The Influence of Environmental Parameters on Concurrency Fault Exposures - An Exploratory Study

Sahitya Kakarla
Advanced Empirical Software Testing and Analysis Research Group (AVESTA)
Department of Computer Science
Texas Tech University
sahitya.kakarla@ttu.edu

Akbar Siami Namin
Advanced Empirical Software Testing and Analysis Research Group (AVESTA)
Department of Computer Science
Texas Tech University
akbar.namin@ttu.edu

ABSTRACT

Testing multi-threaded applications is a daunting problem mainly due to the non-deterministic runtime behavior of paralleled programs. Though the execution of multi-threaded applications primarily depends on context switches, it is not clear which environmental factors and to what degree control the threads scheduling. In this paper, we intend to identify environmental factors and their effects on the frequency of concurrency fault occurrences. The test practitioners and researchers will find the identified factors beneficial when testing paralleled applications. We conduct an exploratory study on the effect of multicore platforms on fault exposures. The result provides no support of the hypothesis that number of cores has some impact on fault exhibitions.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentations, Measurement

Keywords

Multi-Core, Solaris Containers, Concurrency, Threading.

1. INTRODUCTION

It is believed that testing and debugging multi-threaded applications is much harder than sequential programs primarily because of non-deterministic behavior imposed by scheduling mechanisms, which control the execution order of threads. The threads contentions or context switches occur when the underlying system prioritizes the execution of threads. Though explicit programming commands (e.g. `yield` and `sleep`) may be used to enforce desirable switching of contexts, some other implicit factors (e.g., environmental factors) may intervene threads scheduling and yield undesired computations, known as *interleaving* defects.

Detecting failures caused by undesired interleavings is an extremely complex task. Stoller [4] states that though model checking techniques can be utilized to explore all possible threads scheduling, the model checking approaches lack efficiency and suffer from the scalability problem. Stoller suggests adapting heuristic techniques to invoke likely context switches which lead to faults and interleaving failures.

Eytani et al. [3] point out that concurrency defects are hard to detect because of not only the large number of possible interleavings, but also the difficulty of reproducing interleaving faults. Eytani et al. introduce a randomization algorithm which targets events causing interleaving defects. Burckhardt et al. [2] state that interleaving bugs are usually caused by unanticipated interactions among a few instructions, which are shared by the threads. Burckhardt et al. present a probabilistic concurrency testing scheduler that runs a test program several times with given inputs and computes the probability of finding bugs for each test run.

Though heuristic and probabilistic techniques might be effective to inflate the frequency of interleaving faults exposures, non-algorithmic approaches are also of paramount interests. Some environmental factors such as operating systems, virtual machines, language compilers, hardware architectures, CPU and memory speeds, bus interrupt timings, and number of cores may also influence the runtime behavior of paralleled programs. It has already been shown that different versions of Java virtual machines perform differently with respect to performance and cost [1]. However, it is not clear whether different versions of Java virtual machines have any significant impact on concurrency fault exhibitions.

In this paper, we intend to identify environmental parameters which influence the runtime behavior of paralleled applications. The test practitioners and end-users of concurrent applications will benefit from the results when configuring test-beds and operational field platforms. Hence, it is desirable to configure testing (operational) environments thereby the likelihood of revealing faults increases (decreases). The main contributions of this paper are: *(1) identifying environmental factors that influence the runtime behavior of paralleled applications. (2) An exploratory study on a number of parallel applications to determine the influence of multicore systems on behavior of multi-threaded programs and the frequency of concurrency fault exhibitions.*

2. ENVIRONMENTAL PARAMETERS

Testing multi-threaded applications is an open grand problem primarily due to non-deterministic runtime behaviors. The debugging procedures can be improved substantially when the concurrency defects are reproducible. The primary purpose of this research is to identify a comprehensive list of environmental parameters which may influence fault exposures of paralleled faulty applications. The work will explore a research direction where the identified parameters can be set in controlled experiments in order to ample the exhibitions of concurrency defects and hence agile debugging. In an analogous way, the identified parameters can be set appropriately in configuring the operational platforms in order to lessen the faulty exhibitions of possible faults remaining. We classify the influential parameters into four categories: a) Hardware, b) Software, c) Defect types, and d) Concurrency levels[1].

Hardware Parameters. Hardware components such as hardware architecture, number of CPU cores, cache and buffer size, CPU, memory, and bus interrupt speeds may have some impacts on the behavior of multi-threaded applications. For instance, context switches happen in accordance with the CPU's clock tick. The shared memory area maintained within main memory and the corresponding memory speed may also cause context switches when the time allocated for executing a thread is reached its limit. The core management technology deployed in multicore systems such as *master-slave* or *dataflow-based* core management may also affect the executions[2]. Furthermore, recent technology advances in integrating software and hardware threads handling such as *CoolThreads*, *hyperthreadings*, and *virtualizations*[3] may also influence the runtime behavior of paralleled applications. The more recent technology trend in multicore systems enables allocating multiple cores to running applications and hence improving computations. Though the hardware factors may influence the fault exposures, it is of paramount interest to assess the degrees of influence of these hardware factors on runtime exhibitions of multi-threaded applications.

