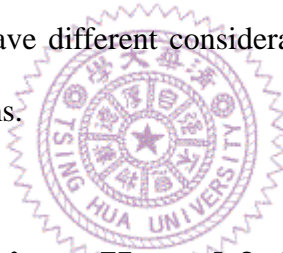


Chapter 5

Linux Load Balancing Mechanisms

Load balancing mechanisms in multiprocessor systems have two compatible objectives. One is to prevent processors from being idle while others processors still have tasks waiting to execute. The other objective is to keep the difference in numbers of ready tasks on all processors as small as possible. To accomplish these objectives, a load balancing mechanism needs to consider three problems. The first is to decide when to invoke load balancing. The second is to decide which processors need load relieve. The third is to decide how many tasks to migrate under different conditions. Linux kernel 2.6.5 and Linux kernel 2.6.10 have different considerations and use different criteria in their approaches to these problems.



5.1 Load Balancing in Linux Kernel 2.6.5

This section first presents the criteria applied by Linux kernel 2.6.5 to determine the movement of tasks among processors. The section then describes the load balancing schemes used by the 2.6.5 load balancing mechanism: They are applied when new and awoken tasks are to be placed into ready queues, when a processor becomes idle, and when rebalancing ticks occur.

5.1.1 Criteria for Task Migration

Three criteria govern when a task can be migrated to another processor. First, the task is not running. Second, the destination processor is in the set of the allowed processors of the task. Third, the task is not cache hot on its current processor.

The condition used to decide whether a task is cache hot is

$$rq->timestamp_last_tick - task->timestamp \leq cache_decay_ticks$$

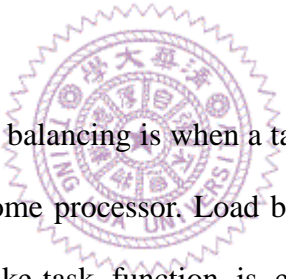
In the formula, the variable *timestamp_last_tick* is a member of the ready queue structure of the task's current processor. The kernel updates this variable at every tick. As stated in the previous chapter, every task has its own timestamp variable. The kernel records the current tick when the task was inserted into ready queue of the processor. In essence, the formula deems the task cache hot and therefore should not be migrated if it was inserted into the ready queue *cache_decay_ticks* or less time ago.

Among all the tasks on a processor that meet the three criteria above, the load balancing function, when executes on the processor, gives preference to the highest dynamic priority task in the expired priority array of that processor. The rationale behind this choice is obvious: the tasks in the expired priority array are waiting and, if not migrated, will likely not to execute soon. Tasks in the active priority array of the processor are selected for migration only when the expired priority array of the processor has no tasks. Every migrated task is put into the active priority array of the destination processor according to its dynamic priority regardless whether it was migrated from the expired priority array or active priority array of the source processor.

5.1.2 Load Balancing Involving New and Awaken Tasks

Because cache (and possibly memory) contents are no longer useful, a good time for load balancing is when a processor starts to execute a new task or a running task calls the *exec()* system call. Load balancing mechanism is invoked on the processor to migrate

the new task if the system is a NUMA and the processor has more than two tasks in the ready queue. On a SMP, or when there are two or fewer ready tasks on the current processor, the new task executes on the current processor. When migrating a new task, the load balancing function executing on the current processor searches all the nodes to find the busiest node in the system. Because nodes may contain different numbers of processors, the current processor does not just compare the total number of ready tasks in each node in this search. Rather, it compares the average numbers of ready tasks per processor in all nodes. It declares the node with the largest number of ready tasks per processor the busiest node. After finding the busiest node, the current processor searches all the processor in the node to find the idlest processor among them (i.e., the processor with the smallest number of ready tasks). The new task is migrated to the idlest processor.



Another good time for load balancing is when a task is awoken from sleep and is to be put into the ready queue of some processor. Load balancing is simple in this case. If the processor on which the wake-task function is executing is one of the allowed processors of the task being waken up, the task is waken up on that processor. Otherwise, the task is waken up on its previous processor (i.e., the processor on which the task executed just prior to it went to wake). After the processor to run the task is chosen, the dynamic priority of the task is recalculated according to its sleep_avg, the sleep_avg of the task is updated, and then the task is put into the active priority array of the chosen processor according to its new dynamic priority.

5.1.3 Load Balancing Upon Ready Queue Becoming Empty

When a running task is going to give up the processor, a ready task on the processor is picked to context switch with it. When the processor has no task in its ready queue at

that moment, load balancing is invoked. By this we mean the idle load balancing function executes to migrate a task from some other processor and thus prevents the processor from going idle.

In a NUMA system, the load balancing function searches only the processors within the same node as the current processor of the to-be-replaced task. A reason is that migrating a task from a processor in other nodes will involve the remote memory access and resultant overhead is prohibitive. In SMP system, all the processors play the same role; hence, all processors are searched.

After deciding the processors to be searched, the load balancing function finds the busiest processor among them and migrate a task from that processor. The task selected for migration is the first task that meets the criteria presented in Section 5.1.1. To avoid the situation where a task is bounced between processors because of frequent load balancing, Linux does not use the current numbers of ready tasks in the processors as the sole criterion for determining which processor is the busiest. It also makes use of historical record on their workload levels. For this purpose, each processor has an array called *pre_cpu_load[]*. There is an element in this array for every processor in the system. When load balancing takes place in a processor because the ready queue of the processor has become empty or a rebalancing tick has occurred on the processor, the load balancing function records the numbers of ready tasks on all the processors at the time in the *pre_cpu_load[]* array of the processor. Before updating this record, however, the load balancing function computes the smaller of the current number of ready tasks on each processor and the recorded number of ready tasks on that processor, and uses the result as the workload level of the processor. The load balancing function then compares the workload levels of all processors and selects as the busiest processor the processor with

the largest workload level.

