Reducing Performance Non-determinism via Cache-aware Page Allocation Strategies

[1]Michal Hocko [1,2]Tomas Kalibera

[1]Dept. of Software Engineering
Faculty of Mathematics and Physics
Charles University

[2]Dept. of Computer Sciences
College of Science
Purdue University

kalibera@dsrg.mff.cuni.cz

ABSTRACT

Performance non-determinism in computer systems complicates evaluation, use, and even development of these systems. In performance evaluation via benchmarking and simulation, non-determinism requires long executions and more complex experiment design. Real-time systems are hard to dimension and tune with non-determinism. The slower benchmarking also slows down system development, as it takes developers longer to see performance implications of their modifications.

Cache-unaware physical page allocation in an operating system is believed to be a significant cause of non-determinism, but there is no published empirical study that would confirm it.

We provide such a study for the Linux operating system, comparing the default cache-unaware page allocation strategy to known cache-aware strategies, page coloring and bin hopping. We have implemented a framework for page allocation strategies in the Linux kernel, employed it for these two strategies, and measured the non-determinism on a large and diverse set of benchmarks. We propose a statistical technique which allows to classify different kinds of performance non-determinism and evaluate their magnitudes. Application of our technique reveals that the two strategies do reduce performance non-determinism without significantly increasing mean response time.

Categories and Subject Descriptors

D.4.2 [**Software**]: Operating Systems—*virtual memory, main memory, allocation/deallocation strategies*; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Experimentation, Performance

Keywords

performance non-determinism, regression benchmarking, software performance, statistical methods

1. INTRODUCTION

Current computer systems are typically being designed for highest mean performance. Performance non-determinism is of less concern. It is harder to measure the non-determinism, and for many non-interactive and non-realtime applications it really does not matter once the applications are developed. This is however not the case for evaluation and consequently also for development of these applications.

Performance evaluation methods become more complex and time consuming once they have to cope with non-determinism. Benchmarks have to be designed such that the source of non-determinism is part of what is repeated. Larger non-determinism then requires more repetitions, consuming more time. Also, different levels of non-determinism typically require more complex analysis techniques, such as ANOVA. A concrete instance of this problem is non-deterministic behavior of just-in-time compiler in a Java Virtual Machine. It was shown in [1] that multiple executions with multiple compilation plans are needed, and ANOVA based evaluation was proposed. Non-determinism also complicates simulation studies [2], requiring more executions and modeling of the non-determinism present in a system.

Decisions made during software development of complex systems often have to be based on performance evaluation of intermediate versions. Thus, complex and long running benchmarks also indirectly complicate software development. In regression benchmarking, benchmarks are often run regularly and automatically during development and their results are automatically summarized. Many projects, including GCC, Eclipse, Linux kernel, TAO, and Mono, have automated regression benchmarking. Ideally, regression benchmarking also automatically detects regressions. To have an acceptable ratio of false positives, automated detection of regressions is then lengthier with any increase of non-determinism and more complex with any new source of non-determinism. A particular issue with performance non-determinism that complicated software design was reported in [3]. Performance changes due to renaming of symbols (which appear non-deterministic in sufficiently large software) were larger than performance changes due to optimizations in a Java Virtual Machine. The performance results were thus misleading the virtual machine designers.

The magnitude of the problem of non-deterministic performance in benchmarking can be surprisingly large and also easy to demonstrate. Figure 1 shows response times measured with the popular FFT Scimark2 benchmark [4] on the C#/Mono platform. The same benchmark is executed 5 times on the same system with no background noise, in each execution measuring the response time of the

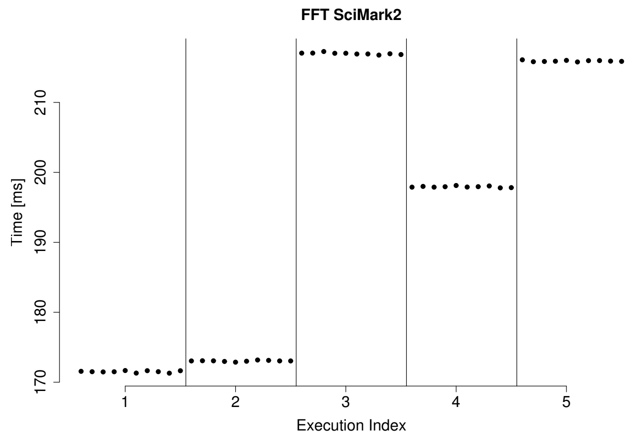

Figure 1: Results from 5 consecutive executions of FFT SciMark2 benchmark in C#/Mono; 10 measurements per execution. Differences between executions reach 30%.

same operation 10 times. Performance within executions is stable, but performance between executions differs by as much as 30%.

With the advent of user-space real-time applications running in modified versions of Linux [5, 6], the ability to evaluate worst-case performance is becoming increasingly important even for correct functionality of applications. Even sacrificing mean performance for easier or more reliable evaluation of worst-case execution time thus becomes a viable option. A concrete example of how non-determinism in execution makes real-time systems tuning harder is the tuning of a real-time garbage collector's rate, for example when using the Metronome collector [7]. The optimal rate that provides reasonable speed, but does not lead to running out of memory (and thus crashing), is usually found experimentally by re-starting an application for different rate values and heap size limits. Non-determinism thus both complicates the tuning and leads to parameters that use resources less efficiently, due to necessary over-dimensioning (high collector rate, large heap).

A significant source of non-determinism in response time is non-deterministic cache-unfriendly physical page allocation [8, 9], which is performed by the operating system out of control of applications. Cache unfriendly physical page allocation may also lead to unnecessarily large number of cache misses, and thus harm mean response time. Earlier cache-aware allocation strategies [10], out of which page-coloring and bin-hopping made it to mainstream operating systems, were primarily aimed at reducing these unnecessary cache misses. The two strategies however also optimize the page allocation for deterministic numbers of cache misses. Page coloring, used by Solaris [11], Windows [12], and Free BSD [13], works on the assumption that memory accesses are spatially-local, and thus makes close virtual memory pages not to collide in the cache. Bin hopping, used by Digital Unix [14], assumes that memory locations are accessed mostly in a fixed order, and thus makes pages allocated soon after each other not to collide.

Despite earlier results showing 10-20% improvements in mean performance thanks to these strategies [10], the gains are expected to be smaller with current highly-associative caches. In Linux, the gains were feared to be in general applications smaller than the overhead of supporting the strategies [15]. Thus, earlier attempts to incorporate page coloring to Linux were rejected (Linux 2.2 [16], Linux 2.4 [17]), though no thorough performance evaluation was carried out. Moreover, mean performance was ultimately favored over predictability at that time.

Our contributions are:

- Linux kernel framework for cache aware page allocation with page coloring and bin hopping strategies as modules.
- Statistical evaluation methodology for classification and evaluation of non-determinism that scales to many benchmarks and provides results which are easy to interpret. It builds on our earlier methodology that lacked this scalability.
- Experimental evaluation run on two platforms using a large amount of benchmarks, both in non-realtime and real-time kernel, which evaluates non-determinism and mean performance with different allocation strategies.