Software Parameters. The scheduling algorithms implemented by operating systems play vital roles in threads contentions. Some of the well-known scheduling algorithms such as *First-Come First-Served*, *Round Robin (RR)*, *Shortest-Job-First (SJF)*, and *Shortest Remaining Time (SRT)* enable CPUs to serve processes and consequently threads differently. For instance:

- Scheduling mechanism implemented by Solaris operating system is based on threads instead of processes. Threads are assigned priority numbers (between 0 and 59) which in turn each is assigned a time quantum. At the end of each quantum, the priority of thread is lowered until it reaches the lowest value (i.e. zero).
- Similar to Solaris operating system, scheduling on Windows XP is based on threads and not processes. However, the scheduling algorithm is *preemptive priority* thereby multiple queues are maintained for threads with 32 priority levels.
- Similar to Windows XP., Linux 2.5 uses preemptive priority algorithms. Unlike Windows XP, however, 140 possible priority levels are defined.

In addition to operating systems, language compilers and virtual machines also implement scheduling algorithms of their own. Therefore, it is expected that operating systems and virtual machines influence concurrency fault exposures significantly. However, the degree of impacts is an important challenging issue, which needs to be addressed.

[1]The classification is not mean to be a complete list.

[2]In the master-slave technology, a single core manages task assignments, whereas, in the dataflow-based core managements, the assignments are based on data dependencies.

[3]A technology to hide the physical characteristic of computing platforms.

Table 1: Programs used. NLOC: net lines of code. Race: Thread execution order and speed problem. Deadlock: Resource allocation contention problem. No Lock: Resource use contention resolution problem.

Program	NLOC	Bug Type
`bubble sort`	236	Race
`airline`	61	Race
`account`	119	Deadlock, Race
`deadlock`	95	Deadlock
`allocation vector`	163	No Lock

Concurrency Defect Types. Though this paper focuses primarily on interleaving defects, other concurrency defects are also of prominent interest. Several concurrency defects such as deadlock, livelock, starvation and race condition occur when the program counter enters critical sections. However, other kinds of concurrency defects such as orphaned threads and weak-reality may occur only under specific circumstances. Nevertheless, along with interleaving defects, these concurrency defects also need insights investigations.

Concurrency Levels. Number of threads required and created during runtime executions of applications may also play an influential role in the concurrency defect exposures. It is not clear whether amplifying the number of threads results in escalading context switching and therefore exhibiting faulty behavior. Intuitively, the number of threads impacts the complexity of threads management unit. However, the level of impacts and tradeoffs are among grand challenging questions which need to be addressed. The statistically-driven models may help in understanding the relationships between number of threads and faults exhibitions and assessment of their tradeoffs.

3. MULTICORE SYSTEMS: A CASE STUDY

We report the result of a case study conducted to investigate the influence of multicore platforms on concurrency fault exposures. The multicore processors are the most radical shifts in the hardware industry. In addition to hardware changes, multicore systems have also introduced new challenges in the software industry. To fully exploit the processing power of multicore processors, software systems need to be parallelized and tested accordingly. The paramount objective of this case study is to investigate whether there is any dependency between the numbers of cores, as an environmental parameter, and the concurrency defect exposures. We describe our experimental procedure and observations in followings.

Subjects Selection. Table 1 lists the subject programs used. The programs, prepared by IBM Haifa research laboratory, are parameterized so that the concurrency level (i.e., number of threads) can be specified prior to the executions. The pre-defined values are `little`, `average`, and `lot` concurrency levels. For instance, the numbers of threads generated by setting the concurrency levels to `little` and `lot` are 10 and 5000 for the `airline` program. Furthermore, each program implements a particular type of concurrency defects as described in Table 1.

Computer Systems. Two Sun machines with different threading technologies are used. The first machine, a

Table 2: The mean values of defect exposures of Solaris containers defined.

