Relax: Automatic Contention Detection and Resolution for Configuration related Performance Tuning

Zhimin Feng
National University of Defense Technology
Changsha, China
fengzhimin16@nudt.edu.cn

Shanshan Li
National University of Defense Technology
Changsha, China
shanshanli@nudt.edu.cn

Xiangke Liao
National University of Defense Technology
Changsha, China
xkliao@nudt.edu.cn

Xiaodong Liu
National University of Defense Technology
Changsha, China
liuxiaodong@nudt.edu.cn

Yunfeng Li
National University of Defense Technology
Changsha, China
liyunfeng12@nudt.edu.cn

Shulin Zhou
National University of Defense Technology
Changsha, China
zhoushulin@nudt.edu.cn

***Abstract*—As the scale and complexity of software expands, the issue of software performance is attracting increasing attention. The causes of performance problems mainly fall into two categories: software bugs and the resource contention among multiple software programs. Software bugs are usually caused by inefficient or unnecessary computation in source code. However, the performance problems caused by resource contention among multiple software programs are usually ignored by most researchers. Unlike software bugs, resource contention is not a bug; as a result, it is difficult to identify the concrete reason for a performance problem given that they share the same symptoms, such as long response time or low system throughput. In this paper, we investigate the performance problems caused by resource contention from a configuration perspective. By studying the response time distribution of software as the workload changes, we find that there is an inflection point of response time with the change of workload. Based on our observations, we design and implement a tool, *Relax*, to automatically detect and resolve resource contention. *Relax* combines resource request delay at the inflection point and the system resource usage rate to identify the performance problems caused by resource contention. Moreover, inspired by the congestion control algorithm in computer networks, *Relax* uses the square-increase and multiplicative-decrease method to adjust the resource-related configurations so as to resolve the contention. Our experiments show that *Relax* can effectively detect and resolve resource contention, and shorten the total software response time by 15.8% $\sim$ 22.8%.**

***Keywords*-resource contention; performance problem; resource-related configurations; resource dependence analysis;**

I. Introduction

In recent years, given the increase in large-scale software designed to solve complex problems, performance issues stand out as elements that prevent applications from meeting their performance requirements. Some of these issues are caused by software bugs [1-6], which are themselves usually caused by inefficient or unnecessary computation in source code [7]. However, there is another situation that may also cause serious performance problems: namely, resource contention among multiple software programs. Moreover, as the development of cloud computing and big data technology has progressed, performance problems caused by resource contention increase rapidly when multiple programs request the same, limited resources. Resource contention may cause significant performance degradation and even program crashes. For example, Figure 1 shows a program crash caused by resource contention in StackOverflow question 41192896 [8]. The cause of the issue is that *MySQL* cannot allocate memory for the buffer pool. Tuning *Apache* configurations, which are related to memory, is one of the solutions for this issue.

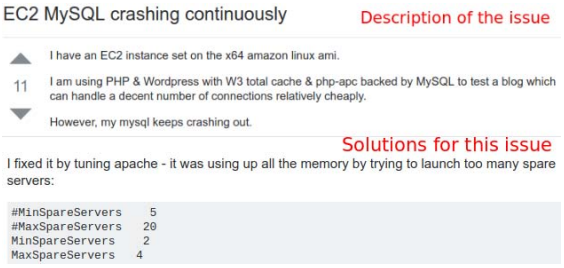


Figure 1. The program crash caused by resource contention

Previous studies find that more than half of the performance problems are related to configurations [9][28]. Additional surveys show about a third of Hadoop's misconfiguration problems result in OutOfMemoryErrors [29]. And setting proper configurations is challenging because the workload and system interaction are often too complicated or change too quickly for users to maintain a proper setting

978-1-7281-1970-0/18/$31.00 ©2018 IEEE
DOI 10.1109/APSEC.2018.00038

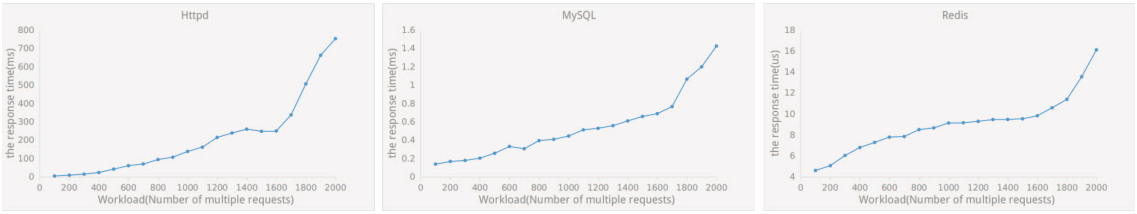


Figure 2. Response time distribution

[29]. In many cases, there is simply no satisfactory static setting [30]. In fact, the reason for resource contention is that the configuration of software cannot adapt the current running environment.

In this paper, we investigate the performance problems caused by resource contention from a configuration perspective. The process of solving these performance problems can be divided into two steps, detecting contention and resolving contention. First, we judge whether resource contention is occurring in the running environment. Second, when resource contention occurs, we reallocate the software's resource usage by iteratively adjusting the resource-related configurations until the contention is resolved.

The study of performance problems caused by resource contention faces many challenges. Firstly, it is difficult to identify the cause of performance problems because the problems caused by software bugs or resource contention exhibit the same symptoms (e.g. long response time or low system throughput). Secondly, since each piece of software takes up system resources, it is difficult to identify which program is the root cause that leads to resource contention in the complex running environment. Thirdly, it is still difficult to effectively resolve the contention, as we cannot determine the extent to which the software resources are adjusted so that the current contention can be resolved.

