUPopologyParams()
{
    DWORD maxCPUID; // highest CPUID leaf index this processor supports
    CPUIDinfo info; // data structure to store register data reported by CPUID

    _CPUID(&info, 0, 0);
    maxCPUID = info.EAX;

    // cpuid leaf B detection
    if (maxCPUID >= 0xB)
    {
        CPUIDinfo CPUInfoB;
        _CPUID(&CPUInfoB, 0xB, 0);
        //glbl_ptr points to assortment of global data, workspace, etc
        glbl_ptr->hasLeafB = (CPUInfoB.EBX != 0);
    }
    _CPUID(&info, 1, 0);

    // Use HWMT feature flag CPUID.01:EDX[28] to treat three configurations:
    if (getBitsFromDWORD(info.EDX, 28, 28))
    {
        // #1, Processors that support CPUID leaf 0BH
        if (glbl_ptr->hasLeafB)
        {
            // use CPUID leaf B to derive extraction parameters
            CPUPopologyLeafBConstants();
        }
        else // #2, Processors that support legacy parameters
            // using CPUID leaf 1 and leaf 4
        {
            CPUPopologyLegacyConstants(&info, maxCPUID);
        }
    }
    else // #3, Prior to HT, there is only one logical
        //processor in a physical package
    {
        glbl_ptr->CoreSelectMask = 0;
        glbl_ptr->SMTMaskWidth = 0;
        glbl_ptr->PkgSelectMask = (-1);
        glbl_ptr->PkgSelectMaskShift = 0;
        glbl_ptr->SMTSelectMask = 0;
    }

    if( glbl_ptr->error)return -1;
    else return 0;
}
```

Table A-2 Modular Structure of Deriving System Topology Enumeration Information

```
/*
 * BuildSystemTopologyTables
 *
 * Construct the processor topology tables and values necessary to
 * support the external functions that display CPU topology and/or
 * cache topology derived from system topology enumeration.
 * Arguments:      None
 * Return:         None, sets glbl_ptr->error if tables or values can not be
 * calculated.
 */
static void      BuildSystemTopologyTables()
{  unsigned lcl_OSProcessorCount, subleaf;
   int      numMappings = 0;
   // call OS-specific service to find out how many logical processors
   // are supported by the OS
   glbl_ptr->OSProcessorCount = lcl_OSProcessorCount = GetMaxCPUSupportedByOS();

   // allocated the memory buffers within the global pointer
   AllocArrays(lcl_OSProcessorCount);

   // Gather all the system-wide constant parameters needed to
   // derive topology information
   if (CPUTopologyParams() ) return ;
   if (CacheTopologyParams() ) return ;

   // For each logical processor, collect APIC ID and
   // parse sub IDs for each APIC ID
   numMappings = QueryParseSubIDs();
   if ( numMappings < 0 ) return ;
   // Derived separate numbering schemes for each level of the cpu topology
   if( AnalyzeCPUHierarchy(numMappings) < 0 ) {
       glbl_ptr->error |= _MSGTYP_TOPOLOGY_NOTANALYZED;
   }
   // an example of building cache topology info for each cache level
   if( glbl_ptr->maxCacheSubleaf != -1) {
       for(subleaf=0; subleaf <= glbl_ptr->maxCacheSubleaf; subleaf++) {
           if( glbl_ptr->EachCacheMaskWidth[subleaf] != 0xffffffff) {
               // ensure there is at least one core in the target level cache
               if (AnalyzeEachCHierarchy(subleaf, numMappings) < 0) {
                   glbl_ptr->error |= _MSGTYP_TOPOLOGY_NOTANALYZED;
               }
           }
       }
   }
}
```

**Table A-3 Data Structure of APIC ID, Sub IDs, and Mapping of Ordinal Based Numbering Schemes**

