USENIX Association

Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

Speeding Up Kernel Scheduler by Reducing Cache Misses - Effects of cache coloring for a task structure -

Shuji YAMAMURA, Akira HIRAI, Mitsuru SATO,

Masao YAMAMOTO, Akira NARUSE, and Kouichi KUMON

Fujitsu Laboratories LTD
4-1-1, Kami-Kodanaka, Nakahara-ku, Kawasaki-shi, Kanagawa-pref, Japan 211-8588
{yamamura, ahirai, msato, masao, naruse, kumon}@flab.fujitsu.co.jp

Abstract

In this paper, we propose and evaluate the experimental implementation of cache coloring for task structures on the Linux kernel 2.4.x. We analyzed the behavior of the scheduler from the viewpoint of memory architecture and found that severe cache conflicts occur in a specific cache line. To solve this issue, we applied cache coloring scheme to task structures. Until now, task structures in Linux kernel could not be moved freely in main memory. We noticed, however, that a small modification can enable the coloring. Under heavy workloads, this technique can dramatically reduce cache misses while traversing the run queue that contains a lot of runnable processes. For the evaluation, we used 4-way and 8-way IA server machines. Web server execution and Chat micro benchmark of scheduler intensive benchmarks were used for the measurement. The experimental results showed that the Web server performance achieves a maximum of 23.3% improvement compared to the standard kernel. In the Chat micro benchmark, the message throughput improves a maximum of 89.6% compared to the standard kernel. Furthermore, our coloring technique gives better scalability as the number of processors increases on a SMP system since the lock contention which protects the run queue is reduced.

1 Introduction

Linux has been widely used for enterprise applications, such as web servers and DBMS. In these environments, Linux is expected to be stable and scalable on SMP systems. However, the current Linux (2.4.x) scheduler design contains a well-known performance problem: a single global run queue protected a spin lock. The scheduler has to traverse the entire run queue to select an appropriate process to run. Therefore as the number of runnable processes increases, traversal time becomes longer. On a SMP system, long traversal time causes both the lock hold time and the lock contention to increase. This limits the scalability of the Linux system. A lot of efforts have been made to improve the scheduler performance by LSE (Linux Scalability Effort) [1]. But no effort focusing on cache and memory architecture has been made yet.

In this study, we observed and analyzed the scheduler behavior from a memory architectural viewpoint using a hardware system bus monitor on the IA32 platform. And we found that a large number of cache misses occur during the run queue traversal. To reduce these cache misses and realize faster traversing, we introduce a new solution into the Linux kernel 2.4.x: *cache coloring for a task structure*.

In this paper, we propose an experimental implementation for coloring which provides large performance gains under heavy workloads. This scheme can reduce cache misses during the run queue traversal and speed up the process scheduling. Furthermore, this method can reduce both the run queue lock hold time and the lock contention on a SMP system.

The rest of this paper is organized as follows: Section 2 summarizes the current scheduler issue in terms of cache efficiency by using a hardware memory bus tracer. Section 3 describes our implementation of cache coloring for a task structure. Section 4 evaluates the performance of our implementation and gives detailed analysis of bus transaction statistics. Section 5 presents the scalability improvement compared with the standard kernel. Section 6 describes related work, and finally we draw conclusions in Section 7.

2 Current Scheduler Issue

The current Linux scheduler has the following two characteristics.

First is a single run queue which is protected by a run queue lock. The run queue is organized as a single double-linked list for all runnable tasks in the system. The scheduler traverses the entire run queue to locate the most deserving process, while holding the run queue lock to ensure exclusive access. As the number of processors increases, the lock contention increases.

Second is the alignment of task structures in physical memory space. The task structure, which contains the process information, is always aligned on an 8KB boundary in physical address space. The problem of this implementation is that each variable in the task structure is always stored at the same offset address of a page frame. This leads to a strong possibility of cache line conflicts for each variable while the scheduler is traversing the long single run queue. At the same time, a large number of cache misses occur.

We observed the memory bus transactions by using hardware bus tracer *GATES* [2]. GATES (General purpose memory Access TracE System) can capture memory transactions on the memory bus of a shared memory multiprocessor system during program execution. Figure 1 shows the statistics of memory traffic on a shared memory bus captured by GATES. In this observation, 256 apache web server processes (*httpd*) are running on an 4-way Pentium Pro 200 MHz server with 512KB L2-cache each. The vertical axis and the horizontal axis represent the number of transactions per single web request and the page offset address, respectively. As shown in this figure, we can see that a tremendous number of transactions occur on the offset address 0x30. This cache line contains the useful variables of a task structure for scheduling. Figure 2 shows scheduler related variables in a task structure of the 2.4.4 kernel. The 32 bytes block surrounded by the thick line contains the variables which is referred to calculate the priority of each process. And this block also contains the pointer (`run_list.next`) to the next task structure. The scheduler traverses all of task structures on the run queue by referring this pointer. During the run queue traversal, the block of each task structure is successively transfered to the few cache lines corresponding to the page offset address from `0x20` to `0x3f`. This causes a lot of cache conflicts on these cache lines, and a large amount of bus transactions are generated seen as Figure 1.