Program	M1000				T1000					
	Default	1-CPU	2-CPU	4-CPU	Default	1-CPU	2-CPU	4-CPU	8-CPU	16-CPU
a) `Little` Concurrency Level										
`bubble sort`	99.1	98.9	98.1	98.4	99.3	97	97.2	96.6	97.5	97
`airline`	98.7	98.9	98.8	98	99	99	99	99.2	99.4	99.1
`account`	0	0	0	0	0	0	0.1	0	0	0
`deadLock`	100	100	100	100	100	100	100	100	100	100
`all. vector`	5.5	3.5	4.2	4.1	3.1	1.6	2.4	2.4	2.1	1.9
b) `Lot` Concurrency Level										
`bubble sort`	3.4	3.9	3	4	7.7	6.4	5.4	5.9	5.8	6
`airline`	76.3	74.3	77	76.2	58.4	56.6	56.9	60.2	55.7	58.2
`account`	0.5	0.2	0.2	0.4	0.2	0.9	0.5	0.4	0.5	0.7
`deadLock`	100	100	100	100	100	100	100	100	100	100
`all. vector`	99	99.8	99.2	99	71.5	77	75.9	74.7	72.6	74.6

Sun Fire T1000 machine with UltraSPARC T1 processor, is a multi-core system supporting 32 concurrent hardware threads. According to Sun, this machine is suitable for tightly coupled multi-threaded applications where threads need little computations in order to accomplish their jobs and hence serve more concurrent threads [5]. The T1000 machine uses *CoolThreads* technology with eight cores each capable of handling four threads. The second machine is a Sun SPARC M3000 Enterprise system powered by a SPARC64 VII processor. The M3000 system supports eight concurrent hardware threads and is ideal for computationally expensive single-threaded workloads.

Solaris Containers. Solaris containers, introduced as part of Solaris 10 UNIX operating system, are similar to Linux *affinities* and are used to control hardware resources and in particular the assignments of CPU cores to applications. Solaris containers are server virtualization techniques enabling allocation of resources to applications. Applications are clustered based on the amount of resources they need. The resource management unit in the Solaris operating system restricts access to resources and isolates workloads and hence leads to a robust security mechanism. The processor pools and sets are created using the `create` UNIX command. We controlled the number of cores allocated for a Solaris *project* using `pset.max` and `pset.min` as parameters of `create pset` command. Solaris containers implement a built-in auto-tuning mechanism to control the minimum number of cores allocated for a running application. The minimum number of cores allocated changes over runtime based upon needs. To reduce possible construct threat, we set the minimum number of cores required to be equal to the maximum number of cores allocated, i.e. `pset.max=pset.min`. Thus, the validity of the desired number of cores allocated to a project is ensured.

Setup. Five *projects* (i.e., containers) are created on the T1000 computer system: `1-CPU`, `2-CPU`, `4-CPU`, `8-CPU`, and `16-CPU`[4]. A `default` container is already pre-defined by the Solaris operating system where the operating system automatically and dynamically (de)allocates cores to each processor set during runtime. Consequently, three containers are created on M3000 computer: `1-CPU`, `2-CPU`, `4-CPU`. The number of containers created is based on the availability of cores on each machine. The UNIX commands `poolcfg` and `projadd` are used to configure the processor sets, pools, and projects created. Furthermore, the UNIX command `mpstate` is used to monitor the utilization of cores and validity of assignments. We consider only `little` and `lot` concurrency levels. We run each program 100 times for each pair of concurrency level and container defined on each machine.

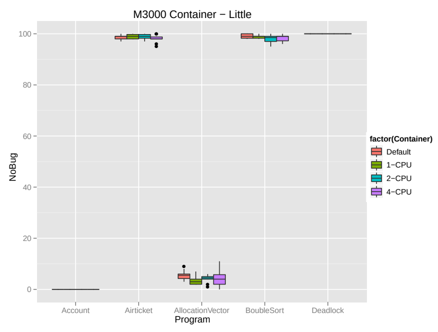
Data Analysis. Table 2 shows the mean values of the number of concurrency fault exposed for each program, concurrency level, and container. The mean values of fault exposures are similar across different containers including the default container. The mean values indicate that the frequencies of concurrency fault exhibited remain nearly unchanged across containers defined on each machine, even though the mean values change for different computer systems.

In addition to the mean values reported, the variations of data are also measured to determine the trend of data. The box plots can be used to represent data variations in terms of median, upper (75%), and lower (25%) quartiles. The box plots shown in Figures 1(a)-1(d) depict the variations of data for each program, concurrency level, and container. The x-axis holds the program's name and the y-axis represents the number of times the concurrency bugs exposed for 100 runs, for each container defined. As figures illustrate the numbers of bugs exposed for each subject program remain similar across different containers. Though the variations across different containers are similar for both `little` and `lot` concurrency levels, the variance of number of bugs exposed for the `lot` concurrency level is wider than those for the `little` concurrency level. Furthermore, the variations of fault expositions on T1000 are wider than those obtained for M3000. However, the variations of data obtained for each machine and program are quite similar across the containers defined.