The source code of our kernel modification can be downloaded from [18]. The rest of the paper is structured as follows. In the introduction of Section 2 we briefly classify sources of performance non-determinism. In the rest of Section 2, we explain the technical OS-level and hardware-level reasons for page allocation being an important source of the non-determinism. In Section 3 we outline the architecture and algorithm of the Linux kernel extension in the context of the Linux kernel. In Section 4 we present our statistical methodology for classification and evaluation of the non-determinism. In Sections 5 and 6 we present and analyze the empirical results.

There are two mostly disjoint blocks of our presentation: (a) the hardware and operating system level description of the page allocation problem and the kernel modifications and (b) statistics and benchmarking content, describing the evaluation methodology and the actually measured results. The analysis of the causes of slowdown observed with page coloring and bin hopping, as well as the conclusion, build on material presented in both of these blocks.

2. SOURCE OF NON-DETERMINISM

Current hardware and software systems include numerous sources of non-determinism or randomness: some randomness originates from the hardware (such as chip temperature, disk rotation time, or hardware interrupts resulting from network traffic), some randomness is intentional (randomized algorithms for complex problems, randomized system behavior as an obstacle for attackers), and some unpredictability results from very complex deterministic processes that may be even unknown to observers (kernel page allocator, system scheduler, or simply a pseudo-random number generator).

All of these types of non-determinism exist in current systems and impact their performance. If all these sources acted independently of each other and of the system state, performance evaluation would still be quite easy – some repetitions and basic statistical evaluation would suffice. Unfortunately, this is not the case. Although some sources of non-determinism, such as context switches or hardware interrupts, impact performance of executed code quite independently of system or application state, there are at least two exceptions:

- Some sources of non-determinism only impact application performance at compile time (*non-determinism in compilation*), and thus a performance tester has to re-compile the tested application and re-run all tests several times to get representative. results [19]
- Some sources of non-determinism only impact whole executions of an application (*non-determinism in execution*), and thus a performance tester has to re-run all tests several times to get representative results [9].

In this paper, we focus on the second type of these two, on non-determinism in execution, and particularly on one of its main causes: page allocation.

2.1 Non-determinism due to Page Allocation

Current processors typically implement virtual memory. They provide applications with a linear (virtual) address space, which is internally divided into blocks (pages) of a fixed size, i.e. 4KB. When a virtual page is accessed by the application, the processor with the guidance of the operating system finds out the corresponding block of physical memory (physical page) that should handle the memory access.

A virtual page is usually identified by several most significant bits of the virtual memory address (page index). The remaining bits then form the page offset. The virtual page index translates to a physical page index, where the same page offset is used. As an example, 32-bit Intel architectures have 32-bit virtual addresses, where 20 most significant bits form the page index and the remaining 12 least significant bits form the page offset. When the processor, with the aid of the operating system's physical page allocator, finds out a physical address for a memory access, it proceeds through a hierarchy of caches potentially to the main memory.

The types of caches and their structure depend on a processor type, but the following assumptions commonly hold:

- The largest difference in access time is between the last level cache and the main memory.
- The caches are real-indexed.
- The caches are set-associative.

Real indexing means that the data stored in the cache is identified by physical (real) memory addresses. *Set-associativity* is a compromise between cache utilization and internal cache complexity, as well as power consumption. In order to explain set associativity, we first explain the two extremes: full associativity and direct mapping. In a *full-associative cache*, data from any memory location can be stored in any location (cache line) in the cache. Therefore, only *capacity misses* can occur – accesses to data that is not in the cache when the cache is full. Last level caches are usually not fully associative for technological limitations – they would be too expensive and take too much power [20]. In a *direct-mapped cache*, data from each memory location has only one location in the cache (cache line) where it can be stored. Consequently, a cache conflict – and then a cache miss – may occur even if the cache is not full. These misses are known as *conflict misses*. In a *set-associative cache*, data from each memory location has a set of several cache lines in the cache where it can be stored. The sets of cache lines are disjoint. The cache thus conceptually behaves like a combination of a single direct-mapped cache and multiple small full-associative caches: the direct mapping is used to select one set for each memory location, and the set then behaves like a full associative cache.

Cache misses have a profound impact on performance – a miss requires that the access is directed to main memory, which is *at least* one order of magnitude more expensive than accessing even the last level of the cache. Capacity misses happen due to limited cache size relative to the working set size — that is, the amount of memory an application actively uses. The working set size depends on the algorithm of the application and its implementation and can be controlled by the programmer. Conflict misses however happen due to limited cache associativity and sub-optimal placement of data in physical memory. The data placement is partially set by the programmer and compiler (page offset) and partially by physical page allocator (page index). In current commonly used operating systems, including Linux, the programmer has no chance to influence the selection of the page index.

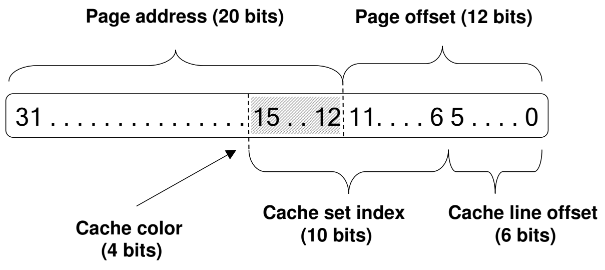

Figure 2: Physical addressing and L2 cache addressing in Dell Precision 340 (Intel 32-bit, 512K 8-way set associative L2 cache, line size 64B).

The magnitude by which the page allocator influences the number of conflict misses, and thus performance, depends on cache parameters and on the page size. In Figure 2 we show a concrete example – the Dell Precision 340. The page size is 4KB, with two levels of cache; the last being 8-way set associative, with a capacity of 512K and a cache line size of 64B. The cache therefore has $512K/(8 * 64B) = 1024$ sets. The least significant 6 bits of the physical address select the 64B of a cache line. The following 10 least significant bits of the physical address select one of the 1024 sets, and the remaining bits of the physical address do not impact the location in cache.

At the same time, the 12 least significant bits of the physical address are the page offset (page size is 4KB), and the remaining 20 bits are the page index. It follows that of the 20 bit page index, the 4 least significant bits are the most important. The page allocator therefore has the power to determine to which of the 16 *bins* of 64 sets data from a virtual memory page would fall in the cache. These indexes of bins that the page allocator would select are sometimes called *cache colors*.

The *page coloring* allocation strategy can be quite easily implemented by re-using the *cache color* of the virtual page index for the physical page index. The cache color are the bits where the cache offset and page index overlap – in our example, that would consist of the four least significant bits of the page index. As a result, data from several (16 in the example) consecutive virtual pages would fall into different sets in the cache, and thus would not cause conflict misses with each other. To prevent conflict misses between different processes, the page colors can be hashed by process identification number [10].

The *bin hopping* allocation strategy then simply defines an ordering of bins and remembers the bin of the last page allocation. Each page is allocated from the bin following the last used bin in the defined ordering.