```

typedef struct {
unsigned int32 APICID;    // the full x2APIC ID or initial APIC ID of a logical
                        // processor assigned by HW
unsigned __int32 OrdIndexOAMsk; // An ordinal index (zero-based) for each logical
                        // processor in the system, 1:1 with "APICID"
// Next three members are the sub IDs for processor topology enumeration
unsigned __int32 pkg_IDAPIC;    // Pkg_ID field, subset of APICID bits
                        // to distinguish different packages
unsigned __int32 Core_IDAPIC;    // Core_ID field, subset of APICID bits to
                        // distinguish different cores in a package
unsigned __int32 SMT_IDAPIC;    // SMT_ID field, subset of APICID bits to
                        // distinguish different logical processors in a core
// the next three members stores a numbering scheme of ordinal index
// for each level of the processor topology.
unsigned __int32 packageORD;    // a zero-based numbering scheme for each physical
                        // package in the system
unsigned __int32 coreORD; // a zero-based numbering scheme for each core in the
                        // same package
unsigned __int32 threadORD; // a zero-based numbering scheme for each thread in
                        // the same core
// Next two members are the sub IDs for cache topology enumeration
unsigned __int32 EaCacheSMTIDAPIC[MAX_CACHE_SUBLEAFS]; // SMT_ID field, subset of
// APICID bits to distinguish different logical processors
// sharing the same cache level
unsigned __int32 EaCacheIDAPIC[MAX_CACHE_SUBLEAFS]; // sub ID to enumerate
// different cache entities of the cache level corresponding
// to the array index/cpuid leaf 4 subleaf index
// the next three members stores a numbering scheme of ordinal index
// for enumerating different cache entities of a cache level, and enumerating
// logical processors sharing the same cache entity.
unsigned __int32 EachCacheORD[MAX_CACHE_SUBLEAFS]; // a zero-based numbering
// scheme for each cache entity of the specified cache level in the system
unsigned __int32 threadPerEaCacheORD[MAX_CACHE_SUBLEAFS]; // a zero-based
// numbering scheme for each logical processor sharing the same cache of the
// specified cache level

} IdAffMskOrdMapping;

/* Alternate technique for ring 3 code to infer the effect of CMOS setting in BIOS
 * that restricted CPUID instruction to report highest leaf index is 2, i.e.
 * MSR IA32_MISC_ENABLES[22] was set to 1; This situation
 * will prevent software from using CPUID to conduct topology enumeration
 * RDMSR instruction is privileged, this alternate routine can run in ring 3.
 */
Int InferBIOSCPUIDLimitSetting()
{
    DWORD maxleaf, max8xleaf;
    CPUIDinfo info; // data structure to store register data reported by CPUID
// check CPUID leaf reporting capability is intact
    CPUID(&info, 0);
    maxleaf = info.EAX;
    CPUID(&info, 0x80000000);
    max8xleaf = info.EAX;
    // Earlier Pentium 4 and Intel Xeon processor (prior to 90nm Intel Pentium 4
    // processor)support extended with max extended leaf index 0x80000004,
    // 90nm Intel Pentium 4 processor and later processors supports higher extended
    // leaf index greater than 0x80000004.
    If ( maxleaf <= 4 && max8xleaf > 0x80000004) return 1;
    else return 0;
}

```

Table A-4 Query APIC ID and Parsing APIC ID into Sub IDs