5.1.4 Load Balancing Upon Arrival of Rebalancing Tick

Load balancing also takes place periodically on each processor. There are four types, called *idle rebalance*, *busy rebalance*, *idle node rebalance*, and *busy node rebalance*. They occur at periodic ticks, hereafter called *rebalancing ticks*. In Linux 2.6.5, the periods of different types of rebalance ticks are given by constants *IDLE_REBALANCE_TICK*, *BUSY_REBALANCE_TICK*, *IDLE_NODE_REBALANCE_TICK*, and *BUSY_NODE_REBALANCE_TICK*, respectively. The default values of these constants are 1, 200, 5, and 400 milliseconds, respectively. Idle rebalance and idle node rebalance happen on a processor only when the processor is idle, while busy rebalance and busy node rebalance takes place only when the processor is busy. Idle rebalance and busy rebalance are used by SMP systems or within the same node in a NUMA system. They are called *intra-node load balancing*. Idle node rebalance and busy node rebalance are used by NUMA systems only and are called *inter-node load balancing*. When rebalancing ticks for intra-node load balancing and inter-node load balancing occur at the same time on a processor, the processor does inter-node load balancing first. After completing the inter-node load balancing, it starts the intra-node load balancing.

In response to each timer tick, each processor checks whether the tick is any of the four rebalancing ticks. If the timer tick is one of rebalancing ticks and the status of the processor is such that rebalancing is to take place, the corresponding load balancing function is invoked. The status of the processor and the operations carried out for each of the four types of rebalancing are explained below. Here we use the term current processor to mean the processor on which the load balancing function executes in response to a

rebalancing tick on the processor. By the current processor doing something, we mean the load balancing function takes the action.

5.1.4.1 Intra-Node Load Balancing

For intra-node load balancing, the current processor searches all the processors in the same node as the current processor to find the busiest processor as the source processor of possible migrated tasks. As explained in Section 5.1.3, in this search, the workload level of each of the other processors used for comparisons is the minimum of the number of ready tasks recorded in the corresponding element of the `pre_cpu_load[]` and the current number of ready tasks on the processor.

If the status of current processor is idle and the rebalancing tick is `IDLE_REBALANCE_TICK`, the current processor only migrates one task from the busiest processor. Otherwise, if the current processor is busy and the rebalancing tick is `BUSY_REBALANCE_TICK`, it computes the workload difference between the source processor and itself. In this case, the current processor is the destination processor of possible migrated tasks. When computing the difference in workload levels of source and destination processors, the workload level of the destination processor is the maximum of the number of ready tasks currently on the processor and recorded value in the `pre_cpu_load[]` array, while the workload of the source processor is the minimum of the two factors. This manner in effect elevates the threshold of imbalance that triggers migration of tasks to the current processor. In this way, the Linux tries to reduce the chance of bouncing tasks between processors due to frequent load balancing.

The nominal threshold in the workload difference between the source and

destination processors is 25% of the workload level of the source processor. If the workload difference is below this threshold, the system is regarded as balanced from the point of view of current processor, and no task is migrated in this case. If the workload difference is at or above the threshold, the number of tasks migrated from the source processor to the current processor is half of the workload difference.

5.1.4.2 Inter-Node Load Balancing

As stated earlier, inter-node load balancing can take place in a NUMA system. To find the source processor in this case, the current processor first searches all the nodes to find out the busiest node. It then searches the processors in that node to find out the busiest processor in the manner described above. At an idle node rebalance tick, the current processor only migrates one task from the busiest processor. At each busy node rebalance tick, the current processor computes the difference between workload levels of source processor (i.e., the busiest processor) and itself (i.e., the destination processor) and decides whether to migrate tasks and how many tasks to migrate in the same manner as described above. The extra work for inter-node load balancing is to find the busiest node in the system. For this purpose, Linux kernel maintains for each processor an array called *pre_node_load[]*. Similar to *pre_cpu_load[]*, this array is used to record the past workload levels of all nodes in the system. Specifically, the value of *i*th element of *pre_node_load[]* gives the past workload level of the *i*th node. At the rebalance tick *t*, the workload level of *i*th node is given by

$$load(t, i) = \frac{load(t-1, i)}{2} + 10 \times \frac{node_nr_running[i]}{nr_cpus_node[i]}$$

where $load(t-1, i)$ is the value recorded in the element at the previous rebalance tick.

The value of the array element *node_nr_running[i]* gives the number of ready tasks in

node i and the value of the array element $nr_cpu_node[i]$ gives the number of processor in node i . The constant 10 is used to provide better resolution. The value of $pre_node_load[]$ is updated accordingly as a part of the load balancing operation.