By studying the response time distribution of software as the workload changes, we find that there exists an inflection point of response time. Based on our observations, we first identify the resource request delay at the inflection point and use *delay-threshold* to denote it. The performance problems caused by resource contention are then identified by judging whether the resource request delay exceeds the *delay-threshold* and the usage rate of system resources exceeds the upper limit. Subsequently, the software that leads to resource contention is identified based on the changes in the resources required by the software before and after the contention. Furthermore, inspired by the square-increase and multiplicative-decrease (SIMD) [10] method in network congestion avoidance, we adjust the resource-related configuration options dynamically to resolve resource contention.

In this paper, we design and implement a tool, *Relax*, to detect and resolve the performance problems caused by resource contention from a configuration perspective. *Relax* can automatically detect resource contention and dynamically adjust resource-related configurations in order to resolve it. The main contributions of this paper are as follows.

- First, we study the response time of software as the workload changes and observe an inflection point of response time. Based on our observations, we identify the performance problems caused by resource contention according to the resource request delay at the inflection point and the usage rate of system resources. This approach can effectively identify the performance problems caused by resource contention.
- Second, we resolve resource contention from a software configuration perspective. Inspired by the congestion control algorithm in computer networks, we design a SIMD method to dynamically adjust the resource-related configuration options.
- Third, we evaluate *Relax* under different workload and configuration settings. Our experiments show that *Relax* can effectively detect and resolve resource contention, and also shorten the total response time of software by 15.8% $\sim$ 22.8%.

The rest of the paper is organized as follows. Section II summarizes the findings of the empirical study. Section III illustrates the architecture of *Relax*. Section IV explain the implementation details of *Relax*. Section V evaluates the effectiveness and overhead of *Relax*. Section VI discusses the limitations of *Relax*. Section VII presents the related work. Lastly, we conclude our work in section VIII.

II. Response Time Analysis in Resource Limitation

We know that multiple software programs may compete for limited resources as the workload changes, resulting in significant performance degradation. Different software workloads have different effects on software response time. In order to study the response time distribution of software as the workload changes, we designed and carried out the following experiments.

A. Experimental Setting

Our experiments are based on three projects: namely *Httpd* [11], *MySQL* [12] and *Redis* [13]. All experiments are performed on a machine with two Intel i5-4590 3.3Ghz processors, 2GB memory, and one 20GB disk. The running

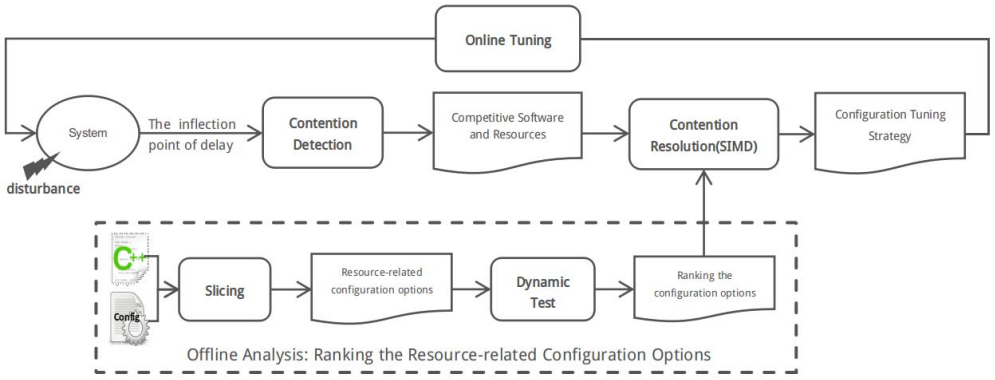


Figure 3. Automatic Contention Detection and Resolution

operating system is Ubuntu 16.04(X86_64) with Linux kernel 3.16.40.

The steps of the specific experiments are as follows. First, we use the different benchmark tools to test the three software programs (*Httpd*, *MySQL* and *Redis*). For *Httpd*, we use the popular *ab* [11] benchmark which has 10 requests for each client; for *MySQL*, we use the *mysqlslap* [12] benchmark tool and set 10 queries for each client; for *Redis*, we use a common benchmark (*redis-benchmark*) [13] set 10000 *SET* requests for each client. We simulate the different numbers of users accessing the three programs concurrently. We then measure the average response time per request for each software program under the different workload settings.

B. Observation

Figure 2 illustrates the response time of the three software programs under the different workloads. For *Httpd*, when the number of concurrent requests is less than 1700, the response time increases almost linearly as the workload increases; however, when the number of concurrent requests exceeds 1700, the response time grows radically due to the network bandwidth contention. *MySQL* and *Redis* exhibit the same feature as *Httpd*, in that the reason behind the significant increase in response time is CPU and memory contention.

Based on the above experiments, we find that an inflection point exists where the software response time changes from linear growth to rapid growth as the workload changes. This inflection point is an important condition for detecting resource contention.

III. *Relax* Overview

In order to resolve the performance problems caused by resource contention, we design and implement a tool, *Relax*, to automatically detect and resolve resource contention.

As shown in Figure 3, *Relax* contains two principal modules, namely the resource contention detection and resource contention resolution module. First, *Relax* determines whether resource contention exists in the running environment. Then, if resource contention is found to occur, *Relax* resolves the issue by reallocating the software resource usage.