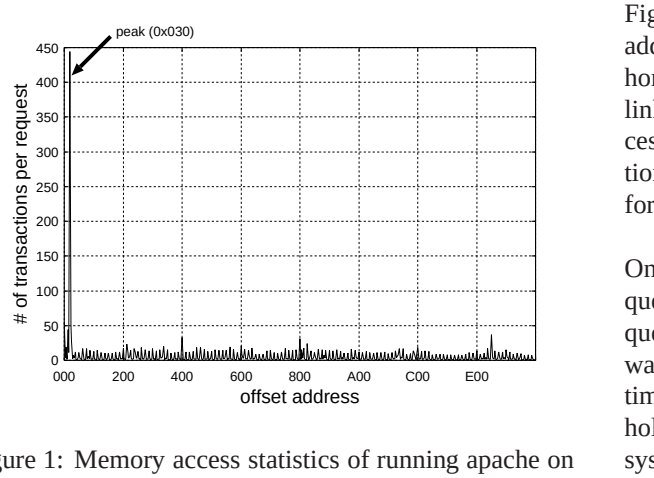


Figure 1: Memory access statistics of running apache on the 4-way Pentium Pro server

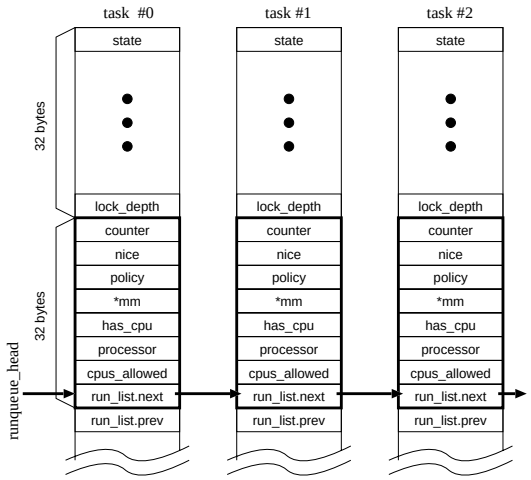


Figure 2: The task structure and the run queue structure

Figure 3 shows the number of transactions to the offset address 0x30 and the web transaction performance. The horizontal axis represents the number of httpd processes linked to the run queue. As the number of httpd processes on the run queue increases, the number of transactions significantly increases and the web transaction performance correspondingly decreases.

On SMP systems, the run queue is protected by the run queue lock. When traversing the run queue causes frequent cache misses, the time the processor must spend waiting to fill the cache just adds to the overall traversal time. As the traversal time becomes longer, so does the hold time for the run queue lock. On a multiprocessor system, this contention for cache lines translates into increased lock contention, which becomes a bottleneck for scaling.

```
static inline struct task_struct * get_current(void)
{
  struct task_struct *current;
  __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
  return current;
}
```

(a) original get_current()

```
static inline struct task_struct * get_current(void)
{
  struct task_struct *current;
  __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
  (unsigned long)current |= ((unsigned long)current >> 10) & 0x00000060;
  return current;
}
```

(b) modified get_current()

Figure 4: modified get_current()

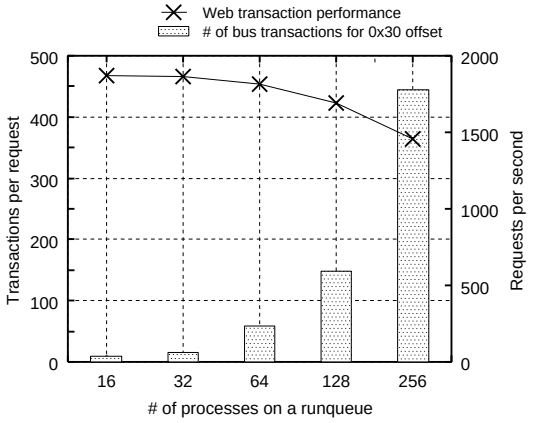


Figure 3: The number of memory bus transactions for the offset address 0x30 and the web server performance

3 Our solution - cache coloring for a task structure -

3.1 Implementation issue for coloring

In general, *cache coloring* [3, 4] is used to address the cache line conflict problem. We expect that the method is able to reduce the cache misses during run queue traversal.

However, cache coloring for the task structure in the Linux kernel seems difficult to implement, because the Linux kernel coding assumes it to be placed on a specific boundary. For example, a procedure to acquire a variable X in a task structure is as follows:

1. Calling `get_current()` (Figure 4 (a)), the function returns the logical-and value of `%esp` (current stack pointer) and `0xffffe000`. This is the 8KB boundary of the `%esp` towards to the address 0.
2. Adding the offset of the requested variable `X` to the above value.
3. Accessing the process specific variable `X` using the address.