The busiest node in the system at rebalance tick t is the node which has the largest workload among all nodes. In other words, $load(busiest_node, t) \geq load(k, t)$ for all k not equal to $busiest_node$. The busiest node is deemed busy enough and some tasks on the node should be migrated to the current node if

$$100 \times load(busiest_node, t) > NODE_THRESHOLD \times load(this_node, t)$$

where $this_node$ refers to the node of the current processor. The default value of $NODE_THRESHOLD$ is 125, meaning that the busiest node is busy enough if its workload level is 25% higher than the workload level of current node. If the inequality is not satisfied, no task is migrated to the current node. Otherwise, the current processor starts to find out one busiest processor within the busiest node and does the rest work as described in Section 5.1.4.1.

5.2 Scheduling Domains and CPU Groups

Scheduling domains [8, 20, 21, 22] is a way to capture and represent the architecture of the underlying multiprocessor system. Its objective is to keep the migrated tasks on processors as close as possible to their memory areas. We refer to the actual scheduling domain of a system and the data structure used to represent it both as the scheduling domain of the system.

The scheduling domain of a multiprocessor system is hierarchical and has multiple

levels. The data structures at each level are also called scheduling domains. A lowest-level scheduling domain contains all the virtual processors in the same physical processor and is called a *CPU domain*. A second-level scheduling domain contains all the processors in the same node of a NUMA system or all processors in a SMP system; it is called a *physical domain*. The highest-level scheduling domain contains all the processors in the system and is called the *node domain*.

For illustrative purpose, Figure 10 and 11 shows, respectively, the topology of a NUMA machine with hyperthreaded physical processors and the corresponding hierarchy of scheduling domains. The NUMA machine has four nodes, and each node has two hyperthreaded (dual-core) physical processors. To represent the topology of the system, the kernel uses three levels of scheduling domains. In Figure 11, each scheduling domain is depicted by a box labeled by the name of the scheduling domain. Each lower level scheduling domain has a pointer, called *parent*, to its parent scheduling domain (i.e., the higher-level scheduling domain to which it belongs).

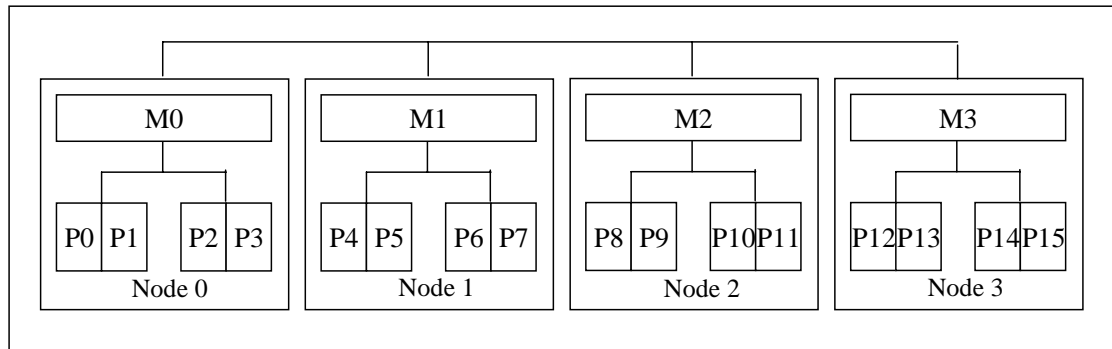


Figure 10. An example of NUMA machine with hyperthreaded physical processors

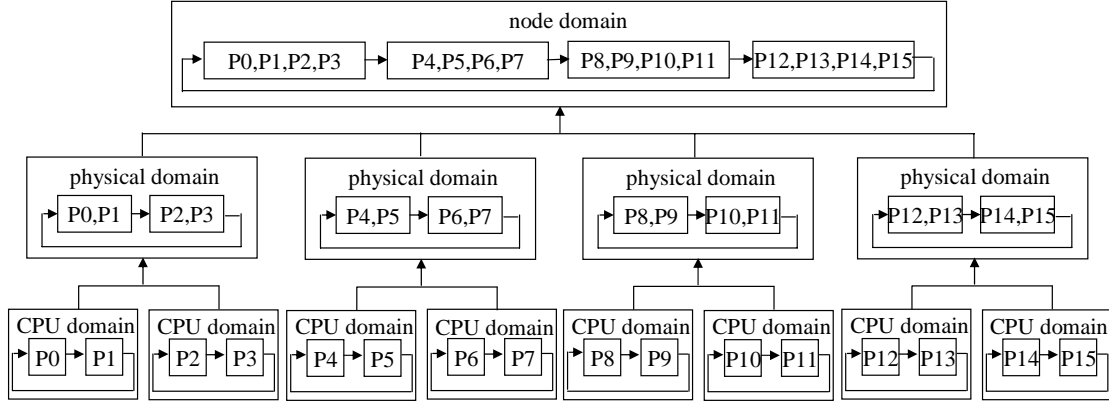


Figure 11. Corresponding scheduling domains of the NUMA system in Figure 10.

Each processor has its own data structures representing multiple levels of scheduling domains. In each scheduling domain data structure, there is a variable called *span* to specify the processors within the scheduling domain. Processors in each scheduling domain are divided into groups, called *CPU groups*. As depicted in Figure 11, CPU groups within each scheduling domain form a circular linked list. We note that the processors in each scheduling domain are exactly the same as the processors members in one of the CPU groups in its parent scheduling domain. Specifically, the information and the constants on all CPU groups in each level of scheduling domain are maintained by the kernel globally for all processors. Figure 12 shows the circular lists maintained by the kernel on CPU groups in the NUMA system in Figure 10. The order in which processors are linked is based on the hardware information. Figure 13 illustrates the data structures related to a processor, processor 6 in this example. The variable groups in each level of scheduling domain of the processor points to the proper location of the processor within the CPU group at the level.