```

/*
 * QueryParseSubIDs
 *
 * Use OS specific service to find out how many logical processors can be accessed
 * by this application.
 * Querying CPUID on each logical processor requires using OS-specific API to
 * bind current context to each logical processor first.
 * After gathering the APIC ID's for each logical processor,
 * we can parse APIC ID into sub IDs for each topological levels
 * The thread affinity API to bind the current context limits us
 * in dealing with the limit of specific OS
 * The loop to iterate each logical processor managed by the OS can be done
 * in a manner that abstract the OS-specific affinity mask data structure.
 * Here, we construct a generic affinity mask that can handle arbitrary number
 * of logical processors.
 * Return:      0 is no error
 */
long QueryParseSubIDs(void)
{
    unsigned i;
    //DWORD_PTR processAffinity;
    //DWORD_PTR systemAffinity;
    unsigned long numMappings = 0, lcl_OSProcessorCount;
    unsigned long APICID;
    // we already queried OS how many logical processor it sees.
    lcl_OSProcessorCount = glbl_ptr->OSProcessorCount;
    // we will use our generic affinity bitmap that can be generalized from
    // OS specific affinity mask constructs or the bitmap representation of an OS
    AllocateGenericAffinityMask(&glbl_ptr->cpuid_values_processAffinity,
    lcl_OSProcessorCount);
    AllocateGenericAffinityMask(&glbl_ptr->cpuid_values_systemAffinity,
    lcl_OSProcessorCount);
    // Set the affinity bits of our generic affinity bitmap according to
    // the system affinity mask and process affinity mask
    SetChkProcessAffinityConsistency(lcl_OSProcessorCount);
    if (glbl_ptr->error) return -1;

    for (i=0; i < glbl_ptr->OSProcessorCount;i++) {
        // can't asume OS affinity bit mask is contiguous,
        // but we are using our generic bitmap representation for affinity
        if(TestGenericAffinityBit(&glbl_ptr->cpuid_values_processAffinity, i) == 1) {
            // bind the execution context to the ith logical processor
            // using OS-specific API
            if( BindContext(i, glbl_ptr->cpuid_values_OSProcessorCount) ) {
                glbl_ptr->error |= _MSGTYP_UNKNOWNERR_OS;
                break;
            }
            // now the execution context is on the i'th cpu, call the parsing routine
            ParseIDS4EachThread(i, numMappings);
            numMappings++;
        }
    }
    glbl_ptr->EnumeratedThreadCount = numMappings;
    if( glbl_ptr->error)return -1;
    else return numMappings;
};

```

Table A-5 Support Routine for Parsing APIC ID into Sub IDs

```

/*
 * ParseIDS4EachThread
 * after execution context has already bound to the target logical processor
 * Query the 32-bit x2APIC ID if the processor supports it, or
 * Query the 8-bit initial APIC ID for older processors. Apply various
 * system-wide topology constant to parse the APIC ID into various sub IDs
 * Arguments:
 * i : the ordinal index to reference a logical processor in the system
 * numMappings : running count of how many processors we've parsed
 * Return:      0 is no error
 */
unsigned ParseIDS4EachThread(unsigned i, unsigned numMappings)
{
    unsigned APICID;
    unsigned subleaf;

    APICID = glbl_ptr->PApicAffOrdMapping[numMappings].APICID = GetApicID(i);
    glbl_ptr->PApicAffOrdMapping[numMappings].OrdIndexOAMsk = i; // this an
ordinal number that can relate to generic affinitymask
    glbl_ptr->PApicAffOrdMapping[numMappings].pkg_IDAPIC = ((APICID & glbl_ptr-
>PkgSelectMask) >> glbl_ptr->PkgSelectMaskShift);
    glbl_ptr->PApicAffOrdMapping[numMappings].Core_IDAPIC = ((APICID & glbl_ptr-
>CoreSelectMask) >> glbl_ptr->SMTMaskWidth);
    glbl_ptr->PApicAffOrdMapping[numMappings].SMT_IDAPIC = (APICID & glbl_ptr-
>SMTSelectMask);
    if (glbl_ptr->maxCacheSubleaf != -1) {
        for (subleaf=0; subleaf <= glbl_ptr->maxCacheSubleaf; subleaf++) {
            glbl_ptr->PApicAffOrdMapping[numMappings].EaCacheSMTIDAPIC[subleaf]
= (APICID & glbl_ptr->EachCacheSelectMask[subleaf]);
            glbl_ptr->PApicAffOrdMapping[numMappings].EaCacheIDAPIC[subleaf]
= (APICID & (-1 ^ glbl_ptr->EachCacheSelectMask[subleaf]));
        }
    }
    return 0;
}