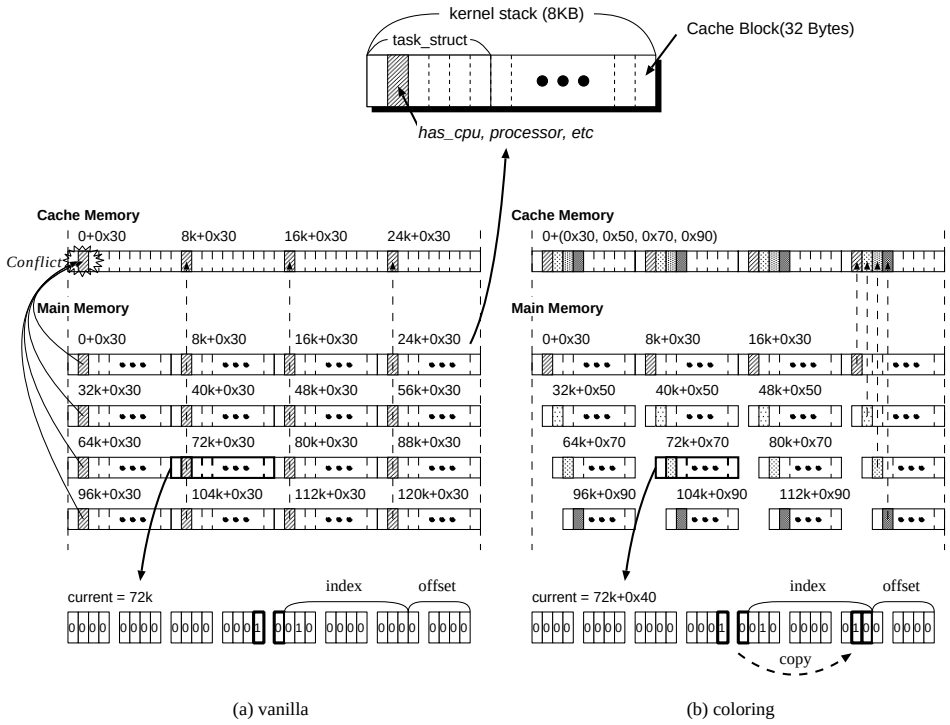
The kernel stack and task structure share an 8KB allocation of memory aligned on an 8KB boundary. The function `get_current()` assumes that the task structure is always at offset 0 within this block. It zeroes out the low-order bits of the stack pointer to produce a pointer to the task structure.

Because of this implementation, a task structure cannot be shifted freely and the task structure coloring has not been implemented yet.

3.2 Our implementation for coloring

Although the task structure had the above constraints, we noticed that a small modification to the `get_current()` function can enable the coloring.

Figure 4 (b) shows a new `get_current()`. We only inserted the single line in bold face to the original. The



(a) vanilla (b) coloring

Figure 5: An example of coloring (four colorings)

new `get_current()` returns a slightly different base address, which is shifted in multiples of cache line size. We achieved low overhead by adding only a few bit operations to the original `get_current()`. This is very important because this function is frequently called in the system. Note that this implementation does not require any changes to the core routines of the standard kernel scheduler.

Figure 5 illustrates the implementation of coloring for a task structure. To be easy to understand, we assume that the cache size is 32KB (which probably is smaller than a real L2 cache size) maintained by a direct mapping scheme. The kernel stacks (the task structure objects) shall be contiguously arranged on a main memory (in fact, they are not necessarily arranged continuously in this way, but are aligned on the 8KB boundary on a main memory). The current kernel stack is surrounded by the thick line in this figure.

Figure 5 (a) shows the case of the standard kernel (we call this kernel *vanilla*). The shaded cache blocks are missed frequently. Their memory address is given above each block. As shown in this figure, since each kernel stack is aligned on an 8KB boundary, task structures which are placed at 32KB distance addresses on a main memory are mapped to the same cache lines. If the scheduler continuously accesses these task structures (such variables as `has_cpu`), cache conflicts occur. For example, grayed data in four kernel stacks of 8KB, 40KB, 72KB, and 104KB in a main memory are transferred to the same line in a cache memory. Therefore, if the scheduler continuously accesses these four task structures, conflicts would occur at the shaded cache lines labeled “8k+0x30”.

On the other hand, Figure 5 (b) shows the case of a kernel using four-coloring. When the coloring kernel calls the modified `get_current()`, it returns 72K+0x40. This is the base address at a 32*2 bytes (equals the size of two cache lines) different offset. In the practical implementation, the modified `get_current()` generates a new `current` value by hashing it with its upper two bits. In this way, when the scheduler continuously accesses the task structures, cache conflicts never occur.

To apply our implementation to different cache configurations, we have to be careful in our choice of address bits to copy from the masked stack pointer to create the colorized pointer to the current task structure. For 2^n colors, we would choose for copying the n bits immediately higher than the bit b satisfying the following equation.

$$b = \log_2 \frac{cache\ size}{cache\ associativity} \qquad (1)$$

Suppose the cache configuration is 512KB with four-way set associative, the address difference of the two blocks on a same line is always multiple of 128KB. This means the lower 17 bits of the two block addresses are always same and using these bits cannot contribute coloring. Thus, to make the cache coloring effective, bits higher than the 17th bit should be used to colorize the structure.