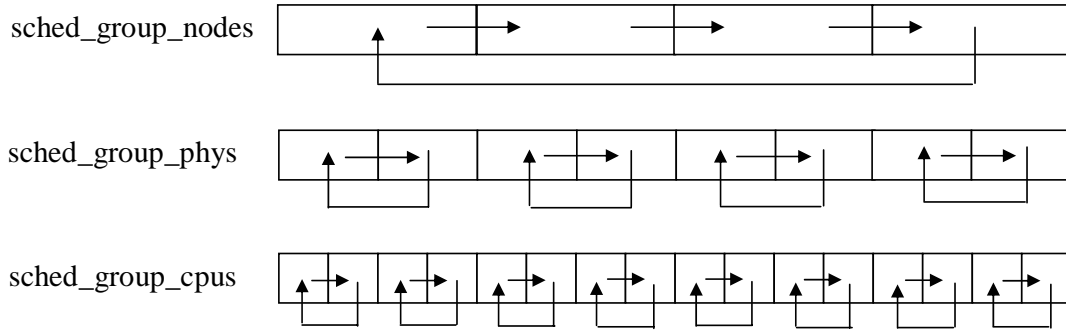


Figure 12. The global variables used to maintain the relationship of all CPU groups

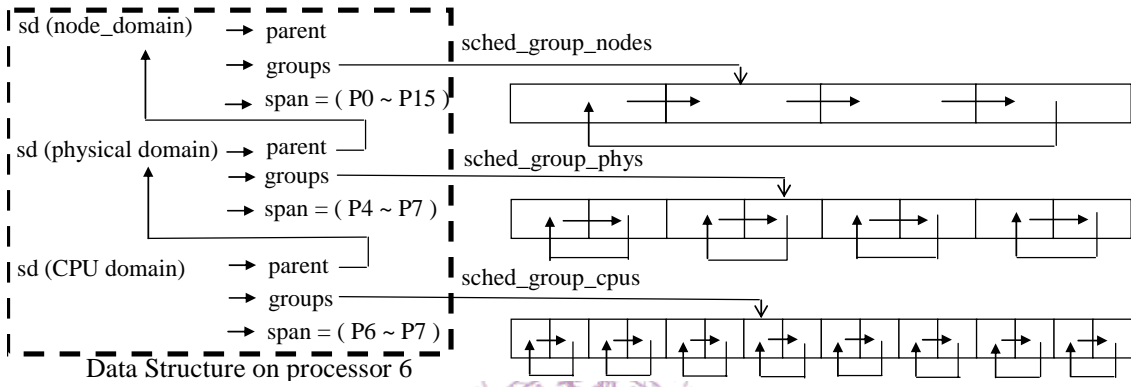


Figure 13. The relationship between the global variables of CPU groups and the scheduling domains in processor 6

Load balancing in 2.6.10 is not classified as inter-node load balancing and intra-node load balancing. Rather, the load balancing happens on a level of scheduling domains, one scheduling domain at a time, to balance the workload among the processors in the scheduling domain. When a processor searches for the busiest processor, it finds the busiest CPU group first and then searches the busiest processor in that CPU group. So, CPU groups are the basic comparing units.

Because the numbers of processors in different CPU groups in a scheduling domain may be different, the workload of each CPU group must be adjusted to represent the average workload per processor. In particular, the capability of a virtual processor in a hyperthreaded physical processor is smaller than the capability of a physical processor.

Adjusting the workload by using number of processors is not appropriate. For this reason, Linux kernel 2.6.10 uses a constant called *cpu_power* to represent the capability of each CPU group. The values of *cpu_power* in different levels of scheduling domains are different. Specifically, the value of *cpu_power* of a CPU group in a CPU domain is given by

$$cpu_power = SCHED_LOAD_SCALE$$

The constant *SCHED_LOAD_SCALE* is used to provide better resolution in many situations, and its value is 128 by default.

The value of *cpu_power* of a CPU group in a physical domain is given by

$$cpu_power = SCHED_LOAD_SCALE + SCHED_LOAD_SCALE \times \frac{(cpus_weight(cpumask) - 1)}{10}$$

The function *cpus_weight* is used to compute the number of virtual processors in each hyperthreaded physical processor. In Figure 11, the *cpu_power* of a CPU group in physical domain of a virtual processor is 1.1 units based on the formula. The value of *cpu_power* of a CPU group in node domain is the sum the values of *cpu_power* of the CPU groups in physical domains of all physical processors in the same node. For example, in Figure 11, the *cpu_power* of a CPU group in node domain is 2.2.

Besides *span*, *parent*, and *groups*, each scheduling domain has many variables and constants; they are shown in Figure 14. The constant values of *flags* are listed in the box on right hand side in Figure 14. As their name indicates, the constants specify possible policies of the scheduling domain. The usages of the policies will be explained in Section 5.3. *min_interval*, *max_interval*, and *busy_factor* define the frequency of rebalancing in

the scheduling domain. The variables *balance_interval* and *last_balance* are used to record the history of rebalancing. *imbalance_pct* is a threshold used to decide whether a busiest candidate processor is busy enough. *cache_hot_time* is used to decide whether a task is cache hot. *per_cpu_gain* is used when the system has hyperthreaded physical processors. There is a load balancing mechanism called active load balancing to remedy the situation when periodic load balancing fails too many times. *cache_nice_tries* and the variable *nr_balance_failed* are used to decide whether to invoke active load balancing. The default values of parameters in Linux kernel 2.6.10 are listed in Figure 15.

