



166166

14 MAY 2002



1

G00002

PREDICTIVE CACHE LOADING BY PROGRAM ADDRESS DISCONTINUITY HISTORY

FIELD OF THE INVENTION

The present invention relates to information handling systems and methods for operating information handling systems and, more particularly, to apparatuses and methods for loading cache memory in an information handling system having a multitasking operating system.

BACKGROUND OF THE INVENTION

As microprocessor or central processing unit (CPU) operating systems begin to rely on multitasking as a standard means of operation, the external data bus bandwidth, i.e., the data bus coupling the CPU to other units in an information handling system, becomes the system performance bottleneck during a task switch. As hereinafter used, the term "task" shall mean any separate program sequence or set of instructions. Typically, a task is considered to be any separate program sequence that lies beyond the current i pages of operation. For example, i could be 2 or 3 pages.

A task is an operating system (OS) dispatchable unit of work or computational activity. By performing multitasking or switching between tasks, an OS can both distribute CPU

access on an interactive, as well as batch, basis and increase CPU throughput and utilization.

Each application, when running on a CPU, is manifest by one or more OS tasks. That is, the OS assigns and dispatches a finite number of tasks to counterpart functions as specified by an application or applications. Usually, the number of tasks is significantly less than the number of functions sought by applications to be concurrently executed.

A task, also termed a process, is a program in execution. As a task or process executes, it changes its state. The state of a task is defined by its current activity (new, active, waiting, or halted). Relatedly, the state vector manifestation of a task is a construct denominated as a "task control block" (TCB).

To maximize CPU utilization, some task should be running at all times. In serial machines, this is accomplished by queued access. That is, tasks which are ready and awaiting execution are kept on a list called a work or task ready queue. Recurrently, an OS scheduler/dispatcher determines which task in the ready queue is next to be given access to the CPU. In this regard, a running task continues execution until it either terminates, goes into a wait state (such as caused by performing I/O), has exceeded available time, or is preempted.

Multitasking is accomplished by running or processing each task a little at a time, in a round robin approach. As tasks are switched, new code is loaded into an instruction cache memory. If enough task switches are accomplished since the last running of the original task, the code of the original task no longer resides in cache memory. When the task switch occurs, the code for the original task must be reloaded into cache memory just as if it was starting the task for the first time. This means that at each task switch, the external data bus is needed to load the cache. That is, the code corresponding to the original task is transferred from system memory via the external data bus to the cache memory residing within the microprocessor.

Since this external bus is running at some fraction of the internal processor speed, the data rate from the external bus to the internal cache is saturated, and the processor can stall. This problem applies to set associative and fully associative caches. This problem cannot presently be handled by conventional prefetch mechanisms, since the microprocessor does not know when this discontinuity in program flow caused by each task switch is going to occur or what new code is needed until it actually happens.

SUMMARY OF THE INVENTION

The present invention provides a solution to processor stalling caused by task switches by utilizing the bus bandwidth available immediately before a task switch. The aspects of the invention take advantage of the available external bus bandwidth, which occurs when the processor is addressing its internal cache during the processing of a current task. The aspects of the invention predict the next task needed in the internal cache and preload that task before the processor completes processing of the current task.

In accordance with a first aspect of the invention there is provided an information handling system comprising:

- a processor for processing a plurality of tasks in a program of instructions;
- cache memory for storing a predetermined number of said tasks;
- a task queue for storing a task list arranged in a chronological processing order from a current task to a next task;
- means for assigning a task ID to each of said tasks in the task queue; and
- means for preloading said next task into said cache memory during processing of said current task by the processor, said means for preloading comprising:
 - means for determining a number of bytes between the current task and the next task, during processing of said current task by the processor, based on the task IDs; and
- means responsive to the means for determining for activating, during processing of said current task by the processor, the means for preloading said next task into said cache memory if said means for determining determines that the number of bytes between the current task and said next task is not smaller than n , wherein n is a positive integer large enough to indicate a task switch.

In accordance with a second aspect of the invention there is provided a method for preloading cache memory in an information handling system comprising a processor for processing a plurality of tasks in a program of instructions and cache memory for storing a predetermined number of the tasks, said method comprising the steps of.

- storing a task list arranged in a chronological processing order from a current task to a next task;
- identifying the tasks including assigning a task ID to each of the tasks in the task list;

determining a number of bytes between the current task and the next task, based on the task IDs, before the processor completes processing of the current task; and

determining if the number of bytes is not smaller than n , wherein n is a positive integer large enough to indicate a task switch, including preloading the next task into cache memory if the number of bytes is not smaller than n before the processor completes processing of the current task.

The aspects of the present invention advantageously utilizes bus bandwidth available immediately before a task switch to prevent processor stalling.