/*
 * GetApicID
 * Returns APIC ID from leaf B if it else from leaf 1
 * Arguments:      None
 * Return:         APIC ID
 */
static unsigned GetApicID() {
    CPUIDinfo info;

    if ( glbl_ptr->hasLeafB) {
        CPUID(&info,0xB); // query subleaf 0 of leaf B
        return info.EDX; // x2APIC ID
    }
    CPUID(&info,1);
    return (BYTE)(getBitsFromDWORD(info.EBX,24,31)); // zero extend 8-bit initial
APIC ID
}

```

## Table A-6 OS-Specific Wrapper Functions

```

#define LNX_MY1CON 1

extern  GLKTSN_T * gbl_ptr;
extern int countBits(DWORD_PTR x);

/*
 * BindContext
 * A wrapper function that can compile under two OS environments. The size
 * of the bitmap that underlies cpu_set_t is configurable at Linux Kernel C
 * ompile time. Each distro may set limit on its own. Some newer Linux distro
 * may support 256 logical processors, For simplicity we don't show
 * the check for range on the ordinal index of the target cpu in Linux,
 * interested reader can check Linux kernel documentation.
 * Current Windows OS has size limit of 64 cpu in 64-bit mode, 32 in 32-bit mode
 * the size limit is checked.
 * Arguments:
 *   cpu :   the ordinal index to reference a logical processor in the system
 * Return:   0 is no error
 */
unsigned BindContext(unsigned cpu)
{unsigned ret = -1;
#ifdef __linux__
    cpu_set_t currentCPU;
    // add check for size of cpumask_t.
    __CPU_ZERO(&currentCPU);
    // turn on the equivalent bit inside the bitmap corresponding to affinitymask
    __CPU_SET(cpu, &currentCPU);
    if ( !sched_setaffinity (0, sizeof(currentCPU), &currentCPU) )
    {   ret = 0;
    }
#else
    DWORD_PTR affinity, last_affinity;
    if( cpu >= MAX_LOG_CPU ) return ret;
    // flip on the bit in the affinity mask corresponding to the input ordinal
index
    affinity = (DWORD_PTR)(LNX_MY1CON << cpu);
    if ( SetThreadAffinityMask(GetCurrentThread(),affinity) )
    { ret = 0;}
#endif
    return ret;
}

/*
 * GetMaxCPUSupportedByOS
 * A wrapper function that calls OS specific system API to find out
 * how many logical processor the OS supports
 * Return:   a non-zero value
 */
unsigned GetMaxCPUSupportedByOS()
{unsigned lcl_OSProcessorCount = 0;
#ifdef __linux__
    //This will tell us how many CPUs are currently enabled.
    lcl_OSProcessorCount = sysconf(_SC_NPROCESSORS_CONF);
#else
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    lcl_OSProcessorCount = si.dwNumberOfProcessors;
#endif
    return lcl_OSProcessorCount;
}

```

Table A-7 Derivation of CPUID Leaf 0BH Parameters for Topology Enumeration