/* constants */		/* possible values of flags */	
sched_domain	*parent	SD_LOAD_BALANCE	1
sched_group	*groups	SD_BALANCE_NEWIDLE	2
cpumask_t	span	SD_BALANCE_EXEC	4
long	min_interval	SD_WAKE_IDLE	8
long	max_interval	SD_WAKE_AFFINE	16
int	busy_factor	SD_WAKE_BALANCE	32
int	cache_nice_tries	SD_SHARE_CPUPOWER	64
int	flags		
int	cache_hot_time		
int	imbalance_pct		
int	per_cpu_gain		
/* changeable variables */			
long	last_balance		
int	balance_interval		
int	nr_balance_failed		

Figure 14. The variables related to load balancing mechanisms

CPU domain :	Physical domain :	Node domain :
min_interval = 1 max_interval = 2 busy_factor = 8 cache_nice_tries = 0 flags = SD_WAKE_IDLE SD_WAKE_AFFINE SD_BALANCE_EXEC SD_BALANCE_NEWIDLE SD_SHARE_CPUPOWER cache_hot_time = 0 imbalance_pct = 110 per_cpu_gain = 25	min_interval = 1 max_interval = 4 busy_factor = 64 cache_nice_tries = 1 flags = SD_WAKE_AFFINE SD_WAKE_BALANCE SD_BALANCE_EXEC SD_BALANCE_NEWIDLE SD_LOAD_BALANCE cache_hot_time = 2500 imbalance_pct = 125 per_cpu_gain = 100	min_interval = 8 max_interval = 32 busy_factor = 32 cache_nice_tries = 1 flags = SD_WAKE_BALANCE SD_BALANCE_EXEC SD_LOAD_BALANCE cache_hot_time = 10000 imbalance_pct = 125 per_cpu_gain = 100

Figure 15. Default values of parameters in three levels of scheduling domains

5.3 Load Balancing in Linux Kernel 2.6.10

We are now ready to describe the criteria for tasks migration and the conditions triggering load balancing in Linux 2.6.10. The policy flags and the default settings used by the load balancing mechanism under each condition are explained.

5.3.1 Criteria for Task Migration

For the purpose of deciding whether task migration is warranted, the workload of each processor is computed from *current load* and *historical workload*. In this way, 2.6.5 and 2.6.10 versions are similar. They differ in the ways they compute the values of the factors: The current load of the processor is the current number of ready tasks on the processor multiplied by `SCHED_LOAD_SCALE`. The historical workload of the processor is the value of *cpu_load*, which is maintained by every processor for itself. At each time tick, the processor updates the value of *cpu_load* to be the average of its current number of ready tasks multiplied by `SCHED_LOAD_SCALE` and *cpu_load*. Generally speaking, when a processor is considered by a load balancing function to be a candidate of the source processor, its workload is the minimum of the two factors. When the processor is a candidate of the destination processor, its workload is the maximum of the two factors.

Some criteria used to decide whether a task can be migrated are similar to the criteria of Linux kernel 2.6.5. Namely, a task can be migrated if it is not running currently and is allowed to execute on the destination processor. Unlike the 2.6.5 version, which always checks to be sure that the task is not cache hot, Linux kernel 2.6.10 checks cache hot only under two conditions when making decisions on load balancing in a scheduling domain: First, the destination processor is idle currently. Second, the number

`nr_balance_failed` of times periodic load balancing fails in this scheduling domain of the source processor is less than the threshold `cache_nice_tries` for invoking active load balancing. When neither condition is true, the criterion of cache hot is not checked. Section 5.3.6 will provide further details on `nr_balance_failed` and `cache_nice_tries`.

The formula used to decide whether a task on a processor is cache hot is

$$timestamp_last_tick(rq) - last_ran(task) < cache_hot_time(sched_domain)$$

The right hand side of the inequality is `cache_hot_time` of the scheduling domain level. It specifies the estimated time period of cache hot in each scheduling domain at the level. The different values of the parameter are listed in Figure 15. The variable `timestamp_last_tick` is the current timer tick of the ready queue of the processor, and the variable `last_ran` of every task records the timer tick at which the task was stopped from running and context switched. Therefore, the left hand side of the inequality in the above formula is the difference between the current time and the latest time when the task ran. The formula says that the task is regarded as cache hot and should not be migrated if this difference is less than `cache_hot_time` of the scheduling domain.

Another important difference between load balancing in the 2.6.5 and 2.6.10 versions is the priority array into which a migrated task is inserted: When a task is migrated between processors, the task is put into the corresponding priority array on the destination processor in Linux kernel 2.6.10. In other words, Linux 2.6.10 kernel puts a task removed from the expired (active) priority array of source processor into the expired (active) priority array of the destination processor. In contrast, a migrated task is always inserted in the active priority array in Linux kernel 2.6.5.