And, we can control the number of colorings by the number of bits to be copied. For example, in order to implement eight colorings, we should use three bits for hashing task structures (replacing "`0x60`" by "`0xe0`" in Figure 4 (b)).

The coloring kernel needs modifications in kernel source files other than the one containing `get_current()`. However, the patch file is only 225 lines.

3.3 Negative effect of cache coloring

While the coloring performs effectively so as to reduce cache misses during traversing a run queue, there may be a negative effect on performance. We will explain it with Figure 5.

As shown in Figure 5 (a), many conflicts occur at four cache lines on vanilla kernel. On the other hand, as shown in Figure 5 (b), the number of cache lines used for variables, such as `has_cpu`, are increased to 16. In this case, there is a possibility that the data stored on a colored line before coloring are unfortunately pushed out. As a result, the total amount of bus traffic may increase and lower the overall system performance.

In the latter section for performance evaluation, we will study the effect of our coloring scheme and also analyze such negative influence.

4 Performance Evaluation

The rest of this paper gives quantitative evaluation results of the coloring kernel.

4.1 Methods

To verify the cache coloring improvement, we chose a web server program as a scheduler intensive benchmark, and evaluated the performance improvement. In the experiment, Apache 1.3.19 is used as the web server, and two Linux kernels are used for comparison: vanilla Linux 2.4.4 kernel and modified kernel with using 32 colors for the task structure. The web clients ran WebBench 3.0 [6] to generate web requests for the server. The specification of the server machine is shown in Table 1. We used three different sizes of L2 cache for evaluating the coloring effects, and all of them have four-way set associative caches. During the benchmark execution, 256 client threads were running on 28 standard PCs and they simultaneously sent requests to the server machine. On the server-side, 256 to a maximum of 1024 server threads (httpd) [1] were runnable and were linked on the run queue of the server system.

CPU	Pentium Pro 200MHz $\times$ 4
Memory	640MB
L2 cache	256KB, 512KB, 1024KB 4 way set associative
NIC	Intel EtherExpress Pro/100 $\times$ 4

Table 1: The specification of the server machine

4.2 Results

At first, we show the performance of the coloring scheme with three different cache configurations, in Figure 6. The horizontal axis is the cache size, and the vertical axis is the performance improvement ratio based on the vanilla kernel. In the experiment, the maximum number of httpd processes on a server is set to 256, 512 and 1024, and the number of colors was fixed to 32. We show performance data of the coloring scheme for three different cache configurations.

Second, to measure the coloring effect directly, the number of bus transactions on the memory bus was measured with 1024 httpd processes. In the Figure 7, bus transactions are sorted in the following two categories:

colored lines: By coloring, as illustrated in Section 3.2, the run queue list elements are distributed into several cache lines. The total number of bus transactions for these cache lines is expected to decrease due to the coloring.

other lines: The bus transactions of the cache lines other than the above.

[1] We changed `MaxClients` parameter value in the configuration file of Apache.

Finally, we measured the number of L2 cache misses and total L2 read counts during the execution of `list_for_each` loop using the performance monitoring counters built into the Pentium processor. This loop traverses the global run queue. Table 2 gives the average of L2 cache miss ratio with three cache sizes. The miss ratio is calculated by the following equation:

$$L2\ cache\ miss\ ratio = \frac{L2\ cache\ miss\ counts}{L2\ read\ counts} \quad (2)$$

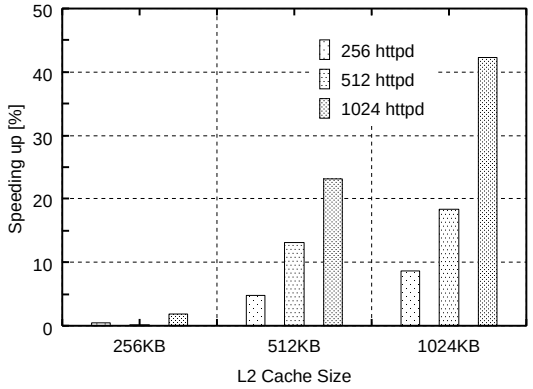


Figure 6: Performance improvement compared to vanilla kernel

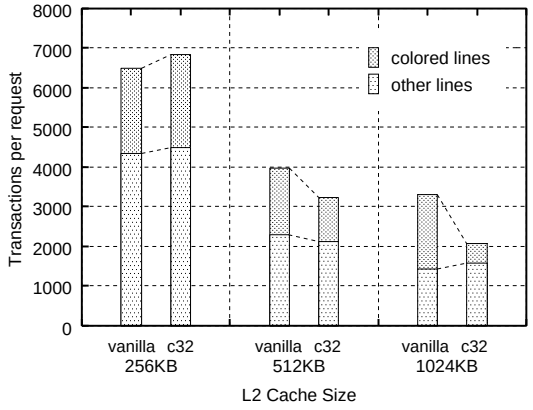


Figure 7: The number of memory bus transactions (1024 httpd processes)