```

// Derive bitmask extraction parameters used to extract/decompose x2APIC ID.
// The algorithm assumes CPUID feature symmetry across all physical packages.
// Since CPUID reporting by each logical processor in a physical package
// are identical, we only execute CPUID on one logical processor to derive
// these system-wide parameters
int CPUPopologyLeafBConstants()
{
    CPUIDinfo infoB;
    int wasCoreReported = 0;
    int wasThreadReported = 0;
    int subLeaf = 0, levelType, levelShift;
    unsigned long coreplusSMT_Mask = 0;
    do
    {
        // we already tested CPUID leaf 0xB contain valid sub-leaves,
        _CPUID(&infoB,0xB,subLeaf);
        if (infoB.EBX == 0)
        {
            if EBX[15:0], this subleaf is not valid, we can exit the loop
            break;
        }
        levelType = getBitsFromDWORD(infoB.ECX,8,15);
        levelShift = getBitsFromDWORD(infoB.EAX,0,4);
        switch (levelType)
        {
            case 1: //level type is SMT, so levelShift is the SMT_Mask_Width
                glbl_ptr->SMTSelectMask = ~((-1) << levelShift);
                glbl_ptr->SMTMaskWidth = levelShift;
                wasThreadReported = 1;
                break;
            case 2: //level type is Core, so levelShift is the CorePlsuSMT_Mask_Width
                coreplusSMT_Mask = ~((-1) << levelShift);
                glbl_ptr->PkgSelectMaskShift = levelShift;
                glbl_ptr->PkgSelectMask = (-1) ^ coreplusSMT_Mask;
                wasCoreReported = 1;
                break;
            default:
                // handle in the future
                break;
        }
        subLeaf++;
    } while (1);

    if (wasThreadReported && wasCoreReported)
    {
        glbl_ptr->CoreSelectMask = coreplusSMT_Mask ^ glbl_ptr->SMTSelectMask;
    }
    else if (!wasCoreReported && wasThreadReported)
    {
        glbl_ptr->CoreSelectMask = 0;
        glbl_ptr->PkgSelectMaskShift = glbl_ptr->SMTMaskWidth;
        glbl_ptr->PkgSelectMask = (-1) ^ glbl_ptr->SMTSelectMask;
    }
    else //(case where !wasThreadReported)
    {
        // throw an error, this should not happen if hardware function normally
        glbl_ptr->error |= _MSGTYP_GENERAL_ERROR;
    }
    if( glbl_ptr->error)return -1;
    else return 0;
}

```

**Table A-8 Derivation of Legacy CPUID Leaf 1 and Leaf 4 Parameters for Topology Enumeration**

```

// Calculate parameters used to extract/decompose Initial APIC ID.
// The algorithm assumes CPUID feature symmetry across all physical packages.
// Since CPUID reporting by each logical processor in a physical package are
// identical, we only execute CPUID on one logical processor
/*
 * CPUTopologyLegacyConstants
 * Derive bitmask extraction parameter using CPUID leaf 1 and leaf 4
 * Arguments:
 *   info      Point to structure containing CPUID instruction leaf 1 data
 *   maxCPUID  Maximum CPUID Leaf number supported by the processor
 * Return:    0 is no error
 */
int CPUTopologyLegacyConstants( CPUIDinfo *pinfo, DWORD maxCPUID)
{
    unsigned corePlusSMTIDMaxCnt;
    unsigned   coreIDMaxCnt = 1;
    unsigned   SMTIDPerCoreMaxCnt = 1;
    // CPUID.1:EBX[23:16] provides the max # IDs that can be enumerated
    // under the CorePlusSMT_SelectMask

    corePlusSMTIDMaxCnt = getBitsFromDWORD(pinfo->EBX,16,23);

    if (maxCPUID >= 4) // support CPUID 4?
    {
        CPUIDinfo info4;
        _CPUID(&info4, 4, 0);
        // (CPUID.(EAX=4, ECX=00:EAX[31:26] +1 ) provides the max # of Core_IDs
        // that's allocated in a package, this is // related to coreMaskWidth

        coreIDMaxCnt = getBitsFromDWORD(info4.EAX,26,31)+1;
        SMTIDPerCoreMaxCnt = corePlusSMTIDMaxCnt / coreIDMaxCnt;
    }
    else// no support for CPUID leaf 4 but caller has verified HT support
    {
        if (!glbl_ptr->Alert_BiosCPUIDmaxLimitSetting) {
            coreIDMaxCnt = 1;
            SMTIDPerCoreMaxCnt = corePlusSMTIDMaxCnt / coreIDMaxCnt;
        }
        else { // we got here most likely because
            // IA32_MISC_ENABLES[22] was set to 1 by BIOS
            glbl_ptr->error |= _MSGTYP_CHECKBIOS_CPUIDMAXSETTING;
            // IA32_MISC_ENABLES[22] may have been set to 1,
            // it will cause inaccurate reporting
        }
    }