The aspects of the present invention advantageously predict the next task to be processed so that the next task can be loaded into cache memory prior to task execution.

14 MAY 2002

6

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is an information handling system embodying the present invention.

Figure 2 illustrates a discontinuity in a program of instructions.

Figure 3 is a block diagram of the present invention.

Figures 4A-4C are a flow diagram illustrating the operation of the present invention.

DETAILED DESCRIPTION OF A PREFERRED EMBODIMENT
OF THE INVENTION

A representative hardware environment for practicing the present invention is depicted in Figure 1, which illustrates a typical hardware configuration of an information handling system having a multitasking operating system in accordance with the subject invention, comprising at least one processor or central processing unit (CPU) 10 with one or more integrated data transfer units 11, one or more levels of cache memory 13, and a number of other units such as one or more execution units (not shown) or a memory management unit (not shown). CPU 10 is interconnected via system bus 12 to random access memory (RAM) 14, read only memory (ROM) 16, and input/output (I/O) adapter 18 for connecting peripheral devices such as disc units 20 and tape drives 40 to bus 12, user interface adapter 22 for connecting keyboard 24, mouse 26, speaker 28, microphone 32, and/or other user interface devices such as a touch screen device (not shown) to bus 12, communication adapter 34 for connecting the information handling system to a data processing network, and display adapter 36 for connecting bus 12 to display device 38.

Cache memory 13 is one part of the overall concept of storage hierarchies where faster but less dense memories are placed closer to CPU 10, and slower but more dense or bigger memories back them up. Cache memory 13 may have one or more levels of storage hierarchy wherein the most frequently

requested data by CPU 10 is stored in a first level of cache memory, the next most frequently requested data is stored in a second level of cache memory and the least frequently requested data is stored in a main memory such as RAM 14 or ROM 16 located outside of CPU 10. When CPU 10 needs to make a request for data or instructions from memory, data transfer units 11 search cache memory 13 for the address of the desired data or instruction. If the data or instruction is available a hit occurs and the data or instruction is transferred. If the data or instruction is not in cache memory 13, then a miss, sometimes called a cache fault, occurs. If cache memory 13 has multiple levels, each level is searched one at a time until a hit occurs.

In general, the present invention takes advantage of the fact that in multitasking, each task is processed a little at a time in a round robin fashion. This means that circular list of tasks can be determined by watching the tasks run in their current sequence. The present invention keeps track of the different tasks in a task list arranged in chronological processing order to predict what task should be loaded in cache memory next and at what time. The present invention accomplishes this by looking at the next task in the sequence from the task list and loading the next task when the current task is almost finished.

Referring now to Figure 2, a program flow diagram is illustrated. Figure 2 illustrates a first or current task of instructions or programs 12 and a second task or next

task of instructions or programs 14. Upon execution or processing of task 12, a task switch or jump command occurs in section I 16 of task 12. The jump command or task switch command causes the flow of instructions to jump to the beginning of task 14, which occurs at section AA 18. Thus, a discontinuity in the processing order of instructions or programs has occurred, since the program of instructions has skipped section J 20 and jumped to a new section of the program of instructions. No discontinuity in the processing of the program of instructions occurs if each section or instruction in the program of instructions is processed in order.

The present invention detects the size of a jump in the instructions from section I 16 to section AA 18, for example, and based upon the size of the jump, determines whether a task switch has occurred, that is, a switch from processing of a first task 12 to a second task 14.

Referring now to Figure 3, further detail of CPU 10 is illustrated in block diagram format to encompass the features of the present invention. It will be appreciated that other components of CPU 10 are not shown as they are well known in the art, and such details are not necessary for describing the present invention.

CPU 10 is coupled to bus 12 for transferring information, such as data and instructions, between CPU 10 and the other units of the information handling system, such

as RAM memory 14 or ROM memory 16 which appear in Figure 1. CPU 10 includes an internal cache memory 13 as previously described. According to the invention, CPU 10 further includes a task queue 42. Task queue 42 comprises a plurality of task registers 44, 46, 48, 50, 52 for storing a task list in a chronological processing order from a current task to a next task. As used herein, chronological processing order means the order in which CPU 10 processes tasks. As discussed previously, CPU 10 processes each task a little at a time in a round robin fashion, meaning that the order of processing of the tasks can be determined by watching the tasks run in their current sequence, i.e., chronological processing order.

Each entry of task queue 42 consists of a set of registers containing a task ID 54 identifying a particular task, a previous ID 56 identifying the task processed previous to the particular task identified by task ID 54, the next ID 58 identifying the next task processed after the particular task identified by task ID 54, and a timer value 60 indicating the time required to process the task identified by task ID 54.