A. Resource contention detection

Based on our observation in Section II, *Relax* detects resource contention according to the resource request delay at the flexion point and the usage rate of system resources in the operating system kernel. Then, *Relax* infers the competitive software and the competitive resource types based on both the software resource usage and the resource usage of the whole system. In Section IV-A, we illustrate the design and implementation details of this module.

B. Resource contention resolution

There is simply no satisfactory static configuration setting in many cases [30]. In fact, setting improper configurations is the root cause of resource contention. Thus we resolve resource contention from the perspective of software configurations. The specific process is divided into the following steps: first, *Relax* gets the configuration options that are related to the resources using a slicing method; second, *Relax* iteratively adjusts the configuration options that have a high impact on resources to resolve the current contention. More details are provided in Section IV-B.

IV. *Relax* Design and Implementation

In this section, we present the detailed design and implementation of *Relax*. Our goal is to resolve the performance problems caused by resource contention; accordingly, we first detect resource contention, then resolve it.

A. Resource contention detection

We divide the system resources into two major categories: (1) shared resources, (2) non-shared resources. Table I presents the classification of resources. Here, the term shared

Table I
RESOURCE CLASSIFICATION

Classification	Resource
Non-shared	Network port, Device serial port,etc
Shared	CPU, MEM, IO, NET,etc

resources refers to resources that allow use by multiple programs simultaneously, such as CPU resources, memory resources, etc. Non-shared resources are resources that can only be used by one program at a time, such as network ports, etc.

Non-shared resources It is easier to detect non-shared resource contention in the running environment. Once the non-shared resources is occupied, other programs cannot occupy it again. Therefore, we only need to insert a small amount of code into the system function tasked with requesting these resources that detects whether the request is successful. If the request fails, we can determine which program occupies the resource.

Shared resources We know that software cannot get enough resources in a timely fashion when resource contention occurs, resulting in resource request delays. Therefore, we use delays in resource requests as an important condition for detecting shared resource contention.

There are three main reasons behind software delay: (1) ready queue waiting delay (2) IO request delay (3) the number of page replacement. We call the time between process states from the ready state to the running state the *ready queue waiting delay*. We know that the operating system divides the process states into Created, Running, Ready, Blocked, and Terminated. When the process is in the ready state, and if the current CPU resource is idle, then the process is scheduled to execute; if the current CPU resource is short, the process will remain in the ready state until the scheduler selects it to be scheduled for execution. Thus, the ready queue waiting delay is the time between process states from the ready state to running state. The IO request delay is the time between the process of initiating an IO request to the completion of the request. The number of page replacement is the replacement number of between physical memory and virtual memory (SWAP). Table II shows the relationship between software delay factors and system resources.

We use *delay-threshold* to denote the resource request

Table II
RELATIONSHIP BETWEEN SYSTEM RESOURCES AND SOFTWARE DELAY FACTORS

System resources	Software delay factors
CPU	Ready queue waiting delay
MEM	The number of page replacement
IO	IO request delay
NET	IO request delay

Algorithm 1 Resource contention detection algorithm

Result: γ, η
1: $\gamma \leftarrow \emptyset$
2: $\eta \leftarrow \emptyset$
3: delay-threshold = getDelayThreshold()
4: **while** $True$ **do**
5: softwareDelay = getSoftwareDelay()
6: **if** $softwareDelay >$ delay-threshold **then**
7: candidateSet = getCandidateSet()
8: delayRes = getSoftwareDelayResource()
9: γ = candidateSet $\bigcap$ delayRes
10: **if** $\gamma \neq \emptyset$ **then**
11: value = getSoftwareChange()
12: η = getCompetitiveSoftware(value)
13: **end if**
14: **return** (γ, η)
15: **end if**
16: **end while**

delay at the inflection point, as discussed in Section II. We cannot use delay-threshold as the sole condition for contention detection because the inflection point of response time may occur when a single software has performance problems, for example, a performance bug can cause significant performance degradation [28]. Through the observation of resource contention among multiple software, we find that system resources are gradually exhausted by competitive software, which leads that the usage rate of system resources exceeds the upper limit. Thus, the main idea behind contention detection involves judging whether the resource request delay exceeds the *delay-threshold* and the system resource usage rate exceeds the upper limit.

Algorithm 1 outlines the specific process of shared resource contention detection. First, we get the *delay-threshold* (Line 3). Then, *Relax* determines whether resource contention is occurring (Line 6). Next, *Relax* identifies the competitive resource types and the software that leads to resource contention (Line 7-13). More specifically, *Relax* first gets the competitive resource set by judging whether the usage rate of system resources exceeds the preset threshold (Line 7). Second, *Relax* gets the set of resources that may lead to software delays (Line 8). Third, *Relax* gets the intersection of candidateSet and delayRes (Line 9); here, we use γ to denote the set of the competitive resource types. For example, if the usage rate of system CPU and MEM exceeds the preset threshold and there is frequent page replacement in the running environment, *Relax* can infer that there is contention for MEM resources. Finally, *Relax* only needs to calculate the differences between the competitive resource usage of each software program before and after the contention. With reference to these differences, *Relax* identifies the program that exceeds the preset maximum differences as the competitive software program (Line 11-

```
1  //ha_init_key_cache(sql/handler.cc)
2  int ha_init_key_cache(const char *name, KEY_CACHE *
       key_cache)
3  {
4    ......
5    size_t tmp_buff_size= (size_t) key_cache->
       param_buff_size;
6    ......
7    DBUG_RETURN(!init_key_cache(key_cache,
       tmp_block_size,tmp_buff_size,division_limit,
       age_threshold));
8  }
9  //init_key_cache(mysys/mf_keycache.c)
10 int init_key_cache(KEY_CACHE *keycache, uint
       key_cache_block_size, size_t use_mem, uint
       division_limit, uint age_threshold)
11 {
12   ......
13   blocks= (ulong) (use_mem / (sizeof(BLOCK_LINK) + 2
       * sizeof(HASH_LINK) + sizeof(HASH_LINK*) *
       5/4 + key_cache_block_size));
14   ......
15   my_large_malloc((size_t) blocks * keycache->
       key_cache_block_size, MYF(0)))
16   ......
17 }
18 //my_large_malloc(mysys/my_largepage.c)
19 uchar* my_large_malloc(size_t size, myf my_flags)
20 {
21   ......
22   ptr = my_large_malloc_int(size, my_flags)
23   ......
24 }
25 //my_large_malloc_int(mysys/my_largepage.c)
26 uchar* my_large_malloc_int(size_t size, myf my_flags
       )
27 {
28   ......
29   shmid = shmget(IPC_PRIVATE, size, SHM_HUGETLB |
       SHM_R | SHM_W);
30   ......
31 }
```