Both page coloring and bin hopping are however only heuristics that rely on a given memory access pattern. When applications use a different pattern, they still might suffer from conflict cache misses, and thus from sub-optimal performance. The resulting non-determinism in performance comes from the unpredictable state of the page allocator at application start-up and, in a typical case when the system is running multiple applications or services at a time, also from unpredictable changes of the allocator state during application run-time.

3. LINUX KERNEL EXTENSION

The kernel extension we have implemented was intended to become a basis for further research on tuning performance through virtual-to-physical mapping, rather than only a quick implementation of page coloring and bin hopping. The main design goals for the extension and required user-land libraries were: completeness, extensibility and non-intrusiveness.

By *completeness* we refer to wide support for tuning virtual-to-physical memory mapping: support for process (local) allocation strategies, support for even multiple allocation strategies for a single running process, support for user-space control over existing virtual-to-physical mapping, and control over allocation strategies for newly created mappings.

By *extensibility* we refer to the option to implement allocation strategies as kernel modules, without the need for further modification of the physical page allocator.

Our extension is designed to be *non-intrusive* for system performance, potential bugs, as well as code maintenance. To achieve this goal, we have implemented a clone of the existing Linux page allocator. Our clone re-uses the existing data structures and differs in functionality to a minimum extent possible. Therefore, there should be no performance overhead for applications that do not "activate" the extension. Even if there was a bug in the extension, it would not appear in a running system before the extension would actually be used (i.e. a page allocation strategy would be set for a particular process). Also, thanks to the code separation from the original page allocator, further maintenance of the original allocator is not made harder by the extension.

In the rest of this section, we provide additional technical details about the algorithm and architecture of the kernel extension.

Linux uses the *buddy algorithm* for page allocation [21]. The basic idea is simple. Only aligned blocks of 2^n pages are allocated. The allocator maintains a list of free blocks for each n (order) and serves each allocation request by a smallest-order block. If there is no free block of such order, a block of higher order is halved: one block is used for the allocation request and the other is stored among free blocks of lower order. The two blocks are called "buddies", hence the name for the allocator. A block can only be merged with its buddy.

The allocator we have created differs from the original allocator in the selection of free blocks. The original allocator always takes the first free block of correct order, and if a block of larger order has to be divided, it always uses the beginning of that block for the allocation.

Our allocator selects free blocks based on hints. A *hint* contains information on required allocation strategy and some parameters for the strategy. An *allocation strategy*, typically implemented as a kernel module, is represented by callback functions. One of the callback functions implements the core of the allocation algorithm: given a free block of physical pages, it decides which part of the block can be used for the allocation, if any.

On allocation request, the allocator sequentially checks all free blocks using the strategy's callback function. If no free block of a given size is accepted by the strategy, the allocator follows by scanning larger blocks, dividing them if accepted by the strategy. If no free block of any order is accepted by the strategy, the allocator uses any free block.

Hints for page allocation are indexed by virtual addresses and are stored in each process's memory descriptor. Hints can be specified for individual virtual addresses and as a default for the whole virtual address space of a process; the default hint is used if no hint is specifically set for a particular virtual address.

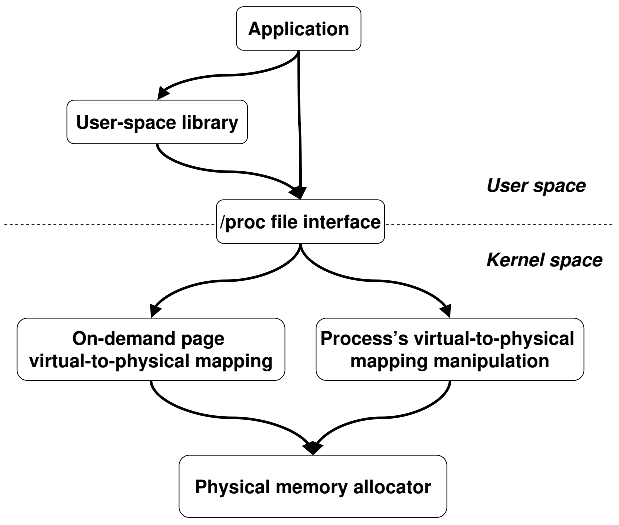
Application
User-space library
/proc file interface
User space
Kernel space
On-demand page virtual-to-physical mapping
Process's virtual-to-physical mapping manipulation
Physical memory allocator

Figure 3: Components of implemented kernel extension.

With the strategies we have implemented, these options allow, for example, the specification of exact physical addresses or exact page colors of pages, the selection of page coloring strategy on a per-data-structure level, as well as using any of the available strategies globally in a process' virtual address space.

Hints can be set from user space via a specialized API, implemented by IOCTL calls on special files in /proc filesystem. To simplify usage of the API, we have created a user-space library.

Optionally, hints can be inherited on fork, and thus set for newly executed processes. It is therefore possible to run binaries of existing applications without a need for modification or dynamic loader tricks. Inheritance of the default hint for fork and/or exec can be enabled or disabled during kernel compilation time, and thus the setting holds for the whole running system.

In addition to page coloring and bin hopping, we have implemented two extra strategies: exact strategy and modulo strategy. The *exact strategy* can be used for specifying exact mappings of virtual to physical pages, which can help in creating a repeatable environment between successive executions of a process.

However, as certain physical pages may become occupied by the kernel or another process, it might be more useful to only re-create the coloring of physical pages. This can be achieved by the *modulo strategy*, which allows for the specification of the physical page colors for virtual pages. For simplicity, the page coloring strategy is implemented as a special case of the modulo strategy, where the colors of virtual pages are used as physical page colors.

Via the provided API, applications not only can specify hints for their virtual-to-physical memory mapping, but can also modify already existing mappings. Our kernel extension can even re-map certain virtual pages that occupy the demanded physical pages, employing the page migration capabilities of the current kernel. Further, applications can read their virtual-to-physical mapping from a special file in the /proc filesystem.

A complete view on the components of the implemented extension is given in Figure 3. More information on the extension can be found in [22, 18]. The extension can be downloaded from [18].

4. EVALUATION METHODOLOGY

We base our evaluation on benchmark experiments and their statistical processing. The main objective is the comparison of mean response time and non-determinism in response time of applications running in the original Linux kernel vs. modified Linux kernel with page coloring and bin hopping.

In addition, we want to fraction the non-determinism into non-determinism in execution, which is only observable between different executions of a benchmark experiment, and non-determinism in measurement, which is also observable between measurements of repeated operations within a single execution. As mentioned, non-determinism in execution is of much higher concern for performance evaluation, thus the fractioning.

4.1 Measurement Techniques

In order to get results representative of a widest possible range of applications, we have selected a diverse set of benchmarks. A detailed description of individual benchmarks is provided in Appendix A.

The dimensions of diversity of the benchmarks include: native (C) vs. virtualized (C#) environments, different problem types (numerical, remote communication, compilation, cryptography, multimedia compression, lossless compression, and other), varying problem size of numerical benchmarks (cache size, double and half of the cache size), varying measurement granularity (whole process execution, execution of one operation), different just-in-time optimization options of the C# runtime (all optimizations supported, a default optimizations set), memory allocation in benchmarks measuring a single method call (within each repetition of the measurement, once for all measurements), and authorship (benchmarks that we wrote or modified, unaltered well-known benchmarks).