CPU 10 further includes a program counter 62 and a program address generator 64. CPU 10 further includes control logic 66 operably coupled to task queue 42, cache 13, program counter 62, program address generator 64, and a bus controller 68, which controls information being sent to or from CPU 10 via bus 12.

Control logic 66 monitors program counter 62 and the addresses generated by program address generator 64 to determine if a discontinuity has occurred. If a discontinuity is identified, control logic 66 checks task queue 42 to see if the task is a new task. If a new task has been identified, then a new task location is allocated by control logic 66. If the task is an existing task, a task control sequencer 66a of task control logic 66 is prompted. Task control logic 66 maintains the chronological processing order of task queue 42 and controls the loading of new data or instructions into cache 13 by following the task list and checking the cache memory to see if data or instructions are not present using a hit compare logic. If the data is not present in cache memory 13, then task control logic 66b prompts bus controller 68 to preload the data or instructions via bus 12 to cache memory 13 to thereby ensure a future cache hit during the period prior to the end of the current task.

Referring now to Figures 4A-4C, a flow diagram shows one illustrative method and operation of the present invention. The method begins in step 100. The next three steps are necessary to initialize the method. In step 102, an instruction cycle counter or timer is reset, all task registers are reset, all cycle counts are set to maximum, and the pointers are set to none. Pointers are addresses that are the starting locations of a task. In step 104, the current pointer is moved to the first available or empty task register. In step 106, the last physical instruction

address from program counter 62 is set to present physical instruction address. In step 108, the present instruction physical address is compared to the last instruction physical address. In step 110, a determination is made whether a jump in instructions larger than n bytes has occurred. Step 110 is used to detect discontinuities in the program of instructions to thereby identify tasks. If the jump is not larger than n bytes and, therefore, no discontinuity is detected, the process looks to see if the end of the present task is near by following process path 112. If the present task end is near, the process looks to see if the instructions and data for the next task are already in the cache. If the instructions and data for the next task are not already in the cache, it begins prefetching these instructions and data, accessing the last used address first.

If the jump is larger than n bytes, as determined in step 110, a discontinuity has been detected, and the process follows process path 114 to record the last instruction address and the elapsed time of the terminated task. The logic next determines if the identified task is any of the previously recognized tasks. If the identified task is a previously recognized task, the logic reestablishes links to the circular list, that is, the task list, in the task queue. If the identified task is a new task, the process provides a method for resynchronizing or reestablishing the chronological processing order arranged in the task list.

If the jump is not determined to be a discontinuity and the process follows path 112, the process then proceeds to step 116. In step 116, a determination is made whether the next pointer is active. If the next pointer is not active, the process returns to step 108. If the next pointer is active, the process proceeds to step 118 wherein the contents of the instruction cycle counter are compared to the count stored in the current task register.

In step 120, a determination is made whether the cycle count is greater than "count" minus k. If the cycle is not greater than "count" minus k, the process returns to step 108. If the cycle count is greater than count minus k, the process proceeds to step 122.

In step 122, a determination is made whether the last page address of the next task is stored in cache memory. If the last page address of the next task is not stored in cache memory, the process proceeds to step 124 wherein the last instruction page is prefetched from the next task register, and then the process returns to step 108.

If the last page address of the next task is stored in cache memory, the process proceeds from step 122 to step 126. In step 126, a determination is made whether the last page is stored in the cache memory. If the last page is not stored in the cache memory, the process proceeds to step 128, wherein the last page is prefetched from the next task register, and the process then returns to step 108. If the

last page address is not stored in cache memory, the process proceeds directly from step 126 to step 108.

If the jump is determined to be larger than n bytes in step 110, and thus a discontinuity has occurred and the process proceeds along path 114, the first step encountered is step 130. In step 130, the last physical address and contents of instruction cycle counter are stored in the current task register. The process then proceeds to step 132, wherein the last pointer is moved to the current task register. The process then proceeds to step 134, wherein the instruction cycle counter is reset. In step 136, a determination is made whether the present instruction is within m bytes of the address in any task register. Step 136 is used to determine whether the current task is already in the task list.

If the present instruction is not within m bytes of the address in any task register and, thus, the current task is not already in the task list, the process proceeds to step 138. In step 138, if the next pointer is active, the process proceeds to step 140. If the next pointer is not active, the process returns to step 108. In step 140, the contents of the next task register are erased, and the process proceeds to step 142, wherein the next pointer is reset to none. In step 144, the current pointer is moved to the first available empty task register, and the process proceeds to step 146, where the link is stored to the last task register in the task queue.

If the present instruction is within m bytes of the address in any task register and, thus, the task is already in the task list, the process proceeds to step 148, where the current pointer is moved to the identified task register. Then the process proceeds to step 150 wherein the next pointer is moved to link the current task register, and the process proceeds to step 146, where the link is stored to the last task register. After completion of step 146, the process returns to step 108.

