Sensitivity of Application Performance to Resource Availability

Boris Penev and Ajitha Rajan

School of Informatics, University of Edinburgh
s1249355@sms.ed.ac.uk, arajan@staffmail.ed.ac.uk,

Abstract. Existing literature has extensively explored and analysed the effects of shared resource contention to improve system throughput and resource utilisation. Nevertheless, a systematic study varying different resource availabilities and examining their combined effect on the performance of an individual application has not been conducted earlier. In this paper, we perform global sensitivity analysis using Monte Carlo simulations of three resource parameters – CPU availability, RAM availability and number of cores, and examine their effect on execution time of industry standard benchmark programs. In addition to understanding application performance sensitivity, our study also lends itself to identifying threshold levels of required resources, below which severe application performance degradation occurs.

1 Introduction

Users, whether on a desktop, laptop or mobile devices, tend to have several applications running concurrently. The performance of any one of these applications, given a hardware platform, is heavily dependent on the resources available, which in turn relies on the resource demands of applications and processes running at that time, shared resource contention and the operating system's resource allocation priorities and decisions. It is common knowledge that resource contention directly impacts application performance [6]. Mars et al. show more than 30% degradation in application performance (from running alone) when run alongside with another application on a multicore chip due to contention for the last level cache. However, varying levels of resource contention and, therefore, resource availability and its resulting impact on application performance is not well understood. Additionally, the user receives little to no feedback on the extent of expected performance degradation of an application due to resource contention and reduced availability.

Previous work has extensively explored and analysed the effects of shared resource contention [6],[10],[5]. The aim in all these studies has been to use the understanding of resource contention to improve system throughput, resource usage, and to enforce QoS and fairness across *all* running applications. *System*, rather than an *individual application*, performance has been the optimisation goal. Bhadauria et al. performed Parsec Benchmark characterisation for cache performance, DRAM speed and bandwidth, thread scaliability for solo benchmark execution (no other running applications) [1]. Dey et al. and Zhuravlev et al. study L1, L2 cache and front side bus contentions for multi-threaded applications individually and with one other co-running application [2],[11]. They show performance degradation occurs in the presence of certain co-running applications but not others. The effect of reduced resource availability as a result of one other co-running application was studied, however the effect of systematically varying different resource availabilities across the whole range (from 0 to 100%) has not been studied. The experiments presented in this paper attempts to fill

this gap. The goal in understanding this effect with respect to an application is to inform the user dynamically of when the performance of the application is expected to fall below acceptable levels due to lack of availability in required resources. The user can use this feedback to guide their actions, which might be to kill other running applications, or to have the operating system prioritise resources to the application of interest, or to simply accept the below par performance. Techniques in existing literature have never actively involved the user in the decision for resource allocation. This work is an attempt to give the user, if they choose, information and ultimately control in managing the performance of an application dynamically.

In this paper, we examine the effect on the execution or completion time of an application when the percentage of available CPU, RAM and number of cores *simultaneously* change. Other resources, such as cache, memory bus, network, I/O decives will also potentially impact the performance of an application. However, in this initial study we do not examine the effect of these other resources. We plan to consider them in the future.

We use global sensitivity analysis [9] to assess the impact of resource availability on application performance since parameter sensitivity is dependent on the interactions and influences of all parameters considered together on application performance rather than one parameter at a time. Sensitivity analysis can be achieved through Monte Carlo sampling [3] – repeated random sampling of input parameters from given distributions. In our setting, in each iteration, we randomly sample values for the three hardware parameters, set the resource availability to the respective values and record execution time of the application for each such setting. The input and output distributions are useful in assessing influence of input parameters on the overall output performance.

We used industry standard benchmarks in our experiments. For 5 of the 6 benchmarks, we found CPU availability was the most influential of the three resource parameters on execution time. For one of the benchmarks, number of cores was the most influential parameter. Our experiment results revealed that it is possible to set and monitor for threshold levels of required resources to avoid performance degradation of an application below user expectations.

2 Experiment

In our experiment, we use applications from EEMBC [8] benchmark suite, SPEC CPU 2006 [4] and a Linux kernel compilation program to study performance degradation from limited resource availability. In actual settings, limited resource availability arises from shared resource contentions with other running applications. We artificially set limits to availability of different resources in our experiments so we can systematically explore a large sample of values in the range of resource availabilities. We ran our experiments on a desktop running Scientific Linux 7.1 with 8 GB RAM, Intel i5-3470 quad core CPU at 3.20GHz. In the following sections, we discuss the tools we use to limit available resources, the Monte Carlo approach we use to understand the effect of limited resources, and finally the benchmark applications used in our experiment.