At an L2 cache size of 256KB, we see little or no performance improvement from colorizing (Fig 6). In fact, the number of memory bus transactions per request increased slightly (Fig 7). This is the negative effect of coloring described in Section 3.3. In the case of 32 colorings, $(256KB/8KB) \times 32colors = 1024$ processes ideally should be stored in a cache memory. As shown in Table 2, however, the ratio of L2 cache miss in `list_for_each` loop, achieves only 17%-age points reduction. This is because all of task structures are not always contiguously allocated in main memory. Therefore the theoretically possible number of task structures was not stored in cache memory. Since the effect of coloring is partially canceled by the negative effect of increasing bus transactions, there was less performance improvement in the case of 256KB L2 cache.

	256 KB	512 KB	1024 KB
vanilla	99.7%	99.6%	99.5%
32 coloring	82.7%	35.8%	8.2%

Table 2: L2 cache miss ratio during searching the run queue (1024 httpd processes)

On the other hand, in Figure 6, significant improvement is observed with both 512KB and 1024KB L2 cache size. The 32 coloring kernel achieves a maximum of 23.1% and 42.3% performance improvement compared to the vanilla one. Also the cache miss ratio is dramatically reduced. Without coloring, the cache miss rate was 99.5% for L2 accesses. Using 32 colors, the L2 cache miss rate decreased to 35.8% and 8.2% with 512KB and 1024KB L2 cache, respectively. As a result, the number of bus transactions for colored cache lines decreases by 33.3% and 73.0% compared to the vanilla kernel (Fig. 7).

As shown above, the cache coloring is more effective for a large cache size, because the large cache area can be utilized to reduce cache conflicts for the run queue.

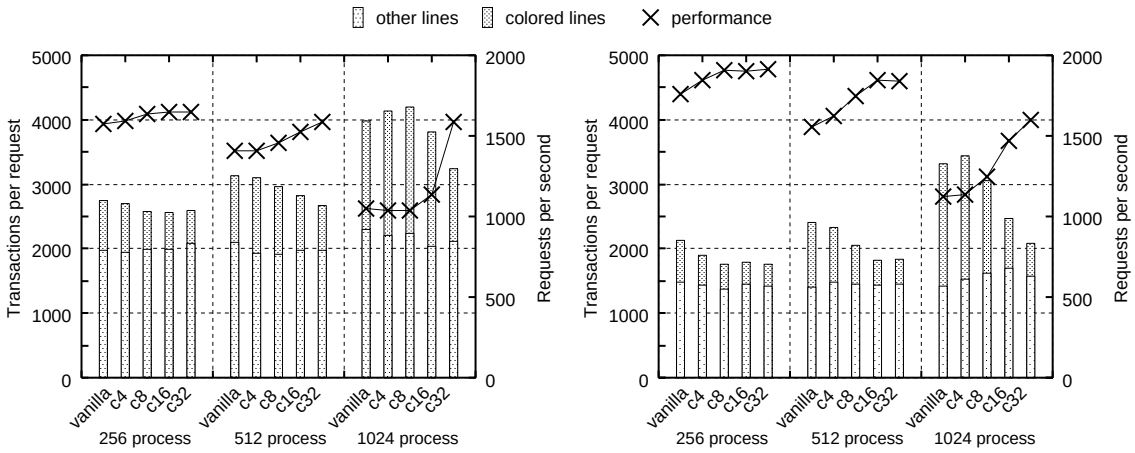
4.3 The relationship between the number of processes and the number of colorings

In this section, we show the performance difference as the number of colors changes.

By n coloring, the single severely conflicting line in a current scheduler is distributed into n lines. To minimize total cache conflicts, the number of colors should satisfy the following equation:

$$c \geq \frac{p}{(s/8[KB])} \quad (3)$$

In this equation, c, p, and s are the number of colors, the number of processes, and the cache size in KB, respectively. 8KB is the Linux kernel stack size. Ideally, a cache memory can store $s/8$ kernel stacks. For example, in the



(a) 512KB L2 cache

(b) 1024KB L2 cache

Figure 8: The number of bus transactions and performance changing the number of colorings

case of 1024 processes running on the 512KB cache system, 16 or more colors can prevent conflicts.

To verify the model, we measured the performance varying both the number of colors (4, 8, 16, and 32) and the number of httpd processes (256, 512, and 1024). We used 512KB and 1024KB L2 cache system for the experiment. Figure 8 shows the result of web processing performance and the number of bus transactions. In these graphs, c n denotes n-coloring kernel.

At first, we discuss the case of 512KB cache size. Using the equation (3), c = 4, 8, and 16 colors should be required as a minimum in the case of 256, 512 and 1024 running processes, respectively. As shown in Figure 8 (a), the result almost confirms this requirement: When the number of the coloring is less than the above requirement, the performance is not improved. As the number of coloring increase above the value c of the equation (3), the number of bus transactions for the colored line sharply decreases and the performance improves.