We have run the benchmarks on two platforms: Dell Precision 340 (Intel Pentium 4 2GHz, 512M RAM, 512K 8-way set associative L2 cache) and Dell Optiplex 745 (Intel Core 2 Duo 2.1 GHz, 2G RAM, 2M 8-way set associative L2 cache), both with Debian GNU/Linux 4.0 (Etch) with the 2.6.22.6 Linux kernel.

We used two kernel configurations. For the benchmarks of the original Linux kernel, we compiled the vanilla kernel with Debian 4.0 configuration and the Perfmon2 [23, 24] patch for hardware performance monitoring. For the benchmarks of our extension, we only added our extension to the previous kernel configuration. We have compiled page coloring and bin hopping as modules, loading only one of the modules for individual experiments. The Perfmon2 patch allowed us to access hardware performance counters of the processors, so that we could measure the number of L2 cache misses and number of CPU ticks (time).

The experiments were fully automated, including repeated executions of the same benchmarks, reboots for each new benchmark experiment (multiple executions of a benchmark with specific configuration on specific kernel) and reboots with a different kernel.

All benchmarks were run in a special run-level with no interfering system services. In particular, the network interface has been shut down while running the experiments. In order to avoid distortion of the results, the benchmarks collected only raw data. All the evaluation was carried out off-line.

We have additionally re-run the experiments for a subset of the benchmarks in a real-time Linux kernel. We used Linux 2.6.22.9 with RT patches shipped with Ubuntu 7.10. We run the experiments on Dell Precision 380 (Intel Pentium 4 3.8GHz, 3G RAM, 2M 8-way set associative L2 cache). To simulate conditions similar to a realistic real-time system, we run the benchmarks with FIFO scheduling and highest available (real-time) priority.

4.2 Metrics for Statistical Evaluation

Our statistical evaluation follows two basic requirements: easy interpretation and reliability of results. For reliability we estimate precision of all results and make sure that results whose differences are below the precision are not considered different. We estimate the precision using confidence intervals. For several statistics described below, we calculate a 95% two-sided equi-tailed confidence interval and regard its half-width as the measurement error. For different statistics we use different methods to construct this confidence interval. We base our comparisons on overlapping of confidence intervals. If the intervals do not overlap, we conclude that with certain confidence the results are different. Although we do not use this cryptic wording further in the text, if the intervals do overlap, we only can conclude that we do not have enough evidence to show that the results would be different. This simple comparison method is described in [25, 26]. We did not use two-sample and multi-sample statistical tests, as they don't lend themselves well to summarizing the comparisons. Being more defensive in the comparisons is actually a good approach in face of deviations from the assumptions made, but not always met by real data (i.e. independence of consecutive measurements, which is often violated by memory managers re-using memory space using systematic patterns, which we have observed in Mono). Having enough measurements could however still lead to unrealistically narrow confidence intervals and incorrect conclusions. We therefore complement the results by comparisons, where we only report difference, if confidence intervals do not overlap and if the estimates differ more than by a given percentage.

Despite the described diversity of the benchmarks, the only significant difference that matters for the statistical evaluation methods is the measurement granularity – whether a benchmark experiment's execution produces a single measurement, or multiple measurements of the same operation. From the benchmarks we use, the Csibe benchmarks only provide one measurement, while all other benchmarks provide multiple measurements. We follow by a detailed description of evaluation methods for different statistics and these two benchmark types.

Mean response time.

In benchmarks that produce only a single number per execution (Csibe), we estimate mean response time. In our experience, mean response times are typically far from following normal distribution. We therefore use a non-parametric interval, particularly two-sided equi-tailed bootstrap confidence interval, calculated by the percentile method [27].

Results of benchmarks that produce multiple measurements of the same operation per one execution (non-Csibe) could be in theory evaluated using the same method. Intuitively, we could again estimate the mean by the average of all measurements – we could take a single measurement from each execution to get the input data for the previous method. This would however be a waste of the measured results, and thus of the time available for running the experiments. In other words, in a given amount of time, we would get less precise results.

We therefore use a more efficient method for calculating the confidence interval, which is based on analysis of variance (ANOVA) [27, 28]. In [19, 29], we describe this method and evaluate it on a similar scenario to the one presented here. The method constructs the interval using variances within executions and variances in executions' means.

We carried out all the presented statistical evaluation using the R language scripts (about 1200 lines of code).

Non-determinism in measurement.

Non-determinism in measurement can only be estimated in benchmarks that produce multiple measurements per execution (non-Csibe). In these benchmarks, we report coefficient of variation (standard deviation divided by mean) calculated from coefficients of variation in measurements of individual executions. We calculate the precision of the coefficient of variation estimate using the percentile bootstrap confidence interval.

Unclassified non-determinism.

In benchmarks that only produce a single measurement per execution (Csibe), we cannot factor the non-determinism into non-determinism in execution and non-determinism in measurement. We thus quantify this (unclassified) non-determinism with coefficient of variation and estimate the precision with the percentile bootstrap confidence interval.

Non-determinism in execution.

The problem of factoring the non-determinism into non-determinism in execution and non-determinism in measurement can be formulated in statistics as analysis of variance (ANOVA). Classical ANOVA methods however assume normal distribution of errors (fluctuations in performance due to non-determinism), which is far from the multi-modal nature of our data.

Instead of classical one-way ANOVA, we thus use a simpler non-parametric method [9], which measures how many times greater is the standard deviation between executions than the standard deviation within executions (*impact factor* of non-determinism in execution). An impact factor of 1 means that there is no non-determinism in execution, and thus all the observed non-determinism is in fact non-determinism in measurement. The greater the impact factor, the greater (multiplicatively) is also the non-determinism in execution than the non-determinism in measurement. Our experience shows that the impact factor is often in tens and can also be around one hundred.

We estimate the impact factor using a method based on statistical bootstrap [27] as follows. By random, we take groups of measurements from different executions and groups of measurements from a single execution. For each two groups we calculate ratios of standard deviations. Out of the ratios we take the mean as the impact factor estimate. We then again estimate its precision using the percentile bootstrap confidence interval. The resulting experimental measure is thus a single value, impact factor, and its precision, half-width of the confidence interval.

Although we used the median to estimate the impact factor in [29], we now prefer the mean for its better robustness in case of multi-modal data with modes of very similar size – the median can be unstable in the presence of random fluctuations in the mode sizes. Such type of data is typical for performance measurements, where the modes correspond to different states of the system.

4.3 Quantitative Summary

For each benchmark, we apply the evaluation methods described above to compare mean response time and non-determinism in response time of the current Linux kernel vs. page coloring and bin hopping. With thousands experiments we run, we also make thousands of statistical comparisons. Based on these comparisons, we want to draw a conclusion on general applications: does page coloring or bin hopping provide lower response time and/or non-determinism than current Linux kernel ?

To make such a generalization realistic, we have balanced the experiments so that none particular configuration (hardware platform, problem size, problem type, software platform, just in time compilation, memory allocation granularity) is represented more often than other. This balancing required evaluating Csibe and non-Csibe benchmarks separately.