2.1 Setting Hardware Parameters

We consider application performance as its completion time and measure it using the time module in the python script used to run the benchmark and workload. Other performance measures can also be used in place of execution time, but we have not considered them in this paper. In addition to monitoring the output performance, we need to control and vary the input parameters which is amount of available resources–(1) **CPU**, (2) **Memory (RAM)**, and (3) **number of**

cores – for the application of interest. Cgroups is a kernel feature to limit the resource usage for a group of processes [7] and creates little to no overhead since it is a kernel tool. We use Cgroups to modify all three hardware parameters using the `cpuset, cpu` and `memory` subsystems. To achieve limits on cpu availability and number of cores, the `cpu` and `cpuset` subsystem manipulates the scheduler and its policy with respect to the group of processes in cgroup. The memory subsystem manipulates the memory allocation policy to achieve limits on the memory parameter.

2.2 Random Sampling of Parameters

In our experiments, we assume the parameters, CPU availability, #cores and RAM availability, follow a gaussian distribution. If the developer knows one or more of the parameters to follow a different distribution from gaussian, then we can sample according to that. For each benchmark, we sample and generate random settings of the parameters using their gaussian distributions. The CPU availability(in %) has a mean 50, deviation 20, left limit 25 and right limit 100. Number of cores parameter has a mean 2, deviation 1.5, left limit 1 and right limit 4. The memory parameter has a mean 1 GB, deviation 0.4 GB, left limit 1KB and right limit 3 GB. These limits have been chosen based on the hardware configuration and application characteristics (mainly for memory limits). Our base case environment has 50% CPU availability, 2 cores and 1GB memory and we use this as the mean of the normal distribution. The deviation in the distribution corresponds to the variation in the resource availability.

2.3 Subject Programs

We used the Embedded Microprocessor Benchmark Consortium (EEMBC) [8] that provides a diverse suite of benchmarks for microprocessors, microcontrollers and embedded devices with a total of 32 programs and workloads from the automotive, telecommunications, office, networking, and image processing domains. We also used the integer benchmarks from the SPEC CPU 2006 suite [4] (SPECint benchmarks), an industry standard benchmark suite to test CPU performance, with a total of 12 programs and corresponding reference workload for each. We also use Linux kernel compilation(`build_kernel`) as a benchmark. We disable non-essential drivers, networking, cryptography and virtualisation options. This lowers the number of dependencies, making it more tractable for our initial evaluation. We compile it using multiple cores to assess the sensitivity of execution with respect to cores.

3 Results and Analysis

We ran Monte Carlo simulation over the three hardware parameters for all 45 programs. Owing to space limitations, we show the results for only six of the 45 programs in the paper– `build_kernel`, `matrix01` from the EEMBC suite, `astar, bzip2, gobmk, h264` from the SPEC CPU2006 benchmarks, whose descriptions are shown in Table 1. Results for the remaining 39 programs from EEMBC and SPEC CPU 2006 benchmark suites can be accessed at "`http://homepages.inf.ed.ac.uk/arajan/results-ICTSS.pdf`".

Parameter sensitivity can be determined qualitatively by plots of input vs output values, or quantitatively by calculations of correlation coefficients [3]. Table 3 shows scatter plots for the 6 programs, plotting each resource availability against execution time. It is worth noting that although each plot only shows one resource at a time, all three resource availability values have been changed and sampled at the points in the plots.

Program	Description	Benchmark Suite	# Samples
`matrix01`	Matrix operations	EEMBC	1438
`build_kernel`	Building Linux kernel	n/a	101
`astar`	Pathfinding library for 2D maps	SPEC CINT2006	500
`bzip2`	File compression	SPEC CINT2006	500
`gobmk`	Plays the *GO* game	SPEC CINT2006	500
`h.264`	Video encoding	SPEC CINT2006	500

Table 1. Benchmarks and their desciption

Program	ρ CPU avail.	ρ Num. Cores	ρ RAM
matrix01	-0.55	0.05	0.03
build kernel	-0.46	-0.61	0.19
astar	-0.92	-0.07	0.09
bzip2	-0.93	-0.02	-0.01
gobmk	-0.92	0.04	-0.03
h.264	-0.93	0.02	-0.07

Table 2. Sensitivity indices of resource availability (Correlation Coeff. with exec. time)

We compute sensitivity indices for the parameters using Pearson's correlation coefficient (ρ) which represents the sensitivity of the output to input parameter variations [3]. The larger the absolute value of ρ the stronger the degree of linear relationship between the input and output values. A negative value of ρ indicates the output is inversely related to the input. Table 2 shows the sensitivity indices for CPU availability, number of cores and RAM with respect to a program's execution time. Negative values of ρ for resource availability is to be expected since execution time is typically lesser when there is more resource available.