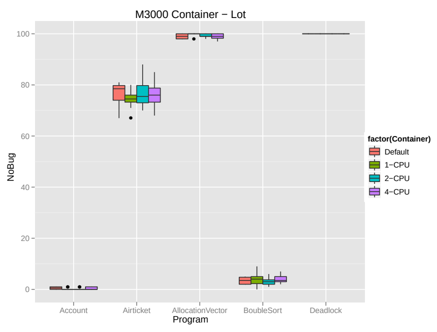
In order to have better insights of the hypothesis, we conducted a number of unpaired two-sample Student's t-test on the data drawn for each pair of containers created on each machine. The computed p-values were substantially greater than 0.05 even without application of the Bonferroni correction[5]. As an example, Table 3 reports the results of unpaired two-sample Student's t-test on the data obtained for Machine T1000 with the `lot` concurrency level. Hence, the study outcomes do not support the assumption of a casual dependency between number of cores and defect exposures on each machine. However, the differences are significant be-

[4]A $k - CPU$ container set asides k cores to the project.

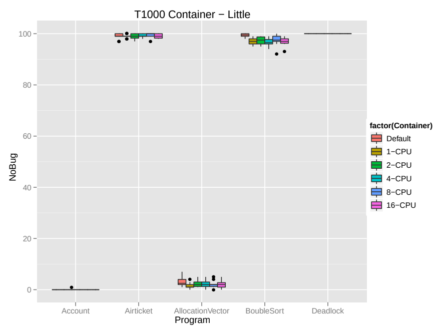
[5]The Bonferroni adjustment is used when there is an instance of multiple comparisons. An application of Bonferroni adjustment on the significance level $\alpha = 0.05$ is necessary here because of multiple comparisons required. Hence, the significance level should be adjusted to α/n where n times comparison is needed. The n values for M1000 and T1000 (i.e. for each program and concurrency level) are 6 and 15, respectively.



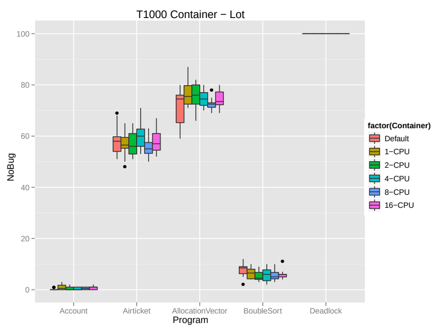
(a) M3000 - Concurrency Level:Little



(b) M3000 - Concurrency Level:Lot



(c) T1000 - Concurrency Level:Little



(d) T1000 - Concurrency Level:Lot

Figure 1: The box plots of variations of fault exposures for the Solaris containers defined.

Table 3: The p-values of the t-test performed on the number of faults exposed for the `lot` concurrency level on the T1000 computer system. D:Default, 1:1-CPU, 2:2-CPU, 4:4-CPU, 8:8-CPU, 16:16-CPU containers.

Containers	D	1	2	4	8	16
D	1	0.936	0.981	0.93	0.934	0.965
1	–	1	0.955	0.993	0.872	0.971
2	–	–	1	0.949	0.917	0.983
4	–	–	–	1	0.866	0.965
8	–	–	–	–	1	0.9
16	–	–	–	–	–	1

tween the computer systems used as well as the concurrency levels defined.

4. DISCUSSION

The main purpose of this explanatory study is to identify environmental parameters which influence concurrency fault exhibitions. Hardware architecture, software components, concurrency defect types, and concurrency levels are four major categories which are believed to have potential influence on runtime behavior of multi-threaded applications. In this paper, we presented a case study on a number of parallel applications to investigate whether multicore systems influence the behavior of paralleled applications. The outcomes do not support the assumption of any dependency between number of cores and fault exposures.

We have already pointed out that hardware architecture and concurrency levels may have some impacts on fault exhibitions. Although our case study focused on the influence of multcore platforms on fault exposures, the box plots depicted in Figure 1 illustrate different outcomes which have been obtained through various concurrency levels as well as hardware architectures. We observe that the variation of fault exposures widens when the number of threads or concurrency level increases. In addition, different outcomes are obtained for the computer systems selected. However, further insights studies are required.

Our work aims to assist researchers in configuring testbeds so that the frequency of exposing concurrency faults increases. In analogy, end users and practitioners can benefit from results in configuring operational field platforms to lessen the frequency of expositions of possible concurrency defects remaining. This work is part of our project to investigate the influential non-programmatic parameters on runtime behavior of multi-threaded applications.

5. REFERENCES

[1] J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande applications. In *The International Conference on The Practical Applications of Java*, pages 63–73, 2000.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. pages 167–178, 2010.

[3] Y. Eytani and T. Latvala. Explaining intermittent concurrent bugs by minimizing scheduling noise. In *Second Haifa Verification Conference (HVC 2006)*, pages 183–197, 2007.

[4] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Second Workshop on Runtime Verification (RV)*, volume 70, pages 142 – 157, 2002.

[5] Sun. Oracle Sun SPARC Enterprise Servers, 2010.