We perform the described statistical comparisons on each triplet of experiments that differ only in page allocation strategy (page coloring, bin hopping, default kernel). This means that all experiments in a triplet agree in platform, benchmark, and other configuration parameters except for the strategy. For each triplet, we rank each of the three strategies as best, second best, or third best in respect to a particular metric. Because our comparisons take precision into account, there may be indecisive results: two or three strategies may be ranked as best, or two may be ranked as second best.

We summarize rankings of each strategy from all triplets, summing up in how many triplets the strategy was ranked first, second, or third best. As the triplets represent different configured test applications modeled by the benchmarks, we can extrapolate in what percentage of applications a particular strategy is better or worse than another strategy. We have put a significant effort into covering as many applications by the benchmarks and their configurations as possible, as detailed in Appendix A. We provide results of the quantitative summary in Section 5.

4.4 Qualitative Summary

Whenever in a triplet the default strategy provides statistically significantly different results from page coloring or bin hopping, we also quantify this difference. We calculate relative overheads – absolute difference divided by the result for default strategy – and report the ratio as percentage. With response time, we refer to negative result as speedup and to positive as slowdown. With non-determinism, we refer to negative result as reduction and positive result as increase.

For mean response time, we also calculate mean speedup (slowdown) over all triplets, not just the ones with statistically significant difference. We are averaging ratios of absolute times with page coloring (bin hopping) over default strategy. Since averaging ratios, we use geometric mean and convert the result to relative slowdown/speedup.

We show selected qualitative results in Sections 5.5 and 5.6 to illustrate the magnitude of the effect page allocation strategy can have on response time. The results supplement to the quantitative summary, which only can be generalized to other applications. Moreover, the qualitative results are to be less trusted than the quantitative ones, as there are no precision estimates of the final values. We believe that more reliable insight into how "big the problem is" can be gotten from comparisons that only regard results differing in a given percentage as different, which are part of the quantitative results presented in Section 5

We also provide a graphical qualitative comparison of the non-determinism in execution with different page allocation strategies in 5.5. We show an empirical cumulative distribution function of impact factor, drawn on a logarithmic scale, based on all experiments executed with non-Csibe benchmarks (non-determinism in execution can only be evaluated for non-Csibe benchmarks).

5. EVALUATION RESULTS

5.1 Mean Response Time

Strategies are compared with respect to mean response time in Figures 4 (non-Csibe benchmarks) and 5 (Csibe benchmarks).

From Figure 4 it follows that in 77% of non-Csibe benchmarks, neither page coloring nor bin hopping provided significantly better mean response time than the default kernel strategy (the default strategy is ranked best). The default kernel strategy was outper-

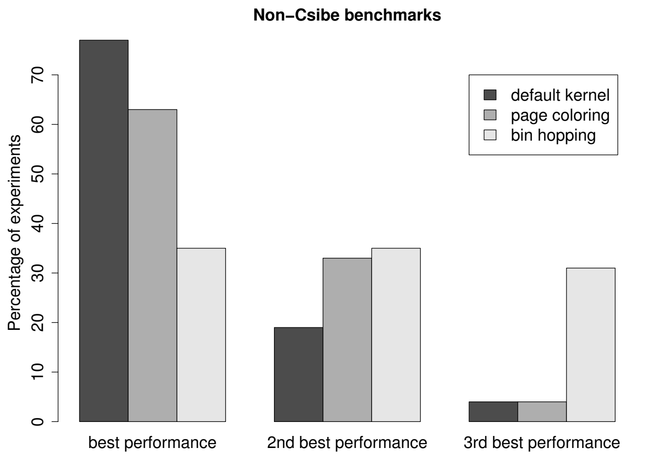

Figure 4: Mean response times measured by non-Csibe benchmarks using different page allocation strategies. In 77% of the experiments, neither page coloring nor bin hopping provided significantly better mean execution time than the default kernel strategy (the default strategy is ranked best).

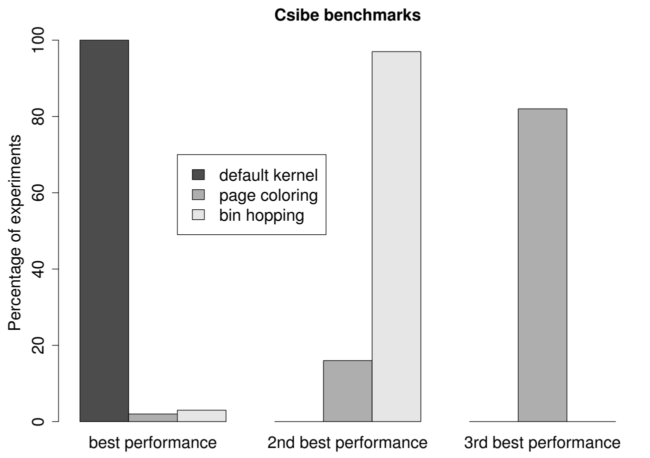

Figure 5: Mean response times measured by Csibe benchmarks using different page allocation strategies. The default kernel strategy was "never" outperformed by page coloring or bin hopping.

Benchmarks	Strategy	% of Experiments for % Mean Diff			
		0	1	3	10
Non-Csibe	Page Coloring	63	71	85	88
	Default	77	81	83	88
	Bin Hopping	35	42	48	56
Csibe	Page Coloring	1	13	62	100
	Default	100	100	100	100
	Bin Hopping	51	59	100	100

Figure 6: The strategies providing the best mean response time. If we only trust differences above 3%, page coloring is the best choice for non-Csibe benchmarks (highest percentage, 85%, of cases when it was ranked first), but not for Csibe ones.

formed by one of the other strategies only in 19% of experiments, and only in 4% of experiments by both of them.

If we only compare page coloring and bin hopping using the same method (this cannot be seen from the figures), bin hopping was outperformed by page coloring in 46% of experiments, but page coloring was outperformed by bin hopping only in 4% of experiments. Note that when comparing only two strategies, there is less compared experiments, and thus the percentages are not comparable with percentages coming from comparisons of three strategies.

Figure 5 shows that in Csibe benchmarks, the default kernel strategy was not outperformed in any of the experiments (in rounded 0% of experiments [1]) neither by page coloring nor by bin hopping. If we only compare page coloring and bin hopping using the same method, we find out that only in 1% of the experiments, bin hopping was outperformed by page coloring, but page coloring was outperformed by bin hopping in as much as 83% of experiments.

Figure 6 then shows the first best strategies with a minimum percentage difference required for means to be regarded different. If we only care about 1% or 3% differences, page coloring performs roughly the same as the default strategy for non-Csibe benchmarks, but is worse for Csibe benchmarks. There is no difference between page coloring and the default strategy if we only care about differences of 7% (non-Csibe) or 9% (Csibe).

In summary, both in non-Csibe and Csibe benchmarks, the best choice for mean execution time would be the default kernel strategy. If bin hopping or page coloring would have to be chosen, then page coloring would be better for non-Csibe benchmarks, but bin hopping would be better for Csibe benchmarks.