From the plots in Table 3, we find that for all benchmarks, except `build_kernel`, CPU availability has the most effect on performance – increased CPU availability results in decreased execution time. This is also reflected by its significantly higher sensitivity index in Table 2 when compared to the other two resources, RAM and #cores. For these 5 benchmark programs, the sensitivity indices for #cores and RAM is small (absolute values in the range of 0.01 to 0.09) and only marginally different. As can be seen in the `Cores` and `Memory` plots for the 5 programs in Table 3, there is no discernible effect on execution time with increased availability. This is because the 5 benchmark programs are not optimised to run on multiple cores (designed as single core benchmarks) and have low memory requirements. Nevertheless, we found that for the `gobmk` benchmark when the available RAM was sampled at values less than 30MB, execution time increased dramatically – by 2 times for 24.1 MB RAM to 1195 times for 0.82 MB. We confirmed the effect was because of the RAM, rather than the other resources, by fixing the values for #cores at 2 and CPU available at 100%. Among available RAM values beyond 30MB, the difference in execution time was negligible. This implies that performance degradation is rapid when RAM available is below a certain threshold, which is 30MB for `gobmk`. We can use this information to notify the user when available RAM nears the threshold, using performance monitoring tools, to avoid this performance degradation. We did not see this phenomenon with available RAM for the other 4 benchmarks.

`build_kernel` exhibited a different trend from the other benchmarks. We find that execution time is most affected by changes in #cores (highest sensitivity index) – an increased number of CPU cores results in decreased build time. This is seen in the plot for `build_kernel Cores` in Table 3 as the number of cores increases from 1 to 4, the execution time observed reduces significantly. The make tool and the Linux kernel build processes are well optimised for multi-

threaded execution on multicore processors. Although not as high as #cores, CPU availability also has a significant impact and sensitivity index. Increased CPU availability results in decreased build time as seen in the `build_kernel Availability` plot. Thus, #cores and CPU availability are both important for `build_kernel` execution time. Changes in RAM availability produced no noticeable result in the build time and this may be because memory requirements for `build_kernel` are not high.

Using the information on parameter sensitivity, the plots in Table 3 and the raw data from running the experiment, one can identify thresholds for the different parameters beyond which performance degradation is unacceptable for the user. Identifying the threshold of unacceptability will require knowledge of expected performance levels (as per user) which can be inferred using historical data or user input. For instance, for the `bzip2` benchmark, if we assume unacceptable performance is execution slowdown (from best case time) greater than 33%, then we pick threshold 60% for CPU availability (see plots in Table 3), since execution time increases by more than 33% for lower than 60% available CPU. RAM and # cores do not have a profound effect on execution time, so we do not pick thresholds for these resources for `bzip2`.

4 Conclusion

For the industry standard benchmarks used in this paper, we found that varying the input parameters using Monte Carlo simulations and examining the effect on execution time was an effective way to study application performance sensitivity to availability of different resources. Systematically analysing global resource sensitivity has not been studied previously. We found that such a study helps in identifying thresholds of unacceptable performance degradation. Resources can be monitored for these threshold values and the information communicated to the user or the system to act upon as needed.

References

1. M. Bhadauria, V. Weaver, and S. McKee. Understanding parsec performance on contemporary cmps. In *IISWC 2009*, pages 98–107. IEEE, 2009.
2. T. Dey, W. Wang, J. Davidson, and ML Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *ISPASS 2011*, pages 76–86. IEEE, 2011.
3. DM Hamby. A review of techniques for parameter sensitivity analysis of environmental models. *Environmental monitoring and assessment*, 32(2):135–154, 1994.
4. John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
5. H. Jin, R. Hood, J. Chang, J. Djomehri, D. Jespersen, K. Taylor, R. Biswas, and P. Mehrotra. Characterizing application performance sensitivity to resource contention in multicore architectures. *NASA Ames Research Center, Tech. Rep. NAS-09-002*, 2009.
6. J. Mars, N. Vachharajani, R. Hundt, and ML Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th CGO*, pages 257–265. ACM, 2010.
7. P. Menage, P Jackson, and C Lameter. Cgroups. *Available on-line at: http://www. mjmwired. net/kernel/Documentation/cgroups. txt*, 2008.
8. J.A. Poovey, M Levy, S Gal-On, and T Conte. A benchmark characterization of the eembc benchmark suite. *Micro, IEEE*, PP(99):1–1, 2009.
9. H. Wagner. Global sensitivity analysis. *Operations Research*, 43(6):948–969, 1995.
10. C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *ISPASS 2010*, pages 76–86.
11. S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.