    glbl_ptr->SMTSelectMask = createMask(SMTIDPerCoreMaxCnt,&glbl_ptr->SMTMaskWidth);
    glbl_ptr->CoreSelectMask = createMask(coreIDMaxCnt,&glbl_ptr->
>PkgSelectMaskShift);
    glbl_ptr->PkgSelectMaskShift += glbl_ptr->SMTMaskWidth;
    glbl_ptr->CoreSelectMask <<= glbl_ptr->SMTMaskWidth;
    glbl_ptr->PkgSelectMask = (-1) ^ (glbl_ptr->CoreSelectMask | glbl_ptr->
>SMTSelectMask);
    return 0;
}

```

**Table A-9 Support Functions to Generate Bitmask for Extraction of Sub IDs**

```

/*
 * myBitScanReverse
 *
 * Equivalent functionality of BSR
 * This c-emulation of the BSR instruction is shown here for tool
portability

 * Arguments:
 *   index      bit offset of the most significant bit that's not 0 found in mask
 *   mask       input data to search the most significant bit
 * Return:      1 if a non-zero bit is found, otherwise 0
 */
unsigned char myBitScanReverse(unsigned long * index, unsigned long mask)
{unsigned long i;

    for(i=(8*sizeof(unsigned long)); i > 0; i--) {
        if((mask & (LNX_MY1CON << (i-1))) != 0) {
            *index = (i-1);
            break;
        }
    }
    return ( mask != 0);
}

/* createMask
 *
 * Derive a bit mask and associated mask width (# of bits) such that
 * the bit mask is wide enough to select the specified number of
 * distinct values "numEntries" within the bit field defined by maskWidth.
 * Arguments:
 *   numEntries : The number of entries in the bit field for which a mask needs to
be created
 *   maskWidth: Optional argument, pointer to argument that get the mask width (#
of bits)
 *
 * Return:      Created mask of all 1's up to the maskWidth
 */
static unsigned long createMask(unsigned numEntries, unsigned *maskWidth)
{
    unsigned i;
    unsigned long k;

    // NearestPo2(numEntries) is the nearest power of 2 integer that is not less
// than numEntries
// The most significant bit of (numEntries * 2 -1) matches the above definition

    k = (unsigned long)(numEntries) * 2 -1;

    if (myBitScanReverse(&i, k) == 0)
    {
        // No bits set
        if (maskWidth) *maskWidth = 0;
        return 0;
    }

    if (maskWidth) *maskWidth = i;

    if (i == 31) return -1;

    return (1 << i) -1;
}

```

Table A-10a Part 1 of Algorithm to Sort Sub IDs into Hierarchical Numbering Scheme

```

/*
 * AnalyzeCPUIierarchy
 * Analyze the Pkg_ID, Core_ID to derive hierarchical ordinal numbering scheme
 * Arguments:
 *   numMappings:  the number of logical processors successfully queried
 *                 with SMT_ID, Core_ID, Pkg_ID extracted
 * Return:       0 is no error
 */
static int AnalyzeCPUIierarchy(unsigned long numMappings)
{
    unsigned i, ckDim, maxPackageDetected = 0;
    unsigned APICID;
    unsigned packageID, coreID, threadID;
    unsigned *pDetectCoreIDsperPkg, *pDetectedPkgIDs;
    // allocate workspace to sort parents and siblings in the topology
    // starting from pkg_ID and work our ways down each inner level
    pDetectedPkgIDs = (unsigned long *)_alloca( numMappings * sizeof(unsigned long)
);
    if(pDetectedPkgIDs == NULL) return -1;
    // we got a 1-D array to store unique Pkg_ID as we sort thru
    // each logical processor
    memset(pDetectedPkgIDs, 0xff, numMappings*sizeof(unsigned long) );
    ckDim = numMappings * ( 1 << glbl_ptr->PkgSelectMaskShift);
    pDetectCoreIDsperPkg = (unsigned long *)_alloca( ckDim * sizeof(unsigned long) );
    if(pDetectCoreIDsperPkg == NULL) return -1;
    // we got a 2-D array to store unique Core_ID within each Pkg_ID,
    // as we sort thru each logical processor
    memset(pDetectCoreIDsperPkg, 0xff, ckDim * sizeof(unsigned long));