5.2 Non-determinism in Measurement

Non-determinism in measurement of non-Csibe benchmarks was evaluated using confidence intervals for coefficient of variation in measurements within a benchmark experiment execution. The default kernel strategy was outperformed in 29% of experiments. That is, in 29% of experiments, the default kernel strategy had higher non-determinism than some other strategy. Bin hopping was outperformed in 31% of experiments, and page coloring only in 15% of experiments. Thus, for 75% of experiments, page coloring lead to either the most predictable execution out of these strategies, or else it was no worse (in a statistical sense) than the best.

When compared only with bin hopping, page coloring was better in 44% of experiments and worse in only 2% of experiments. When compared only with the default kernel strategy, page coloring was better in 35% of experiments and worse in 13% of experiments. In summary, as far as non-determinism in measurement is concerned, page coloring would be the best choice.

5.3 Unclassified Non-determinism

The unclassified non-determinism can be measured both in Csibe and non-Csibe benchmarks. Again, we evaluated the results using bootstrap based confidence intervals for the coefficient of variation.

In the case of non-Csibe benchmarks, the default kernel strategy was outperformed in 19% of experiments. Bin hopping was outperformed in 21% of experiments, and page coloring was only outperformed in 6% of experiments. When compared only with bin hopping, page coloring was better in 19% of experiments and

[1]Note that the percentages are subject to rounding error, 0% does not necessarily mean that the situation did not happen.

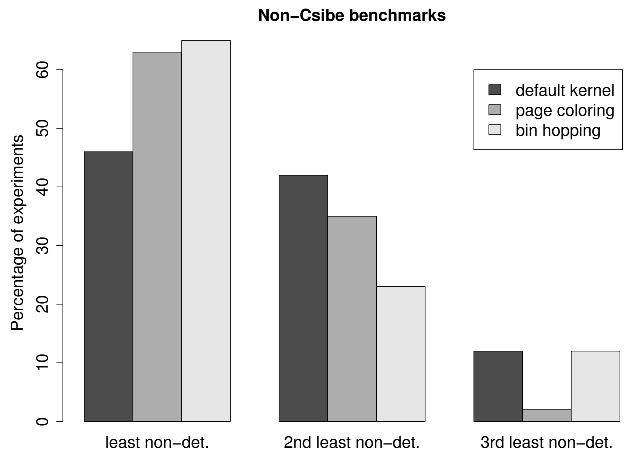

Figure 7: Performance non-determinism in execution of different page allocation strategies, measured by non-Csibe benchmarks. In 54% of the experiments, the default kernel strategy produced larger non-determinism than bin hopping or page coloring.

worse in 4% of experiments. When compared only with the default kernel strategy, page coloring was "never" (in rounded 0% of experiments) worse, but the default kernel strategy was worse in 25% of experiments. In the case of Csibe benchmarks, the default kernel strategy did not have significantly worse non-determinism than any other strategy in "any" experiment. However, the other strategies were also good, when compared only with the default kernel strategy, page coloring was "never" worse and bin hopping was only worse in 1% of experiments.

In summary, as far as (unclassified) non-determinism is concerned, page coloring seems to be the best choice: it is better than the default kernel strategy in non-Csibe benchmarks and it is about the same as the default kernel strategy in Csibe benchmarks.

5.4 Non-determinism in Execution

The non-determinism in execution can only be evaluated for non-Csibe benchmarks, which provide multiple measurements per execution. Non-determinism was measured using confidence interval for impact factor, as described in Section 4. The results are shown in Figure 7.

The Figure shows that in 54% of experiments, the default kernel strategy produced larger non-determinism in execution than one of page coloring and bin hopping. If we compare only page coloring and bin hopping, in one third of the experiments they produce similar non-determinism (not statistically significant differences in impact factor), in one third of the experiments page coloring is better than bin hopping, and in one third bin hopping is better than page coloring.

If we compare only page coloring with the default kernel strategy, page coloring is better than the default kernel strategy in 37% experiments, in 13% experiments is worse. If we only compare bin hopping with the default kernel strategy, bin hopping is better in 40% and worse in 25% of the experiments.

Figure 8 then shows the first best strategies with a minimum percentage difference required for impact factors to be regarded different. If we only care about 100% differences, page coloring is best in 79% of experiments, bin hopping in 88% and the default strategy in 75%. It is interesting that differences between strategies are still measurable if we only care about 800% differences in impact factors. This well emphasizes how important the strategies are for non-determinism. If we only care about 50% differences, the pair comparisons with default strategy change (this cannot be seen from the table). Page coloring is better in 23% and worse in 12% of experiments, while bin hopping is better in 33% and worse in 13% of experiments.

Strategy	% of Experiments for % Im.F. Diff				
	0	50	100	500	800
Page Coloring	63	71	79	94	96
Default	46	63	75	90	92
Bin Hopping	65	79	88	98	98

Figure 8: The strategies with least non-determinism in execution (non-Csibe). Page coloring is still better than the default strategy even if we only care about eight-fold (800%) differences between impact factors.

In summary, both bin hopping and page coloring provide lower non-determinism in execution than the default kernel strategy. Bin hopping seems to be slightly better.

5.5 Qualitative Improvements and Degradations

Out of all executed non-Csibe benchmarks, the highest statistically significant slowdown of page coloring was 41%, the highest speedup of page coloring was 49%. The highest slowdown of bin hopping was as much as 275% and the highest speedup was 49%. The Csibe benchmarks were much less influenced by page allocation strategies, for page coloring maximum speedup was 2% and maximum slow-down 6%. For bin hopping, maximum speedup was 4%, maximum slow-down 9%. On average, as calculated by a geometric mean from all benchmarks, the performance change was none (0%) for both page coloring and bin hopping on both Csibe and non-Csibe benchmarks.

Focusing on changes in the non-determinism in execution measured by impact factor, non-Csibe benchmarks with page coloring reported between a 54% reduction in non-determinism and a 477% increase, while with bin hopping they reported between a 61% reduction and a 62% increase. These numbers suggest that strategies should be selected on a per-application basis.

Figure 9 shows the magnitude of the reduction of non-determinism in execution in non-Csibe benchmarks. The plot shows an empirical cumulative distribution function of the impact factor. Especially for the range of impact factor of $\sqrt{10}$ to 100, both page coloring and bin hopping provide smaller non-determinism in execution than the default kernel strategy.

5.6 Qualitative Results in Real-Time Setting

The real-time results were more stable than the non-realtime ones. After rounding, no benchmark reported a significant mean performance change with page coloring, and no change to 1% slowdown with bin hopping (no change on average). Changes in non-determinism were between a 27% reduction and a 14% increase with page coloring, and between a 34% reduction and a 17% increase with bin hopping.

Out of the benchmarks used by others (Csibe, Scimark, and FFT), both Scimark and FFT were in some configurations very sensitive to cache effects, but the Csibe benchmarks were reasonably stable.