The coloring effect on 1024 KB cache size showed a similar relation between the performance and the number of colorings. 2, 4, and 8 colorings should be used when 256, 512, and 1024 processes are running on a system, respectively. The graph in Figure 8 (b) shows that the number of bus transactions decreases significantly when more than 2, 4, and 8 coloring is used and the total performance improves.

In order for the coloring scheme to be effective, the number of colors must be larger than the threshold selected by equation (3). Otherwise, the effect of the coloring scheme is hindered by the side-effect of increasing bus transactions with higher cache conflicts.

5 Scalability enhancement with coloring

In this section, we describe the effect of coloring from the viewpoint of scalability. We find that our coloring method is effective in the case of many CPUs.

5.1 Experimental Environment and Benchmarks

The experimental environment is shown in Table 3 and we used the 2.4.4 distribution of the Linux kernel. To evaluate the effect of our coloring method, we ran the following two benchmarks on an 8-way IA server machine: WebBench 3.0 and Chat micro benchmark [7, 8, 9]. A lot of processes are created when these benchmarks are running. In particular, since the run queue length of Chat is longer than that of WebBench, lock contention is expected to be higher. We can expect that the coloring effect in Chat micro benchmark shows larger scalability than in WebBench.

In the case of WebBench, 256 maximum client-threads running on 28 standard PCs, simultaneously send requests to the server machine. During our experimentation, 256 httpd processes are always runnable on the server machine.

	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
32 coloring	8.2%	13.3%	15.8%	17.6%	18.0%	20.6%	22.6%	23.3%
Multi-Queue	–3.0%	13.6%	18.2%	21.7%	23.8%	26.9%	28.7%	30.6%

Table 4: Speeding up to vanilla scheduler (WebBench)

CPU	Pentium III Xeon 550MHz × 8
Memory	1GB
L2 cache	1024KB (4-way set associative)
NIC	Netgear GA622T (1GbE) × 4

Table 3: The specification of the server machine

The Chat micro benchmark is a stand-alone type benchmark, and no client machine is used. This benchmark simulates chat rooms with multiple users exchanging messages using TCP sockets. Each chat room consists of 20 users, and each user broadcasts a number of 100 byte messages in the room. To handle message exchange, one user program creates four threads. Thus 80 threads are created per room. The characteristic parameters of the Chat micro benchmark are the number of rooms and the number of messages per user. We choose 30 rooms and 300 messages per user as parameters, so that 2400 processes (threads) are generated on a system during experimentation.

In this research, we also evaluated the multi-queue [8] (we called MQ) scheduler proposed by IBM, which is an alternative to vanilla scheduler. MQ scheduler separates the run queue and the run queue lock for each CPU in the system.

5.2 Result

5.2.1 WebBench

Figure 9 shows WebBench results for the following three kernel: standard kernel (vanilla), 32 coloring kernel (c32), and multi-queue scheduler kernel (MQ). Table 4 shows performance gains of c32 and MQ compared with vanilla.

We can see in this graph that both of c32 and MQ show better performance than vanilla. In Table 4, c32 and MQ achieves maximum of 23.3% and 30.6% improvement on an 8-way, respectively. In fact, MQ shows more performance improvement than c32. However, MQ requires more extensive changes to be implemented than c32. Although requiring less modification, c32 nearly achieves

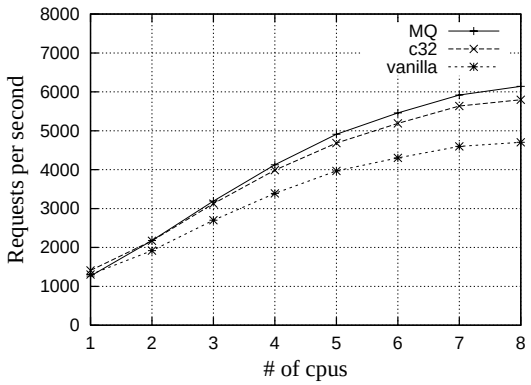


Figure 9: WebBench Performance (# of httpd processes: 256)

the performance improvement of MQ. Furthermore, c32 achieves speedup over all of the CPU set. In contrast, MQ's performance drops on single CPU system.

In Table 4, as the number of CPUs increases, c32 gains larger speeding up. When the number of CPUs is one, the performance gains is 8.2%. When the number of CPUs becomes 8, the performance gains 23.3%. The major reason is the improved lock statistics. The reduction of cache misses in scheduler makes the traversal time shorter and dramatically decreases the holding time for the run queue lock. Thus, our coloring method could perform more effectively on a SMP system rather than a uniprocessor system.

To confirm this, we also measured following two characteristics.

L2 cache miss ratio: measuring the number of L2 cache miss ratio during the execution of `list_for_each` loop. The miss ratio is calculated by the equation (1) as described in Section 4.2. We used performance monitoring counters built into the Pentium processor.