5.3.2 Migration of New Task

As in Linux 2.6.5, when a processor starts to execute a new task or a task calls the *exec()* system call, the load balancing function is also invoked in 2.6.10. If the number of ready tasks on the current processor is less than two, the load balancing function does nothing and the task will execute on the current processor. Otherwise, the load balancing function tries to find the idlest processor and migrate the new task to that processor. To do so, the load balancing function finds the highest-level scheduling domain of the current processor with the *SD_BALANCE_EXEC* flag set. It then finds the idlest processor among all the processors that are in the span of the scheduling domain and are in the set of allowed processors of the task. Section 5.3.1 already described how workload levels of the processors are calculated for the purpose of this search in general. An exception is the workload *load(this_CPU)* of the current processor. It is equal to the workload computed as described in Section 5.3.1 plus one, to account for the new task in case the new task is executed on the processor.

After finding the idlest processor, the load balancing function uses the formula below to decide whether the processor in fact is suitable for the new task:

$$load(idlest_cpu) \times (100 + imbalance_pct / 2) < load(this_cpu) \times 100$$

If the inequality is met, the task is migrated to the idlest processor. Otherwise, the new task executes on the current processor. The *imbalance_pct* has different values in each level of scheduling domains. Consequently, the thresholds of imbalance (i.e., *load(this_CPU) – load(idlest_CPU)*) resulting in migration of the new task are different in different levels of scheduling domain. Because the current contents in cache and memory are not going to be used by the new task no matter where the new task will

execute, the overhead of load balancing is relatively small. For this reason, the `SD_BALANCE_EXEC` flag is usually set in all levels of scheduling domains. The new task can be migrated to any of its allowed processors, including a processor in another node in NUMA systems.

5.3.3 Load Balancing Involving Awaken Tasks

Load balancing mechanism is invoked when a task is being awoken. The current processor in this case is the processor on which wake-task function is executing. The function chooses the *possible next processor* for the to-be-awoken task according to the `SD_WAKE_AFFINE` and `SD_WAKE_BALANCE` flags and the current workloads of all processors. By next processor, we mean the processor on which the to-be-awoken task will execute. If the `SD_WAKE_IDLE` flag is set, the kernel chooses the *actual next processor* on which the awoken task will actually execute; the actual next processor may or may not be the possible next processor depending on the criteria described below. If the flag is not set, the actual next processor is the possible next processor. After choosing the actual next processor, the awoken task is migrated to that processor: Its dynamic priority is recalculated, and it is inserted into the active priority array of the processor according to its new dynamic priority.

The choice for the possible next processor is between the current processor and the previous processor. By previous processor, we mean the processor on which the task executed just prior to going to wait. If the current processor happens to be the previous processor of the (to-be-awoken) task, the processor is chosen as the possible next processor. Otherwise, the formula below is used to decide whether the previous processor is a better choice of the possible next processor.

$$load(\text{previou}) < \frac{SCHED_LOAD_SCALE}{2} \ \&\& \ load(\text{current}) > \frac{SCHED_LOAD_SCALE}{2}$$

If the workload levels of both the previous processor and the current processor satisfy the inequalities, the previous processor is the possible next processor. The load of the previous processor is computed as the minimum of its current load and historical load, and the load of the current processor is computed as the maximum of the two factors. If only the left inequality is satisfied, the previous processor is almost idle. It is chosen to be the possible next processor if it is less loaded than the current processor.

If workloads of the previous processor and current processor do not satisfy the inequalities in the formula, the current processor does several additional checks to decide whether to choose itself as the possible next processor. Some constraints of the scheduling domain containing both the previous processor and the current processor must be met. These constraints are defined by the `SD_WAKE_AFFINE` and `SD_WAKE_BALANCE` flags. If the scheduling domain has constraints and the constraints cannot be met, the possible next processor is the current processor. If the scheduling domain has no constraints or if all the constraints in all scheduling domains can be met, the possible next processor is the previous processor.

If the `SD_WAKE_AFFINE` flag of a scheduling domain is set, the task should be cache hot in this scheduling domain. The formula used to determine whether the task is cache hot is the one presented in Section 5.3.1. If the `SD_WAKE_BALANCE` flag of a scheduling domain is set, the workloads of the current processor and the previous processor should be more or less balanced. In other words their workloads do not satisfy the inequality below.

$$\left(imbalance_pct + \left(\frac{imbalance_pct - 100}{2} \right) \right) \times load(current) \leq 100 \times load(previous)$$

When the inequality holds, the workloads of the processors are regarded as imbalanced. As stated earlier, the values of `imbalance_pct` are different in different levels of scheduling domains. Hence the threshold differences in workloads satisfying this inequality in different scheduling domains are also different.

After choosing the possible next processor, the next step is to select the actual next processor. If the `SD_WAKE_IDLE` flag of the lowest-level scheduling domain of the possible next processor is not set, the possible next processor is chosen to be the actual next processor. If the flag is set, the current processor tries to find an idle processor in the scheduling domain to be the actual next processor. Preference is given to the possible next processor. So, if the possible next processor happens to be idle when the current processor tries to find an idle processor, the possible next processor is chosen to be the actual next processor. If the possible next processor is not idle, the current processor searches all processors in the scheduling domain, starting from the processor with smallest serial number, to find an idle processor. If no idle processor is found, the possible next processor is chosen to be the actual next processor.