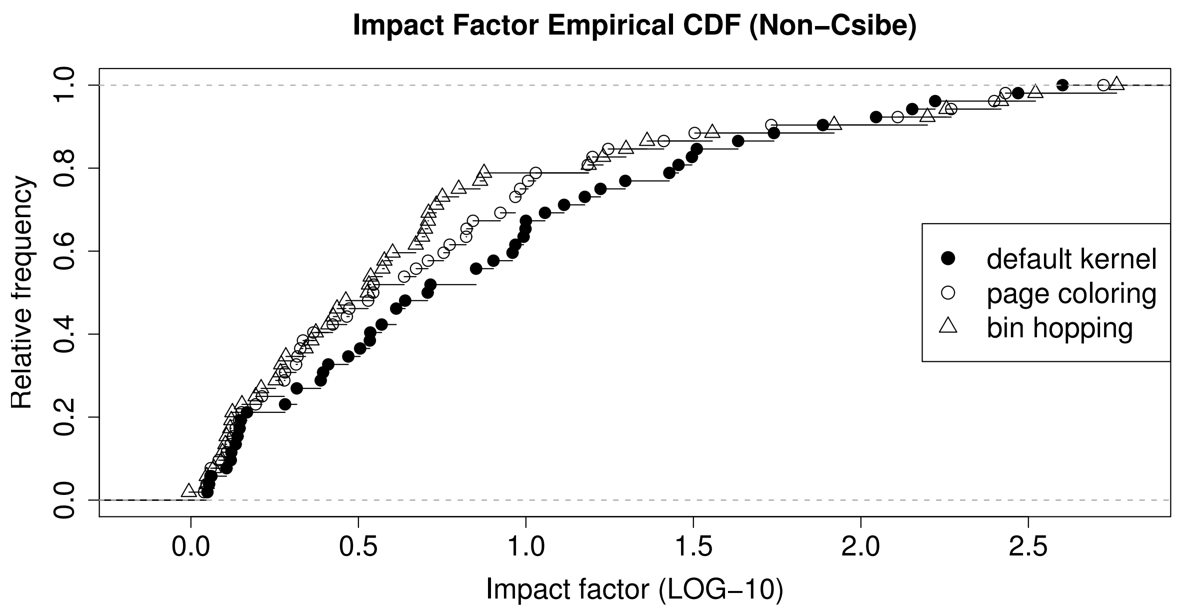

Figure 9: Empirical cumulative distribution function of impact factor (all non-Csibe experiments). Higher frequency is better, because it means smaller non-determinism in execution, as more experiments have ended up with impact factor of particular upper bound.

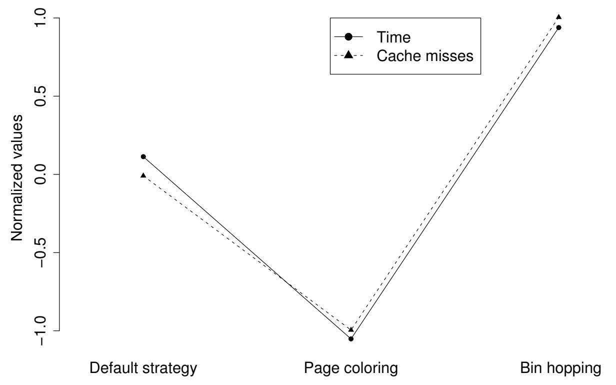

Figure 10: Response time vs. the number of cache misses for a concrete benchmark (FFT Scimark) and different page allocation strategies. The plot suggests that differences of measured time among strategies come from different numbers of cache misses.

6. CAUSE OF THE SLOW-DOWN

The results shown in the previous section might seem counter-intuitive at first sight. Although page coloring and bin-hopping incorporate some cache friendliness into the memory allocation, and reduce the performance non-determinism, at the same time they sometimes slow down the execution.

The reason why this can happen is that the reduction in non-determinism does not come from the strategies being cache friendly. It is simply a consequence of the strategies providing the application with memory pages colored deterministically depending on the application behavior.

The results therefore suggest that the default kernel strategy is not bad at reducing cache misses, which is also in line with results published in [30]. Both page coloring and bin hopping are not always good at reducing cache misses as they assume a concrete memory access pattern that is not common to all applications. Applications not following the assumed pattern would be slowed down, since they would suffer from more cache misses.

To verify that this explains slowdowns seen in our results, we verify that the slowdowns are really caused by sub-optimally colored pages, and not, for instance, by a run-time overhead in our implementation of page coloring or bin hopping. The purpose is only to check the cause of the slowdown, not to extensively evaluate all overheads. The expected overhead of having the strategies in the kernel would be in page allocation, which takes place largely during the benchmark warm-up period, and thus is not included into these measurements.

For this verification, we have extended the benchmarks to collect the numbers of memory cache misses via hardware performance counters. We further focused the evaluation on experiments that had statistically significant differences in measured times for all three strategies: the default kernel strategy, page coloring, and bin hopping. In these experiments, we looked for correlation between the response time and cache misses.

As there were only three strategies to compare, we checked the correlation visually. For each of 20 non-Csibe experiments matching the above criteria, we generated a plot with normalized time and normalized number of cache misses, like the one shown in Figure 10. In 18 of the 20 plots, the times and cache misses almost or completely overlapped, like in Figure 10, confirming that the slowdown was caused by the cache misses, and thus the page allocation strategies. For an analysis why the default Linux strategy is good at minimizing cache misses, although it is not by design cache-aware, we refer to [30].

7. CONCLUSION

Several operating systems use a cache-aware page allocation strategy, mostly page coloring, to reduce conflict misses in memory caches, thus improving mean performance. The strategies should also reduce performance non-determinism. Both benefits have however been disputed on current 8-way associative memory caches, and thus some systems do not implement a cache aware strategy at all (Linux), or do not support it in default kernel (FreeBSD). There was no in-depth experimental evaluation that would support the decision on mean performance grounds, and to our knowledge no evaluation whatsoever targeted at non-determinism.

We have implemented page coloring and bin hopping strategies in the Linux kernel and based on 4500 experiments statistically evaluated their impact on mean performance and performance non-determinism. Although in some experiments page coloring or bin hopping have improved mean performance, in majority of them the default Linux allocator was not outperformed.

We however found out that both page coloring and bin hopping are more efficient than the default Linux kernel allocator at reducing the non-determinism. Page coloring worked even better with a real-time kernel and real-time scheduling, where predictability is typically preferred over mean performance. Bin hopping caused some slowdowns on general benchmarks, but sometimes reduced the non-determinism better than page coloring. In summary, both page coloring and bin hopping have minimal mean performance overhead and do reduce non-determinism.

Choosing among the two might be well based on external decisions: for instance, in regression benchmarking, bin hopping might be preferred as it is bound to time aspect of application behavior (what happens next), which programmers have better control over than space aspect (memory layout), which is determined more by the compiler, so that even subtle changes such as renaming variables might impact performance. On the other hand, a deployed (real-time) system might benefit from page-coloring, which would reduce the non-determinism with slightly lower slowdowns compared to bin hopping.

We have indeed tried to characterize applications that would benefit most (least) from the individual page allocation strategies in terms of increased performance (reduced non-determinism). When seeking such characterization, we took into account characteristics such as application type (numerical application, compilation, remote procedure call), or environment (native, managed), problem size compared to cache size, the cache size itself, or the platform's number of page colors. Interestingly, our results suggest that such high level characterization cannot be made by fair means.