Figure 4. The configuration options indirectly affect the usage of resources

12). We use η to denote it.

B. Resource contention resolution

Recall that more than half of the performance problems are caused by configurations, and these configurations commonly affect memory, disk usage, etc [9][28]. Accordingly, we resolve resource contention from a software configuration perspective. We use different methods to resolve the contention caused by non-shared and shared resources. It is easier to resolve non-shared resource contention in the running environment. In this case, *Relax* only informs the administrator of the conflict information, after which the administrator can adjust the resources to resolve the conflict. For shared resource contention, *Relax* adjusts the resource-related configuration options dynamically. The main idea behind this resolution is that *Relax* adjusts the triggered configuration options that have a greater impact on resources.

1) ***Configurations Filter****:* Given that we resolve resource contention by adjusting the resource-related configuration options, we should therefore know which configuration options are related to resources. Specifically, we first apply

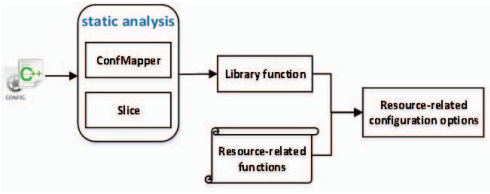


Figure 5. Configurations Filter

program slicing technique to get the library functions affected by configuration options. Then we determine whether the library functions affected by configuration options are related to the usage of resources. Through the above method, we can determine whether configuration option is related to resources. For example, Figure 4 shows the analysis process of the *key_buffer_size* configuration option in *MySQL*. We find that the *key_buffer_size* variable finally affects the first parameter of the *shmget* function. We know that the *shmget* function is a library function used to request shared memory, and the first parameter is used to control the size of the requested memory; therefore, we can speculate that the *key_buffer_size* configuration option is strongly related to memory resources.

Figure 5 outlines the resource-related configuration options filtering process. The input is a software source code and a configuration file, while the output is a set of resource-related configuration options. The implementation is shown in more detail below.

As existing *C/C++* program slicing tools, such as *srcSlice* [15], cannot slice the whole project, while our goal is to slice the configuration variables in the whole project, we implemented a tool for slicing *C/C++* programs based on the *srcml* [14].

We know that a precompiled macro is ordinarily used to control different implementations of the same function under different configuration settings. For instance, Figure 6 shows that the *hp_hashnr* function in *MySQL* is implemented differently under different configuration settings, if the *MySQL* source code is not preprocessed, some ambiguities may arise

```
1  #ifndef NEW_HASH_FUNCTION
2  /* Calc hashvalue for a key */
3  ulong hp_hashnr(register HP_KEYDEF *keydef, register
       const uchar *key)
4  {
5    ......
6  }
7  #else
8  ulong hp_hashnr(register HP_KEYDEF *keydef, register
       const uchar *key)
9  {
10   ......
11 }
12 #endif
```

Figure 6. Different implementations of the same function

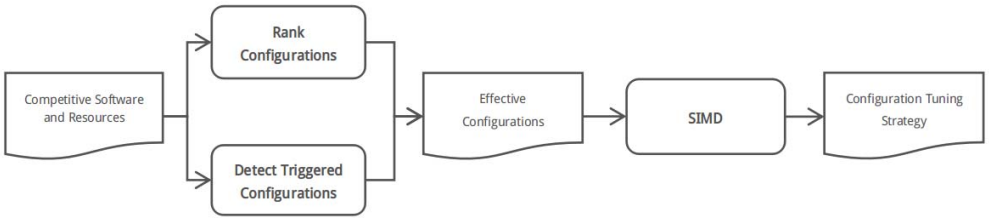


Figure 7. Resource Contention Resolution

when we parse the *hp_hashnr* function, which increases the program analysis error rate. In order to improve the accuracy of program analysis, we first precompile the program source code to expand the macro and eliminate ambiguous function declarations. We then use the *srcml* tool to convert the precompiled code to XML format. We obtain the call graph of the whole project by parsing the XML file, then use the *ConfMapper* [18] to obtain the corresponding variables of the configuration options in the source code. We next slice the configuration options in order to obtain the affected library functions based on both data flow analysis and control flow analysis. Finally, we determine whether the configuration option is related to resources by analyzing whether the affected library function belongs to the resource-related library function.