5.3.4 Load Balancing on Idle Processor

When a task is giving up the processor or starts to wait and the current processor of the task has no ready task to replace it, the processor invokes a load balancing function to search and migrate a task to the processor from the busiest processor in the busiest CPU group in a scheduling domain. The search for the task starts from the lowest-level scheduling domain and expands to the entire domain hierarchy until a scheduling domain

with the *SD_BALANCE_NEWIDLE* flag set is found. If the current processor can migrate a task from the busiest processor in the busiest CPU group in that scheduling domain, the load balancing function migrates the task and exits. Otherwise, it looks for another scheduling domain with the *SD_BALANCE_NEWIDLE* flag set and tries to migrate a task from that domain. If the current processor fails to migrate a task to execute on itself after searching all scheduling domains, load balancing effort is abandoned. Because the system does not want to suffer much overhead doing load balancing to prevent a processor from becoming idle, the *SD_BALANCE_NEWIDLE* flag is usually not set in the node domain.

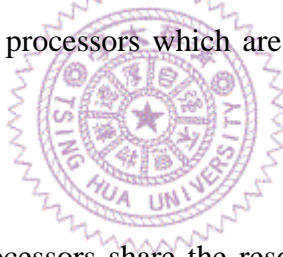
To find the busiest CPU group in a scheduling domain, the (the load balancing function executing on) current processor compares all the CPU groups in the scheduling domain. The workload of each CPU group used for this comparison is the total workload of all processors in the CPU group multiplied by *SCHED_LOAD_SCALE* and divided by the *cpu_power* of the CPU group. If the current processor is not in the CPU group, the group is a candidate of the source CPU group which sends tasks. The workload of each processor in a candidate source CPU group is the minimum of its current workload and historical workload. The workload of each processor in the same CPU group as the current processor is the maximum of the two factors.

After finding the busiest CPU groups, the current processor compares the workload levels of all the processors in the group to find the busiest processor among them. Because all the processors are candidate of source processors, the workload level of every processor is the minimum of its current workload and historical workload. If the busiest processor also does not have ready tasks, the load balancing in this scheduling domain terminates. Otherwise, the current processor migrates a task from the busiest processor to

execute on itself and context switch with the task yielding the processor.

5.3.5 Load Balancing on Hyperthreaded Machines

Load balancing in hyperthreaded machine is sometimes handled specially by Linux kernel 2.6.10. The special treatment applies only when the *SD_SHARE_CPUPower* flag of the CPU domain is set. As it will become evident in the next paragraph, it is possible for a processor to be idle while still has ready tasks. At each timer tick, if a processor is idle and still has ready tasks, the processor reschedules to decide whether to execute the next task selected to run. When a processor is idle and has no ready task, the processor invokes the load balancing function presented in Section 5.3.4 to migrate a task from the busiest processor to execute on itself. If the processor cannot migrate a task from other processors, it notifies all sibling processors which are idle and still have ready tasks to reschedule.



Because all the virtual processors share the resource of a hyperthreaded physical processor, the current processor considers the conditions of its siblings in the same physical processor when deciding whether to start the next task. The next task on the current processor is suspended under two conditions: First, the next task selected to run on the current processor is not a real-time task, but the task currently running on the sibling of the processor is a real-time task. Second, the remaining time slice of the task currently running on the sibling of the processor satisfies the inequality below,

$$smt_curr \rightarrow timeslice \times \frac{100 - per_cpu_gain}{100} > time_slice(p)$$

In this inequality, *smt_curr* is the task running on the sibling processor. *per_cpu_gain* is a constant of the scheduling domain; its default value is listed in Figure 15. *time_slice()* is a

function used to compute the default time slice of a task, and p is the next task on current processor. For example, if the value of `per_cpu_gain` is 25 and the default time slice of task p is 120 ms, the inequality is satisfied when the remaining time slice of the task running on the sibling processor is bigger than 160 ms.

If the task running on the sibling processor is not a real-time task, and the next task selected to run on the current processor is a real-time task or the remaining time slice of the next task satisfies the inequality below, the next task on the current processor starts to execute and the current processor notifies the sibling processor to do reschedule.

$$p \rightarrow \text{timeslice} \times \frac{100 - \text{per_cpu_gain}}{100} > \text{time_slice}(\text{smt_curr})$$

This formula has similar meaning as the last formula. A special case is when the next task selected to run on the current processor and the task running on the sibling processor are both real-time tasks, then both tasks execute on the sibling virtual processors.

5.3.6 Periodic Load Balancing and Active Load Balancing

The periodic load balancing function is invoked only in the scheduling domains with the `SD_LOAD_BALANCE` flag set. We use the term current processor to mean the processor on which the load balancing function executes when a rebalance tick occurs to check whether to do rebalance on the processor: At each timer tick, the current processor starts from the lowest-level scheduling domain and searches the domain hierarchy to decide whether the rebalancing is needed in each level of scheduling domains with set `SD_LOAD_BALANCE` flag. In Linux kernel 2.6.10, the length of rebalance period is called `balance_interval` and is used to compute the rebalance criterion. The upper bound and lower bound of `balance_interval` are `max_interval` and `min_interval`, respectively.

max_interval and min_interval have different values in different levels of scheduling domains.

The rebalance criterion of a scheduling domain is computed from the balance_interval of the domain and the condition of the current processor: If the processor is idle, the rebalance criterion is equal to balance_interval. If the current processor is busy, the rebalance criterion is equal to balance_interval times the busy_factor of this scheduling domain. busy_factor is larger than one; rebalancing is done less frequently on the current processor when the processor is busy.