Although the present invention and its advantages have been described in detail, it should be understood that various changes, substitutions and alterations can be made herein without departing from the spirit or scope of the invention as defined in the appended claims.

CLAIMS

1. An information handling system comprising:
 - a processor for processing a plurality of tasks in a program of instructions;
 - cache memory for storing a predetermined number of said tasks;
 - a task queue for storing a task list arranged in a chronological processing order from a current task to a next task;
 - means for assigning a task ID to each of said tasks in the task queue; and
 - means for preloading said next task into said cache memory during processing of said current task by the processor, said means for preloading comprising:
 - means for determining a number of bytes between the current task and the next task, during processing of said current task by the processor, based on the task IDs; and
 - means responsive to the means for determining for activating, during processing of said current task by the processor, the means for preloading said next task into said cache memory if said means for determining determines that the number of bytes between the current task and said next task is not smaller than n , wherein n is a positive integer large enough to indicate a task switch.
2. The information handling system, as recited in claim 1, further comprising a task controller for controlling preloading of said next tasks into said cache memory, said task controller including means for determining if said next task is stored in said cache memory before said processor completes processing of said current task, said task controller further including:
 - means for checking said task list to determine if said list contains an entry corresponding to said next task; and
 - means for comparing said task ID of said next task to said task IDs already assigned to determine if said task ID of said next task is within m bytes of said task IDs already assigned, wherein m is a positive integer smaller than n .

3. The information handling system, as recited in claim 1, wherein said means for assigning a task ID comprises a program counter and means for assigning task IDs based on said program counter.

4. The information handling system, as recited in claim 3, wherein said means for assigning further comprises:

means for monitoring said program counter for discontinuities, in the processing of said program of instructions, exceeding n bytes;

means for assigning, to a first task, a first program counter value from said program counter upon the occurrence of a first discontinuity in the processing of said program of instructions; and

means for assigning, to a second task, a second program counter value from said program counter upon the occurrence of the next discontinuity in the processing of said program of instructions occurring after said first discontinuity.

5. The information handling system, as recited in claim 4, further comprising:
a timer;

means for starting said timer upon the occurrence of said first discontinuity; and

means for stopping said timer and storing a timer value upon the occurrence of the next discontinuity occurring after said first discontinuity.

6. The information handling system, as recited in claim 1, wherein said task list comprises entries corresponding to said tasks, said entries including a timer value indicating the time required to process that particular task.

7. The information handling system, as recited in claim 6, wherein said entries further include a previous ID identifying the task processed previous to said particular task and a next ID identifying the next task processed after said particular task.

8. The information handling system, as recited in claim 1, further comprising a task controller for controlling loading of said tasks in said cache memory.

9. The information handling system, as recited in claim 1, further comprising means for determining if said next task is stored in said cache memory before said processor completes processing of said current task.

10. A method for preloading cache memory in an information handling system comprising a processor for processing a plurality of tasks in a program of instructions and cache memory for storing a predetermined number of the tasks, said method comprising the steps of:

storing a task list arranged in a chronological processing order from a current task to a next task;

identifying the tasks including assigning a task ID to each of the tasks in the task list;

determining a number of bytes between the current task and the next task, based on the task IDs, before the processor completes processing of the current task; and

determining if the number of bytes is not smaller than n , wherein n is a positive integer large enough to indicate a task switch, including preloading the next task into cache memory if the number of bytes is not smaller than n before the processor completes processing of the current task.

11. The method, as recited in claim 10, further comprising the step of assigning task IDs to the tasks from a program counter.

12. The method, as recited in claim 11, further comprising the steps of:

monitoring the program counter for discontinuities, in the processing of the program of instructions, exceeding n bytes;

assigning, to a first task, a first program counter value from the program counter upon the occurrence of a first discontinuity in the processing of the program of instructions; and

assigning, to a second task, a second program counter value from the program counter upon the occurrence of the next discontinuity, in the processing of the program of instructions, occurring after the first discontinuity.

13. The method, as recited in claim 12, further comprising the steps of:

providing a timer;

starting the time upon the occurrence of the first discontinuity; and

stopping the timer and storing a timer value upon the occurrence of the next discontinuity occurring after the first discontinuity.

14. The method, as recited in claim 10, further comprising the step of providing a task controller for controlling loading of the tasks in the cache memory.

15. The method, as recited in claim 10, wherein the task list comprises entries corresponding to the tasks, wherein the entries include a task ID identifying a particular task and a timer value indicating the time required to process the particular task.

16. The method, as recited in claim 15, wherein the entries further include a previous ID identifying the task processed previous to the particular task and a next ID identifying the next task processed after the particular task.