2) ***Configurations Ranking****:* To effectively resolve the contention, *Relax* first adjusts the configuration options that have a greater impact on resources. Although we can get the resource-related configuration options via static analysis, *Relax* cannot know which configurations have a greater impact on resources. Accordingly, we implement an automatic tool to analyze the relationship between the resource-related configuration options and resources usage based on the existing work *SPL Conqueror* [16][17]. Based on this relationship, we can get the relative influence coefficient of each configuration options. Detailed implementation is shown below.

First, we get the resource-related configuration options by program slicing technique, and the resource-related configuration options are sampled to generate multiple sets of configurations. Then each set of configuration options is tested offline, we can get software's resource usage under the different configuration settings. Finally, based on these test datas, we use stepwise linear regression to learn the relationship between the resource-related configuration options and the resources usage of software.

Table III
TOP 10 MEMORY-RELATED CONFIGURATIONS OF MYSQL

Memory-related configurations	Ranks
sort_buffer_size	1
join_buffer_size	2
key_buffer_size	3
read_buffer_size	4
bulk_insert_buffer_size	5

As shown in Table III, we get the top 5 memory-related configuration options of *MySQL*. From this table, we can see that *sort_buffer_size* has the greatest impact on memory resource. Accordingly, *Relax* first adjust the value of *sort_buffer_size* when memory resources are constrained.

3) ***Triggered Configurations Detection****:* Because not all resource-related configuration options are triggered when the software is running, we use program instrumentation for the resource-related configuration options in order to obtain the triggered configuration options. We insert the marker code into the path affected by resource-related configuration options. This marker code is used to write a different string to the specified file. *Relax* then identifies the triggered configuration options by establishing whether the string exists in the specified file. In this way, *Relax* can accurately determine whether a configuration option is triggered when the software is running. Therefore, when resource contention occurs, *Relax* only needs to adjust the triggered configuration options related to the competitive resources.

4) ***Configuration tuning based on SIMD****:* When resource contention occurs, *Relax* first adjusts the configuration options with a more significant impact on resources. Inspired by the congestion control algorithm in computer networks, *Relax* uses the SIMD method to dynamically adjust the resource-related configuration options. This helps to ensure that resource contention can be resolved effectively. The control rules of SIMD as following:

$$Increase: \quad \omega_{t+R} \leftarrow \omega_t + \alpha\sqrt{\omega_t - \omega_0}, \quad \alpha > 0. \tag{1}$$

$$Decrease: \quad \omega_{t+\delta} \leftarrow \omega_t - \beta\omega_t, \quad 0 < \beta < 1. \tag{2}$$

where ω_t is the value of configuration option at time *t*, *R* is the round-trip time, and δ is the time to detect resource contention.

Figure 7 outlines the specific process of shared resources contention resolution. *Relax* first analyzes the configuration options of the competitive software using program instrumentation and dynamic testing methods, thus identifying the effective configuration options that are triggered and have a greater impact on resources. *Relax* then uses SIMD method to get the configuration tuning strategy.

Table IV
OVERVIEW OF SAMPLE PROGRAMS USED IN THE EVALUATION

Project	Domain	Lang.	LOC	Configuration options
MySQL	Database	C/C++	2239K	461
Redis	Database	C	112K	94
Httpd	Web Server	C	230K	550+

Table V
BENCHMARKS

Software	Benchmark Tools	Release
MySQL	SysBench	0.4.12
Redis	redis-benchmark	4.0.1
Httpd	ab	2.3

V. EVALUATION

In this section, we conduct the experiments to evaluate the effectiveness and overhead of *Relax*. We construct a resource contention scenario centered around building a personal website. To do so, we select three existing real-world programs (see Table IV). Here we assume that *Httpd* has the highest priority, followed by *Redis* and then *MySQL*. When resource contention occurs, we prefer to adjust the resource usage of software that has the lower priority.

A. Experimental Setting

In order to simulate the actual resource contention scenario, we use several benchmarks to conduct all evaluations; these are listed in Table V.

Machines Our experiments are performed on a machine with two Intel i5-4590 3.3Ghz processors, 2GB physical memory, 2GB virtual memory, and one 40GB disk. The running operating system is Ubuntu 16.04(X86_64) with Linux kernel 3.16.40.

B. *Can* Relax *correctly detect and resolve resource contention?*

We next evaluate whether *Relax* can correctly detect and resolve resource contention under the different workload and configuration settings. We first select high, medium and low workloads to test *Relax* under the same configuration settings. Then, we select three different configurations to test *Relax* under the same workload settings. The design is explained in more detail below.