    // iterate through each logical processor in the system.
    // mark up each unique physical package with a zero-based numbering scheme
    // Within each distinct package, mark up distinct cores within that package
    // with a zero-based numbering scheme
    for (i=0; i < numMappings;i++) {
        BOOL PkgMarked;
        unsigned h;
        APICID = glbl_ptr->PApicAffOrdMapping[i].APICID;
        packageID = glbl_ptr->PApicAffOrdMapping[i].pkg_IDAPIC ;
        coreID = glbl_ptr->PApicAffOrdMapping[i].Core_IDAPIC ;
        threadID = glbl_ptr->PApicAffOrdMapping[i].SMT_IDAPIC;

        PkgMarked = FALSE;
    }
}

```

Table A-10b Part 2 of Algorithm to Sort Sub IDs into Hierarchical Numbering Scheme

```

for (h=0;h<maxPackageDetected;h++)
{
    if (pDetectedPkgIDs[h] == packageID)
    {
        BOOL foundCore = FALSE;
        unsigned k;
        PkgMarked = TRUE;
        glbl_ptr->PApicAffOrdMapping[i].packageORD = h;

        // look for core in marked packages
        for (k=0;k<glbl_ptr->perPkg_detectedCoresCount.data[h];k++)
        {
            if (coreID == pDetectCoreIDsperPkg[h* numMappings +k])
            {
                foundCore = TRUE;
                // add thread - can't be that the thread already exists,
                breaks unique APICID spec
                glbl_ptr->PApicAffOrdMapping[i].coreORD = k;
                glbl_ptr->PApicAffOrdMapping[i].threadORD = glbl_ptr->
                perCore_detectedThreadsCount.data[h*MAX_CORES+k];
                glbl_ptr->perCore_detectedThreadsCount.data[h*MAX_CORES+k]++;
                break;
            }
        }
        if (!foundCore)
        { // mark up the Core_ID of an unmarked core in a marked package
            unsigned core = glbl_ptr->perPkg_detectedCoresCount.data[h];
            pDetectCoreIDsperPkg[h* numMappings + core] = coreID;
            // keep track of respective hierarchical counts
            glbl_ptr->perCore_detectedThreadsCount.data[h*MAX_CORES+core] = 1;
            glbl_ptr->perPkg_detectedCoresCount.data[h]++;
            // build a set of numbering system to iterate each topological
            hierarchy
            glbl_ptr->PApicAffOrdMapping[i].coreORD = core;
            glbl_ptr->PApicAffOrdMapping[i].threadORD = 0;
            glbl_ptr->EnumeratedCoreCount++; // this is an unmarked core,
            increment system core count by 1
        }
        break;
    }
}

```

Table A-10c Part 3 of Algorithm to Sort Sub IDs into Hierarchical Numbering Scheme