If optimization of a concrete (real-time) application is sought, the selection of the proper strategy can still be based on experiments with a concrete application on a concrete platform. Our kernel extension [18] is designed to allow this selection to be made automatically by the running application, allowing also "self-tuning" of applications.

Acknowledgments

The authors would like to thank Filip Pizlo and Jan Vitek for their careful reviews and insightful comments that have helped to improve the paper. This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

8. REFERENCES

[1] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 367–384, New York, NY, USA, 2008. ACM.

[2] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA*, pages 7–18, 2003.

[3] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance Workshop, OOPSLA 2004*, October 2004.

[4] Roldan Pozo and Bruce Miller. SciMark 2.0 benchmark. http://math.nist.gov/scimark2/, 2005.

[5] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. http://http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf, 2004.

[6] Roberto Bucher and Lorenzo Dozio. CACSD under RTAI Linux with RTAI-LAB. In *Realtime Linux Workshop*, 2003.

[7] David F. Bacon, Perry Chang, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. pages 285–298, New Orleans, Louisiana, January 2003.

[8] Distributed Systems Research Group. Mono regression benchmarking. http://dsrg.mff.cuni.cz/projects/mono, 2005.

[9] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *Proceedings of SPECTS 2005*, pages 853–862, San Diego, CA, USA, July 2005. SCS.

[10] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, 1992.

[11] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Sun Microsystems Press, Prentice Hall, Aug 2006.

[12] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[13] Matthew Dillon. Design elements of the FreeBSD VM system. http://www.freebsd.org/doc/en/articles/vm-design, 2001.

[14] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 244–255, New York, NY, USA, 1996. ACM Press.

[15] Linus Torvalds. Linux kernel mailing list: Page coloring. http://lkml.org/lkml/2003/12/27/81, Dec 2003.

[16] Richard Gooch. Linux kernel patches I've written, patches for the 2.2.x series kernel. http://www.atnf.csiro.au/people/rgooch/linux/kernel-patches.html, 1999.

[17] Jason Papadopoulos. Linux kernel mailing list: Rewritten page coloring for 2.4.20 kernel. http://lkml.org/lkml/2003/1/4/221, Jan 2003.

[18] Michal Hocko. Tuning virtual memory for performance. http://dsrg.mff.cuni.cz/projects/vmtune, 2007.

[19] Tomas Kalibera and Petr Tuma. Precise regression benchmarking with random effects: Improving Mono benchmark results. In Andras Horvath and Miklos Telek, editors, *Formal Methods and Stochastic Models for Performance Evaluation*, volume 4054 of *Lecture Notes in Computer Science*, pages 63–77. Springer, June 2006.

[20] Erik G. Hallnor and Steven K. Reinhardt. A fully associative software-managed cache design. *SIGARCH Comput. Archit. News*, 28(2):107–116, 2000.

[21] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[22] Michal Hocko. Tuning virtual memory for performance. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2007. http://dsrg.mff.cuni.cz/publications/Hocko-master.pdf.

[23] Stephane Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proceedings of the Linux Symposium*, pages 269–287, 2006. http://www.linuxsymposium.org/2006/proceedings.php.

[24] Stephane Eranian et. al. Perfmon2: the hardware-based performance monitoring interface for Linux. http://perfmon2.sourceforge.net/, 2007.

[25] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley–Interscience, New York, NY, USA, April 1991.

[26] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[27] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, New York, NY, USA, September 2004.

[28] Charles E. McCulloch and Shayle R. Searle. *Generalized, Linear and Mixed Models*. Wiley–Interscience, New York, NY, USA, 2001.

[29] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Automated detection of performance regressions: The Mono experience. In *MASCOTS*, pages 183–190. IEEE Computer Society, 2005.

[30] Satyendra Bahadur, Viswanathan Kalyanakrishnan, and James Westall. An empirical study of the effects of careful page placement in Linux. In *ACM-SE 36: Proceedings of the 36th annual Southeast regional conference*, pages 241–250, New York, NY, USA, 1998. ACM Press.

[31] Don Mayer and O. Buneman. FFT benchmark. ftp://ftp.nosc.mil/pub/aburto/fft.

[32] Novell, Inc. The Mono Project. http://www.mono-project.com, 2006.

[33] Chris Re and Werner Vogels. SciMark – C#. http://rotor.cs.cornell.edu/SciMark/, 2004.

[34] Árpád Beszédes, Rudolf Ferenc, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Csibe benchmark: One year perspective and plans. In *Proceedings of the 2004 GCC Developers' Summit*, pages 7–15, June 2004.

APPENDIX

A. USED BENCHMARKS

FFT.

The Fast Fourier Transformation (FFT) benchmark we use is a slightly transformed version of Don Mayer's benchmark [31] written in C. We have modified it to repeat the FFT transformations in each benchmark execution and report individual measurements.

Scimark2.

From the C version of Scimark2 [4], we have extracted multiple sub-benchmarks: Fast Fourier Transformation (FFT), Dense LU Matrix Factorization (LU), Monte Carlo Integration, Jacobi Successive Over-relaxation (SOR), and Sparse Matrix Multiplication. Our sub-benchmarks repeat the respective operation multiple times and report individual measurements.

Mono Benchmarks.

We have re-used benchmarks from the Mono Regression Benchmarking Suite [8]. The benchmarks are written in C# and executed in Mono [32], an open source implementation of the .NET platform. The individual benchmarks are Fast Fourier Transform (FFT), HTTP Ping, TCP Ping (remote communication), and Rijndael (cryptography).

The FFT benchmark is based on the C# version of the Scimark2 benchmark [33]. It is provided in two versions, one that allocates memory once for all repeated FFT calculations, and one that repeats the allocation in each calculation. We have created these two versions based on our experience with FFT benchmarks – they are very sensitive to memory management [9].

The HTTP Ping and TCP Ping benchmarks are client-server benchmarks that measure remote method invocation of a single method. The client and the server are run as two different processes. For remote communication, HTTP Ping uses SOAP, while TCP Ping benchmark uses plain TCP. Both protocols are supported by Mono class libraries.

The Rijndael benchmark measures encryption and decryption of a short text using the Rijndael algorithm. The algorithm itself is implemented in Mono libraries.

Csibe.

We use Code Size Benchmark Suite (Csibe) [34], a set of C/C++ based benchmarks of the GCC compiler. The benchmarks measure generated code size, compile time and sometimes execution time, providing single measurement per execution.

We have modified the Csibe execution scripts to use command line perfmon application [23], which can measure a given process using hardware performance counters. In compilation benchmarks, we have summarized the measured values for all processes (front-end, back-end, etc.) that contributed to each compilation.

The suite contains about 900 compilation benchmarks, out of which 30 produce benchmark code that can itself be executed. The executable benchmarks include various types of JPEG compression, lossless compression (GZIP, BZIP2, PNG), lexical analysis, binary to hexadecimal (ASCII) encoding, and an abstract machine simulator (VAM).