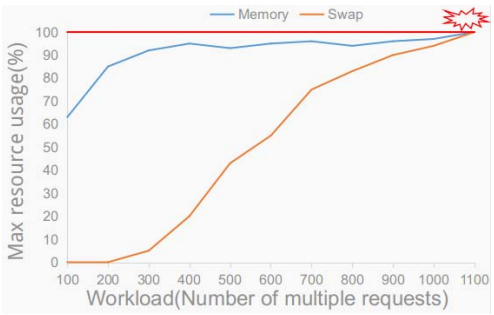


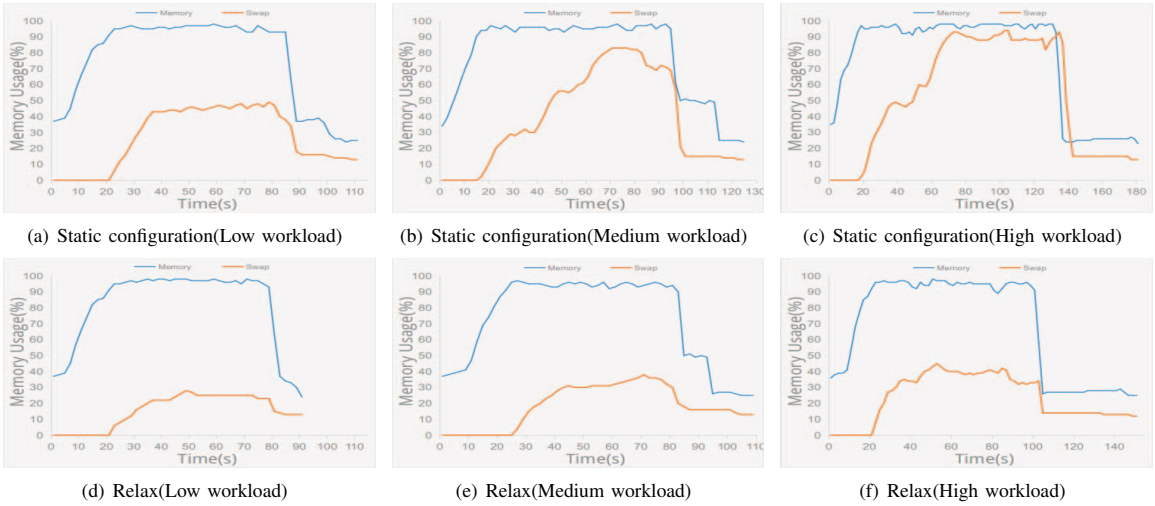
Figure 8. The selection of workload

1) Different workload settings: We construct a memory contention scenario under the high, medium and low workload settings. As shown in Figure 8, we determine the maximum usage of the system memory with the change of workload of the software (*MySQL*, *Redis* and *Httpd*). We find that the total workload of these software programs is equal to 1100, resulting in an operating system crash. The system also experiences a significant delay when the workload exceeds 500. Accordingly, we choose three different workloads between 500 and 1000. More specifically, for *MySQL*,we simulate 500, 800 and 1000 users respectively to add, delete, update and select different tables concurrently (these tables contain 100000 pieces of data); for *Redis*, we use the redis-benchmark benchmark, which has 100000 requests, and set the number of parallel connections at 500, 800 and 1000 respectively; for *Httpd*, we use *ab* benchmark1), which has 1000 requests for each client, and simulate 500, 800 and 1000 users respectively to concurrently request a Web page.

As shown in Figure 9, we get the mean execution time for each program under different workload settings. Figure (a), (b) and (c) show the trend of total memory usage when the three software programs (*MySQL*, *Redis* and *Httpd*) are running under static configurations. Figures (d), (e) and (f) show the trend of total memory usage when the software is running under the *Relax*-adjusted resource-related configuration options. The blue refers to the physical memory usage trend, while the orange refers to the virtual memory usage trend. From these figures, we can see an obvious performance improvement after the resource-related configuration options are adjusted by *Relax*. Table VI presents the mean execution time of the three software programs before and after contention resolution under the different workload settings. We find that the total response time of the three programs is shortened by 16.3%, 15.8% and 22.8% under the low, medium and high workload settings respectively.

2) Different configuration settings: We construct a memory contention scenario under three different configuration settings. First, we keep the workload settings for these programs static while varying the configuration settings. For *MySQL*, we simulate 500 users to add, delete, update and select the different tables concurrently, where these tables have 100 thousand pieces of data; for *Redis*, we use the *redis-benchmark* benchmark, which has 100000 requests, and set the number of parallel connections at 500; for *Httpd*, we use *ab* benchmark, which has 1000 requests for each client, and simulate 500 users concurrently requesting a Web page. Second, we set three configurations for *MySQL*: *key_buffer_size*, *sort_buffer_size*, *join_buffer_size* are set to 32M, 128M and 512M.

As shown in Figure 10, we get the mean execution time for the software under the different configuration settings.



(a) Static configuration(Low workload) (b) Static configuration(Medium workload) (c) Static configuration(High workload)

(d) Relax(Low workload) (e) Relax(Medium workload) (f) Relax(High workload)

Figure 9. Relax vs. static configuration under the different workload settings

Table VI
THE MEAN EXECUTION TIME OF SOFTWARE UNDER THE DIFFERENT WORKLOAD SETTINGS

Software	Mean execution time(s)					
	Low workload		Medium workload		High workload	
	Static optional	*Relax*	Static optional	*Relax*	Static optional	*Relax*
MySQL	84.112	80.385	96.172	84.238	135.329	101.606
Redis	106.363	81.996	115.965	91.126	141.814	103.133
Httpd	109.761	88.934	124.827	108.242	178.482	147.191
Total	300.236	251.288	336.964	283.606	455.625	351.93

Figures (a), (b) and (c) show the trend of total memory usage with the software (*MySQL*, *Redis* running under the static configurations. Figures (d), (e) and (f) show the trend of total memory usage with the software running under *Relax*-adjusted resource-related configuration options. From these figures, we can see an obvious performance improvement after the resource-related configuration options are adjusted by *Relax*. Table VII shows the mean execution time of the three software programs before and after the contention resolution under the different configuration settings. We find that the total response time for the three software programs is shortened by 16.3% $\sim$ 19.9% under the different configuration settings.