```
    if (!PkgMarked)
    { // mark up the pkg_ID and Core_ID of an unmarked package
      pDetectedPkgIDs[maxPackageDetected] = packageID;
      pDetectCoreIDsperPkg[maxPackageDetected* numMappings + 0] = coreID;
      // keep track of respective hierarchical counts
      glbl_ptr->perPkg_detectedCoresCount.data[maxPackageDetected] = 1;
      glbl_ptr->perCore_detectedThreadsCount.data
[maxPackageDetected*MAX_CORES+0] = 1;
      // build a set of zero-based numbering scheme so that
      // each logical processor in the same core can be referenced by a
      // zero-based index
      // each core in the same package can be referenced by another
      // zero-based index
      // each package in the system can be referenced by a third
      // zero-based index scheme.
      // each system wide index i can be mapped to a triplet of
      // zero-based hierarchical indices
      glbl_ptr->PApicAffOrdMapping[i].packageORD = maxPackageDetected;
      glbl_ptr->PApicAffOrdMapping[i].coreORD = 0;
      glbl_ptr->PApicAffOrdMapping[i].threadORD = 0;
      // this is an unmarked pkg, increment pkg count by 1
      maxPackageDetected++;
      // there is at least one core in a package
      glbl_ptr->EnumeratedCoreCount++;
    }
  }
  glbl_ptr->EnumeratedPkgCount = maxPackageDetected;
  return 0;
}
```

Table A-11 Miscellaneous Global Variables, Arrays, Workspaces Organized in a Data Structure

```

typedef struct
{
    // for each logical processor we need spaces to store APIC ID,
    // sub IDs, affinity mappings, etc.
    IdAffMskOrdMapping *pApicAffOrdMapping;
    // workspace for storing hierarchical counts of each level
    DynlArr_str    perPkg_detectedCoresCount;
    Dyn2Arr_str    perCore_detectedThreadsCount;
    // workspace for storing hierarchical counts relative to the cache topology
    // of the largest unified cache (may be shared by several cores)
    DynlArr_str    perCache_detectedCoreCount;
    Dyn2Arr_str    perEachCache_detectedThreadCount;
    // we use an error code to indicate any abnormal situation
    unsigned    error;
    // If CPUID full reporting capability is restricted, we need to know.
    unsigned    Alert_BiosCPUIDmaxLimitSetting;

    unsigned    OSProcessorCount; // how many logical processor the OS sees
    // flag to keep track of whether CPUID leaf 0BH is supported
    unsigned    hasLeafB;
    // keep track of highest CPUID leaf 4 subleaf index in a processor
    unsigned    maxCacheSubleaf;
    // the following global variables are the total counts in the system
    // resulting from software enumeration
    unsigned    EnumeratedPkgCount;
    unsigned    EnumeratedCoreCount;
    unsigned    EnumeratedThreadCount;
    // CPUID ID leaf 4 can report data for several cache levels, we'll
    // keep track of each cache level
    unsigned    EnumeratedEachCacheCount[MAX_CACHE_SUBLEAFS];
    // the following global variables are parameters related to
    // extracting sub IDs from an APIC ID, common to all processors in the system
    unsigned    SMTSelectMask;
    unsigned    PkgSelectMask;
    unsigned    CoreSelectMask;
    unsigned    PkgSelectMaskShift;
    unsigned    SMTMaskWidth;
    // We'll do sub ID extractions using parameters from each cache level
    unsigned    EachCacheSelectMask[MAX_CACHE_SUBLEAFS];
    unsigned    EachCacheMaskWidth[MAX_CACHE_SUBLEAFS];
    // the following global variables are used for product capability identification
    unsigned    HWMT_SMTperCore;
    unsigned    HWMT_SMTperPkg;
    // a data structure that can store simple leaves and complex subleaves of
    // all supported leaf indices of CPUID
    CPUIDinfx    *cpuid_values;
    // workspace of our generic affinitymask structure to allow iteration
    // over each logical processors in the system
    GenericAffinityMask    cpu_generic_processAffinity;
    GenericAffinityMask    cpu_generic_systemAffinity;
    // workspace to assist text display of cache topology information
    cacheDetail_str    cacheDetail[MAX_CACHE_SUBLEAFS];
} GLKTSN_T;

GLKTSN_T *glbl_ptr=NULL;

```