Lock hold time and lock contention: measuring following two statistics: (1) the time that the run queue lock is held, and (2) the fraction of lock requests that found the run queue lock was busy when it was requested. These information are measured by using

	1CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	92.4%	98.0%	98.0%	92.4%	96.2%	94.6%	92.3%	93.9%
32 coloring	6.0%	7.5%	6.8%	7.2%	6.6%	7.4%	11.1%	13.7%

Table 5: L2 cache miss ratio during run queue traversal (WebBench)

		2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	Contention	9.2%	15.4%	19.7%	22.9%	27.2%	35.9%	45.6%
	Hold Mean [us]	42	40	40	41	43	43	36
	Hold Max [us]	133	137	143	157	153	153	156
32 coloring	Contention	2.3%	4.1%	6.0%	7.2%	9.3%	14.0%	23.4%
	Hold Mean [us]	12	12	13	13	13	11	13
	Hold Max [us]	112	129	113	135	132	78	80

Table 6: lock statistics for run queue lock (WebBench)

Lockmeter tool [10].

The results are shown in Table 5 and 6, respectively. In the case of vanilla kernel, the `list_for_each()` loop involves potentially large cache misses as seen in Table 5, resulting more than 90% cache miss ratio. In contrast, 32 coloring shows substantial reduction to about 10% on any number of CPUs system. Moreover, in Table 6, both of the mean of lock hold time and the lock contention are largely reduced on any number of CPUs system by using 32 coloring. The lock contention is reduced from 45.6% to 23.4% on an 8-way system. This leads to better scalability than vanilla kernel.

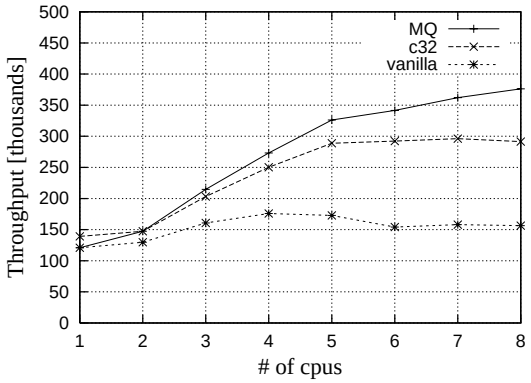


Figure 10: Chat Performance (30 rooms, 300 messages)

5.2.2 Chat

Figure 10 shows the throughput performance on the standard kernel (vanilla), 32 coloring kernel (c32), and multi-queue scheduler kernel (MQ).

As expected, c32 shows significantly better throughput than vanilla. Vanilla kernel slightly scales up to 4 CPUs, but its throughput performance starts dropping from 5 CPUs upward. On the contrary, c32 scales up to 6 CPUs which provides 89.6% throughput improvement compared to vanilla. MQ gains the best throughput among these three kernels.

As similar in WebBench, we collected information for L2 cache miss ratio and lock statistics. The results are shown in Table 7 and 8, respectively. We can see that there is a substantial reduction in L2 cache miss ratio on any number of CPUs, leading to speedup of the run queue traversal. This result shows that the lock hold time is reduced from 40us to 14us on the 8 CPUs system by the coloring scheme. The lock contention is also decreased from 85.8% on vanilla to 69.7% on c32. The run queue lock contention of Chat micro benchmark is higher than that of WebBench.

6 Related work

Several schemes for speeding up Linux scheduler have been proposed.

Kravetz et al. proposed the multi-queue scheduler to enhance scalability of the Linux 2.4.x on large scale SMP machines [8]. The multi-queue scheduler separates the global run queue into a number of queues and distributes them to each CPU. Each CPU maintains its own run queue. This scheduler aims to reduce the run queue lock

	1CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	99.7%	99.8%	76.4%	98.8%	64.5%	85.3%	94.4%	84.8%
32 coloring	3.2%	4.6%	2.8%	2.4%	5.8%	3.6%	2.8%	3.3%

Table 7: L2 cache miss ratio during run queue traversal (Chat)

		2 CPUs	3 CPUs	4 CPUs	5 CPUs	6 CPUs	7 CPUs	8 CPUs
vanilla	Contention	33.0%	48.4%	51.9%	73.0%	75.0%	85.2%	85.8%
	Hold Mean [us]	113	51	32	50	34	54	40
	Hold Max [us]	352	196	125	186	147	270	155
32 coloring	Contention	23.6%	38.7%	48.3%	56.4%	60.6%	61.9%	69.7%
	Hold Mean [us]	26	25	18	19	17	13	14
	Hold Max [us]	235	215	192	200	168	133	166

Table 8: lock statistics for run queue lock (Chat)

contention among processors. On the other hand, our coloring scheme aims to reduce cache conflicts during the run queue traversal. These two methods have different purposes and are orthogonal with each other. Furthermore, our coloring scheme can also be merged with the multi-queue scheduler and accelerate its performance.