In each scheduling domain of each level, there is a variable called last_balance. It records the tick at which the processor did the rebalancing in the scheduling domain the last time. At each timer tick, the current processor computes the difference between the current time and last_balance of the scheduling domain. It then computes the sum of the difference plus the offset computed by *CPU_OFFSET* macro below.

$$CPU_OFFSET(CPU) = \left(HZ \times \frac{CPU}{NR_CPUS} \right)$$

CPU is the serial number of the processor, *HZ* is the total timer ticks per second (i.e., 1000 ticks per second in Linux kernel 2.6.10), and the *NR_CPUS* is the total number of processors in the system. The *CPU_OFFSET* macro returns different values according to the serial numbers of different processors. As a result of adding the offset before the comparison, processors are likely to do rebalancing at different ticks, rather than doing rebalance at the same time.

If the time difference plus offset is larger than the rebalance criterion, the current

processor invokes the rebalancing function of the scheduling domain. After doing load balancing, the current processor updates the `last_balance` of the scheduling domain to be the previous `last_balance` plus the rebalance criterion. If the difference is not greater than rebalance criterion, the rebalancing function is not invoked in the scheduling domain, and the current processor continues to check higher-level scheduling domains.

While executing the load balancing function, the current processor first finds the busiest processor in the busiest CPU group in the scheduling domain in the manner described in Section 5.3.4. The current processor then migrates tasks from the busiest processor. To decide how many tasks should be migrated, the current processor computes the average workload per processor in the scheduling domain. The average workload is defined as the total workload of all processors in the scheduling domain multiplied by `SCHED_LOAD_SCALE` and divided by the total `cpu_power` of all CPU groups in the scheduling domain. The workload of each processor in the same CPU group as the current processor is the maximum of its current workload and historical workload. The workload of each processor in other CPU groups is the minimum of its two factors. The number of tasks to be migrated differs under the four conditions:

- (1) The busiest CPU group cannot be found.
- (2) The busiest CPU group is found, but the workload level of the busiest CPU group is less than the workload level of the CPU group containing the current processor.
- (3) The busiest CPU group is found, and the workload level of the busiest CPU group is larger than the workload level of the CPU group containing the current processor. Moreover, the workload levels of the busiest CPU group and the CPU group containing the current processor satisfy the condition below.

$$(this_load \geq avg_load) \parallel (100 \times max_load \leq imbalance_pct \times this_load)$$

this_load is the average workload of all processors in the CPU group containing the current processor. The *avg_load* is the average workload of all processors in the scheduling domain. The *max_load* is the maximum workload of all processors in the busiest CPU group in the scheduling domain.

- (4) The busiest CPU group is found, and the workload level of the busiest CPU group is larger than the workload level of the CPU group containing the current processor. Moreover, the workload levels of the busiest CPU group and the CPU group containing the current processor do not satisfy the condition presented in (3).

The load balancing function in the scheduling domain terminates when condition (1) is true. Under conditions (2) and (3), the number of tasks migrated is one, if the current processor is idle. If the current processor is not idle, the workload among the processors in the scheduling domain is regarded as balanced, and no tasks is to be migrated. Under condition (4), the number of tasks to be migrated is computed by the formula below.

$$\frac{\min(max_load - avg_load, avg_load - this_load) \times \min(busiest_pwr, this_pwr)}{SCHED_LOAD_SCALE \times SCHED_LOAD_SCALE}$$

busiest_pwr is the *cpu_power* of the busiest CPU group, and *this_pwr* is the *cpu_power* of the CPU group containing the current processor. The load balancing function moves as few tasks as possible to avoid tasks bouncing between processors. The load balancing function does not want to make the workload of current processor above the average workload, nor it wants to make the workload of the busiest processor below the average workload.

When the busiest CPU group or the busiest processor in the busiest CPU group cannot be found, the workload in the scheduling domain is regarded as balanced. In this case, the current processor reduces the frequency of rebalancing in the scheduling domain by multiplying the `balance_interval` by 2. In other cases, the workload in the scheduling domain is regarded as imbalanced, and the number of tasks to be migrated is larger than or equal to 1. If at least a task has been migrated as the result of rebalancing, the `balance_interval` is reset as `min_interval` and the number `nr_balance_failed` of rebalancing failures is set to 0. If no task was migrated, the rebalancing is regarded as a failure. The current processor increases the `balance_interval` by 1 and increases `nr_balance_failed` by 1. Each scheduling domain has a constant, `cache_nice_tries`, that defines the tolerable number of rebalancing failures in the scheduling domain. When `nr_balance_failed` becomes larger than `cache_nice_tried` plus 2, the current processor notifies the busiest processor to invoke the active load balancing function and reset `nr_balance_failed` to 0.

Almost all the load balancing functions in Linux kernel are *pull* schemes: The less loaded processor tries to migrate tasks from the busiest processor. Active load balancing is a *push* scheme. Accordingly, the busiest processor actively migrates tasks to idle processors. The busiest processor starts from the lowest-level scheduling domain and searches all processors in each CPU group in each level of scheduling domains. When the busiest processor finds an idle physical processor, it migrates a task to the idle physical processor and continues to check the next processor. After the busiest processor checks all the processors in all CPU groups in the scheduling domain, it checks the higher-level scheduling domains. The busiest processor continues to push tasks until it has less than two ready tasks or when it finds a scheduling domain without the `SD_LOAD_BALANCE` flag set.