C. The convergence analysis of Relax

We initialize the value of β to 1/2 in Equation (2). We use $Conf$ to denote the configuration options that need to be adjusted. We assume that the default value of $Conf$ is b, the minimum value of $Conf$ is a ($a \leq b$), and $Conf$ is positively related to the resource usage. When resource contention occurs, we need to adjust the value of $Conf$ less than or equal to a specific value (we use c to denote this value; i.e. $a \leq c \leq b$) to resolve current contention. Since we initialize the value of β to 1/2, the range of iteration adjustment is 1 to $log_2(b-a)$. When c is greater than or equal to $\frac{b+a}{2}$, we only need to adjust the configuration options once.

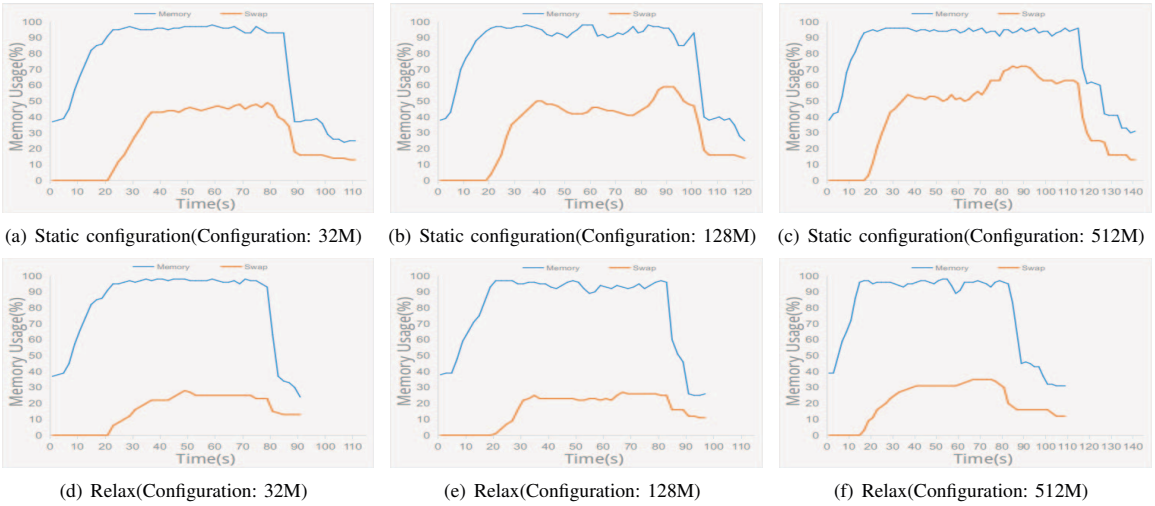
D. Overhead

In order to determine whether a configuration option is triggered when the software is running, we use program instrumentation for the resource-related configuration options; the instrumentation affects the software performance. We use the same benchmark as outlined in Section II to calculate the overhead caused by program instrumentation. Figure 11 illustrates the software performance before and after instrumentation under the different workload settings. The red refers to the source code after instrumentation, while the blue refers to the original source code. From this figure, we can see that the instrumentation overhead is low.

To illustrate the overhead of *Relax*, we monitor the resources usage of *Relax* in the running environment. We found that *Relax*'s resource usage is 2%(CPU), 3%(MEM), 0%(IO) and 0%(NETWORK), we can ignore the overhead of *Relax*.

VI. DISCUSSION

We have several assumptions in design and implement *Relax*. Firstly, we assume that the contention for a type of system resource only affects one type of factor that leads



(a) Static configuration(Configuration: 32M) (b) Static configuration(Configuration: 128M) (c) Static configuration(Configuration: 512M)

(d) Relax(Configuration: 32M) (e) Relax(Configuration: 128M) (f) Relax(Configuration: 512M)

Figure 10. Relax vs. static configuration under the different configuration settings

Table VII
THE MEAN EXECUTION TIME OF SOFTWARE UNDER THE DIFFERENT CONFIGURATION SETTINGS

Software	Mean execution time(s)					
	Configuration: 32M		Configuration: 128M		Configuration: 512M	
	Static optional	*Relax*	Static optional	*Relax*	Static optional	*Relax*
MySQL	84.112	80.385	101.53	84.49	83.88	85.23
Redis	106.363	81.996	117.85	89.92	133.584	103.465
Httpd	109.761	88.934	119.09	96.78	139.39	106.98
Total	300.236	251.288	338.47	271.19	356.854	295.675

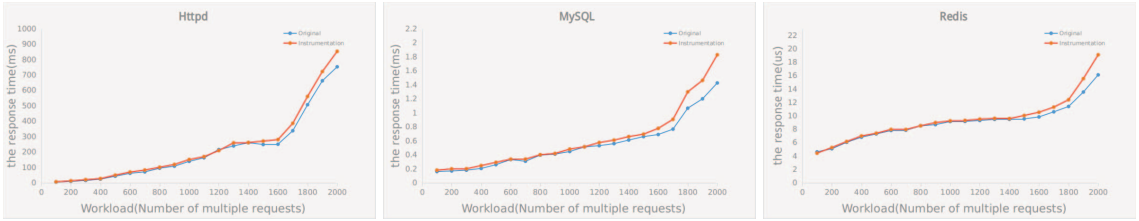


Figure 11. The overhead of instrumentation

to software delay. However, due to the complex relationships among the different types of system resources, the contention for a particular type of system resource may affect many types of factors that lead to software delay. Furthermore, we assume that the software program that causes resource contention supports online configuration adjustment. If the application can only apply configurations by restarting the server, *Relax* will be unable to resolve the contention effectively. Moreover, when the minimum resource usage sum of all software programs is greater than the total amount of resources that the system can provide, *Relax* will also be unable to resolve the contention.