Molloy et al. proposed the ELSC scheduler [11]. This scheduler focuses on pre-calculating base priorities and sorting the run queue for efficient task selection. The base priority is calculated by variables, such as `counter` and `priority`, which do not change while the runnable task is not running on a processor. This scheduler maintains a table which is sorted in the base priority. Each entry of the table contains a double-linked list which involves processes with same priority. The Priority Level Scheduler (PLS) [8] is proposed as a similar scheme. Both ELSC and PLS can make scheduling decisions faster, but cannot eliminate run queue lock contention, therefore do not show further scalability as the number of CPUs increases.

In [12], Sears has pointed out severe cache misses, which occur in the current scheduler, and provided the solution by using a prefetch technique. This improvement has already been merged to the latest kernel (after 2.4.10 version). However, in order for the prefetch technique to perform efficiently, the memory access latency and `goodness()` execution must overlap almost perfectly. The potential problem of this method is that these values depends on the memory system configuration.

Development of the Linux kernel is performed very rapidly on the Linux kernel mailing list. We have already contributed our scheme as a patch. Concurrently, Spraul has also contributed a patch with another implementation for coloring task structures. In his implementation, the kernel allocates task structures through the Slab Allocator [4, 13]. His implementation does not fix the number of colors nor does not depend on the cache configuration. In this respect, it is much more general than ours. On the other hand, our approach needs only a small modification to `get_current()` function, and the coloring effect could be quickly verified. Using our implementation, we can control the number of colors. Therefore, we could analyze the effect of the coloring, reduction of bus traffic, L2 cache misses and lock contentions, as a function of the number of colors.

7 Conclusions

We showed in this paper that the current Linux kernel potentially has scalability problem due to severe cache line conflicts from the placement of task structures in physical memory. We observed memory bus transactions on real SMP server systems and confirmed that large number of cache misses occur in the scheduler under heavy workload.

To address this issue, we proposed and implemented the cache coloring for a task structure. The evaluation result of this implementation demonstrates that the cache miss ratio while traversing the run queue is significantly reduced and the scheduling speed is enhanced. In WebBench, the web transaction performance on the coloring kernel achieved maximum of 42.3% improvement on 4-way Pentium Pro 200MHz system and 23.3% improvement on 8-way Pentium III Xeon 550MHz system. In Chat benchmark, the message throughput on the coloring kernel showed a maximum of 89.6% improvement on 8-way Pentium III system.

Reduction of cache misses can lead to decreasing run queue traversal time. On a SMP system this results in decreasing the lock hold time and lock contention. This is the effect of coloring on a large scale SMP machine. We verified these effects on an 8-way system, and found that the coloring scheme achieves better scalability than the standard kernel.

On the other hand, there is potential disadvantage caused by coloring: useful data are replaced on colored lines. To avoid this problem, we provided a simple model to decide the appropriate number of colorings, and verified the model with the bus transactions data observed on a real system.

As the gap between processor and memory speed grows wider the cache conflict issue caused by the current scheduler becomes more serious. Our coloring scheme is an essential technique for ameliorating this issue. The coloring scheme patch is contributed to the open source community, and is freely available for use and modification. The current patch can be downloaded from *http://www.labs.fujitsu.com/en/techinfo/linux/*.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. We would like to thank Kazuhiro Matsumoto and Andreas Savva for their helpful comments. In addition, we would like to thank the people on the Linux kernel mailing list who provided us with valuable comments.

References

[1] "Linux Scalability Effort Project", http://sourceforge.net/projects/lse

[2] "Development of GATES (Memory Access Trace System for a PC Server)", Mitsuru SATO, Akira NARUSE, Kouichi KUMON, Proceedings of the 59th National Convention IPSJ, September 1999 (in Japanese).

[3] "Solaris Internals Core Kernel Architecture", Jim Mauro and Richard McDougall, SUN MICROSYSTEMS PRESS.

[4] "The Slab Allocator: An Object-Caching Kernel Memory Allocator", Jeff Bonwick, In USENIX Conference Proceedings, pp 87-98, 1994.

[5] "Understanding the Linux Kernel", Daniel P. Bovet, Marco Cesati, O'Reilly & Associates, pp 69-70, October 2000.

[6] "WebBench Homepage", http://etestinglabs.com/benchmarks/webbench/webbench.asp

[7] "Java Technology, Threads, and Scheduling in Linux", R. Bryant and B. Hartner, Java Technology Update, 4(1), Jan 2000.

[8] "Enhancing Linux Scheduler Scalability", Mike Kravetz, Hubertus Franke, Shailabh Nagar, Rajan Ravindran, 5th Annual Linux Showcase & Conference, November, 2001.

[9] "Linux Benchmark Suite Homepage", http://lbs.sourceforge.net/

[10] "Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel", Ray Bryant, John Hawkes, 4th Annual Atlanta Linux Showcase & Conference, October, 2000.

[11] "Scalable Linux Scheduling", S.Molloy and P. Honeyman, In CITI Technical Report 01-7, University of Michigan, May 2001.

[12] "The Elements of Cache Programming Style", Chris B. Sears, 4th Annual Atlanta Linux Showcase & Conference, October, 2000.

[13] "UNIX Internals: The New Frontiers", Uresh Vahalia, Prentice Hall.