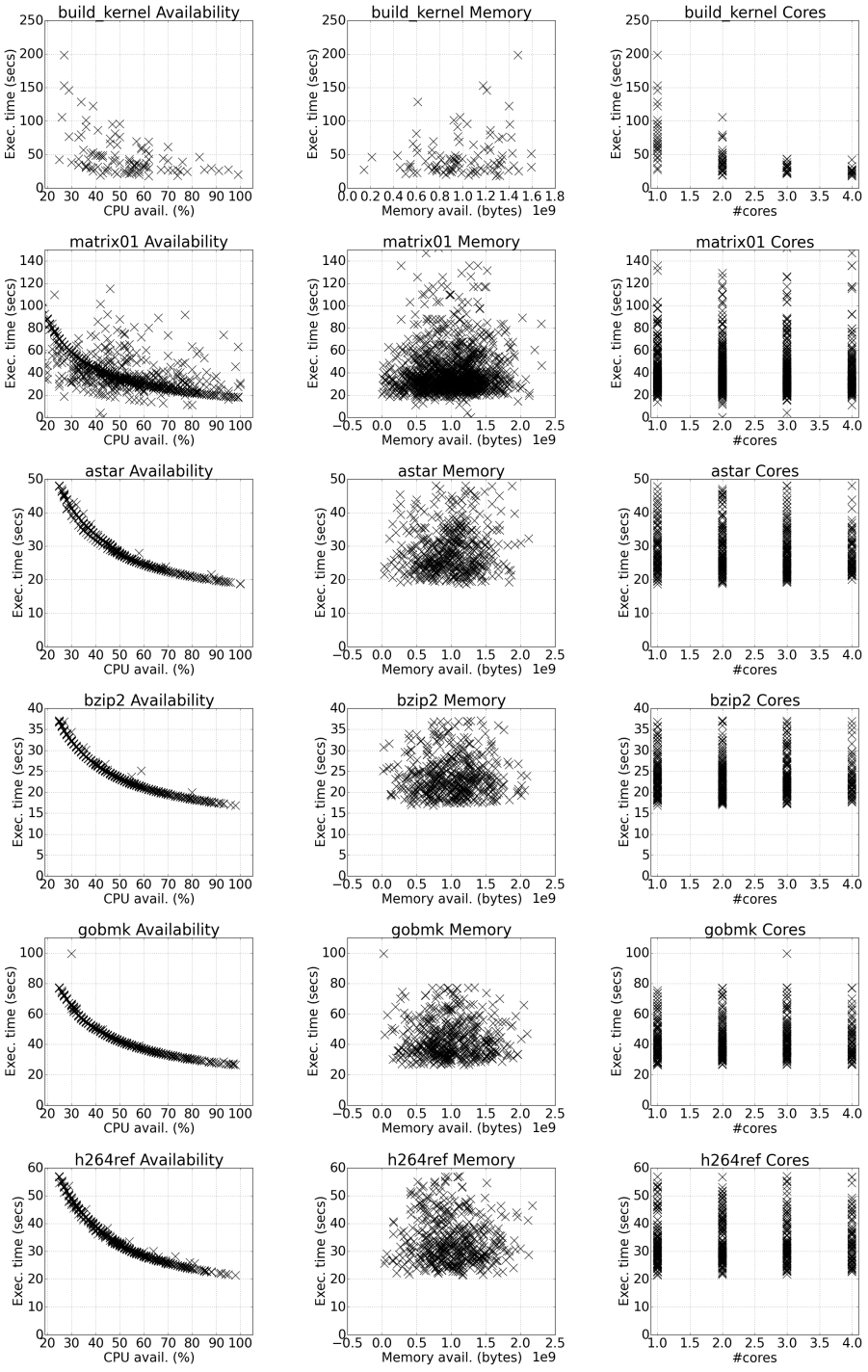

Table 3. Scatter plots of execution time versus resource availability (CPU, RAM, #cores) for build_kernel, 1 EEMBC and 4 SPEC benchmarks