VII. RELATED WORK

Misconfiguration Many empirical studies have looked at misconfigurations [6-8], but did not focus on the performance problems caused by resource contention. Much previous work has proposed using static program analysis [19][20] or statistical analysis [21][22] to identify and fix wrong or abnormal configurations. These techniques mainly target functionality-related misconfigurations, and do not work for the performance problems caused by resource contention, as the proper setting of performance problems caused by resource contention highly depends on the dynamic workload and environment, and can hardly be statistically decided based on common/default settings.

Performance-Bug Detection There are many dynamic and static analysis tools to detect different types of performance bugs, such as run-time bloat [23][24], low-utility data structures [25], cacheable data [26], inefficient loops [2], loops with unnecessary iterations [3], input-dependent loops [27].

These techniques mainly target code bugs, and do not work for the performance problems caused by resource contention, as the performance problems caused by resource contention is not a bug.

VIII. Conclusion

The resource contention among multiple software programs may cause performance problems in the complex running environment. Unlike the performance problems caused by software bugs, resource contention is not a bug. In this paper, we make an exploration on studying the performance problems caused by resource contention from a configuration perspective. And we design and implement a tool, *Relax*, to automatically detect and resolve resource contention. Our experiments show that *Relax* can effectively detect and resolve resource contention, and shorten the total software response time by 15.8% $\sim$ 22.8%.

Acknowledgment

The work described in this paper was substantially supported by National Key R&D Program of China (Project No.2017YFB1001802); National Natural Science Foundation of China (Project No.61872373, 61872375 and U1711261).

References

[1] J. Burnim, S. Juvekar, and K. Sen, Wise: Automated test generation for worst-case complexity, In ICSE, 2009.

[2] A. Nistor, L. Song, D. Marinov, and S. Lu, Toddler: Detecting performance problems via similar memory-access patterns, In ICSE, 2013.

[3] O. Olivo, I. Dillig, and C. Lin, Static detection of asymptotic performance bugs in collection traversals, In PLDI, 2015.

[4] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin, On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach, In CGO, 2015.

[5] S. Lu, S. Lu, Performance Diagnosis for Inefficient Loops, In ICSE, 2017.

[6] Z. Yin, X. Ma, J. Zheng, Y. Zhou, LN. Bairavasundaram, An empirical study on configuration errors in commercial and open source systems, In SOSP, 2011.

[7] J. Yang, C. Yan, P. Subramaniam, A. Cheung and S. Lu, How not to structure your database-backed web applications: a study of performance bugs in the wild, In ICSE, 2018.

[8] https://stackoverflow.com/questions/12384464/ec2-mysql-crashing-continuously/41192896#41192896.

[9] S. Wang, C. Li, H. Hoffmann, and S. Lu, Understanding and AutoAdjusting Performance-Sensitive Configurations, In ASPLOS, 2018.

[10] S. Jin, L. Guo, I. Matta, and A. Bestavros, TCP-friendly SIMD congestion control and its convergence behavior, In ICNP, 2001.

[11] Httpd, https://httpd.apache.org, 2018.

[12] MySQL, https://www.mysql.com, 2018.

[13] Redis, https://redis.io, 2018.

[14] srcml, https://www.srcml.org/, 2018.

[15] CD. Newman, T. Sage, ML. Collard, HW. Alomari, and JI. Maletic, srcSlice: A tool for efficient static forward slicing, In ICSE-C, 2016.

[16] N. Siegmund, S. Kolesnikov, C. Kstner, S. Apel,D. Batory, M. Rosenmller, and G. Saake, Predicting performance via automated feature-interaction detection, In ICSE, 2012.

[17] N. Siegmund, A. Grebhahn, S. Apel, and C. Kstner, Performance-influence models for highly configurable systems, In FSE, 2015.

[18] S. Zhou, X. Liu, S. Li, W. Dong, X. Liao, and Y. Xiong, Confmapper: automated variable finding for configuration items in source code, In QRS, 2016.

[19] A. Rabkin and R. Katz, Precomputing possible configuration error diagnoses, In ASE, 2011.

[20] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, Early detection of configuration errors to reduce failure damage, In OSDI, 2017.

[21] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar, Context-based online configuration-error detection, In USENIX ATC, 2011.

[22] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou, Encore: Exploiting system environment and correlation information for misconfiguration detection, In ASPLOS, 2014.

[23] G. Xu and A. Rountev, Detecting inefficiently-used containers to avoid bloat, In PLDI, 2010.

[24] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, Go with the flow: profiling copies to find runtime bloat, In PLDI, 2009.

[25] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, Finding low-utility data structures, In PLDI, 2010.

[26] K Nguyen and G Xu. Cachetor: Detecting cacheable data to remove bloat. In FSE, 2013.

[27] X. Xiao, S. Han, T. Xie, and D. Zhang, Context-sensitive delta inference for identifying workload-dependent performance bottlenecks, In ISSTA, 2013.

[28] X. Han and T. Yu, An empirical study on performance bugs for highly configurable software systems, In ESEM, 2016.

[29] A Rabkin and RH Katz. How hadoop clusters break. IEEE software, 2013.

[30] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu, Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs, In SOSP, 2015.