Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,4
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner

Thomas Karcher, Victor Pankratius

2011

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner

Thomas Karcher
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
thomas.karcher@kit.edu

Victor Pankratius
Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
victor.pankratius@kit.edu

ABSTRACT

In response to the growing demand for better performance, multicore platforms have become ubiquitous. A critical problem is that the diversity of hardware and software characteristics is increasing, making it a great challenge for parallel application developers to optimize performance and preserve application portability. In addition, unknown workload compositions and run-time interferences make the effects of tuning parameter settings hard to predict. We tackle this problem at its core and present Perpetuum, a novel operating-system-based auto-tuner that is capable of tuning applications cooperatively at run-time. Applications expose tuning parameters and feedback measurements of a repeatedly executed section to the OS. As part of the OS, Perpetuum monitors workloads and adapts tuning parameter values of all running programs to improve performance. In contrast to earlier approaches, this paper is the first to employ OS-based auto-tuning to improve system-wide performance for simultaneously executing multithreaded applications – not just the partial performance of an isolated application. The entire tuning process does not require any user involvement, and applications are automatically re-tuned while executing on new platforms. In addition, this is the first paper to work out the details and present a fully functional OS-integrated auto-tuner based on a modified Linux kernel. We also present successful evaluations on multicore platforms for different application types, such as multimedia and compression.

1. INTRODUCTION

Many software developers are confronted with the development of multithreaded applications in order to exploit the potential of multicore hardware. Performance tuning is especially hard in this context for several reasons:

- Multicore platform characteristics are different, so program optimizations working on one platform might not work on another one.
- The behavior of complex applications is difficult to predict.
- Applications may have many – possibly interdependent – tuning parameters that impact performance.
- The value ranges of tuning parameters and thus the entire search space can be large.
- More than one application can be executed simultaneously, which means that even if performance parameter values have been optimized for one application on a particular platform, the same parameter configuration might not have good performance if applied to several simultaneously executing instances.

Because it is difficult to find a general analytical solution, in practice these problems force programmers to manually try out application parameter configurations and check which ones do improve performance and which ones do not. Auto-tuning approaches have shown great potential to automate this process in a feedback loop. On the one hand, offline tuning has been developed to execute an application, gather run-time feedback, and use an algorithm to calculate new tunable parameter values that are likely to improve performance. However, most of the existing solutions have drawbacks. For example, [1, 2, 3] work just for domain-specific numerical programs (e.g. matrix-multiply or FFT); they generate a set of programs on every platform, out of which the best-performing one is picked. Unfortunately this principle works only for a limited set of application domains. Moreover, the tuning of an application in isolation does not reflect today's usage scenarios on multicore desktops and servers, which requires online tuning. Long-running parallel applications might have to be re-tuned while they are executing; the environment may change when other applications are started, or when the operating system allocates resources (e.g., threads, CPU, memory, etc.) in a different way.

To tackle this problem, our paper makes several novel contributions. This is the first paper to propose Perpetuum, an auto-tuner for parallel applications that is integrated into the operating system kernel. Its design gives us a unique opportunity to tune several applications simultaneously and hide the complexity of the tuning process from the user and the developer. Our auto-tuner is capable of optimizing the performance of shared-memory multithreaded applications while they are running, assuming that applications expose by default their performance-relevant tuning parameters and

Technical Report 2011-4
Karlsruhe Institute of Technology, Germany
Multicore Software Engineering Young Investigator Group
February 4, 2011

the associated value ranges to the OS. If every OS integrates an auto-tuner such as ours, then developers need to worry less about performance portability; every application would be re-tuned while it executes on another platform that has different hardware characteristics. We demonstrate in two extensive case studies that our tuning approach works well and is generally applicable beyond numerical programs. The first study focuses on a reengineering scenario for a parallel compression application, whereas the second study shows a parallel video processing application developed from scratch. Both studies illustrate significant performance improvements in single-process and multi-process execution scenarios. We remark that even though Perpetuum has been developed for shared-memory multicore machines, it could also be employed on cluster nodes to automatically improve single-node multithreaded performance.

The paper is organized as follows. Section 2 discusses principles and assumptions of our run-time auto-tuning approach. Section 3 introduces Perpetuum, our OS-based auto-tuner, and discusses technical details. Section 4 presents a detailed case study on parallel compression with several scenarios illustrating performance optimization in single-process and multi-process contexts. Section 5 elaborates on similar aspects for a parallel video-processing application. Section 6 discusses related work. Section 7 has an outlook on future extensions. Section 8 provides a conclusion.

2. ONLINE TUNING

We now discuss some requirements for the structure of online-tunable applications and illustrate how the optimization cycle generally works.

2.1 Structure of Tunable Applications

We assume that every application tuned at run-time has one compute-intensive "hot-spot", i.e., a modular part of code that is executed over and over again. The programmer is responsible for developing his or her application with such a hot-spot or identify one in existing code. The programmer has to define tuning parameters that influence application performance (but not the results) within a hot-spot. To close the auto-tuning feedback loop, the programmer inserts measurement probes that determine the execution time of the parameterized hot-spot. We also assume that applications have a longer run-time so that the auto-tuner gets a chance to execute several iterations, adapt parameter values, and observe the effects.

As an example, consider the following C code.

```
int threadCount = 1;
addParam(threadCount, 1, 16);
while (calculationRunning) {
  startMeasurement();
  doCalculation(threadCount);
  stopMeasurement();
}
```

The hot-spot in the above example consists of a loop doing some calculation. The tuning parameter `threadCount` has been defined by the developer to control the level of parallelism. It is well conceivable to have additional parameters that steer `doCalculation()`. The `addParam()` function registers the thread count variable and its ranges at the auto-tuner. In future iterations, the auto-tuner will set the variable's values to a number between 1 to 16. Note that it is the responsibility of the programmer that such changes produce consistent results; we'll show in our case studies that it is not very difficult to do in practice. Finally, `startMeasurement()` and `stopMeasurement()` are the probes telling the auto-tuner where to obtain feedback information; they can be configured to use various metrics, e.g., wall-clock time. Using this information, the auto-tuner can evaluate the impact of thread count changes on performance.

2.2 The Optimization Cycle

Figure 1 illustrates the general optimization cycle. The tunable application is started in the execution phase and the auto-tuner gathers all tuning parameters. Performance optimization occurs in a loop. After executing one iteration of the application's tuning hot-spot, the auto-tuner collects feedback information. Based on this information (e.g., elapsed execution time) it calculates for all tuning parameters new values that are expected to improve performance. The tuning parameters are assigned these values before the next iteration begins. The optimization cycle repeats until the application terminates.

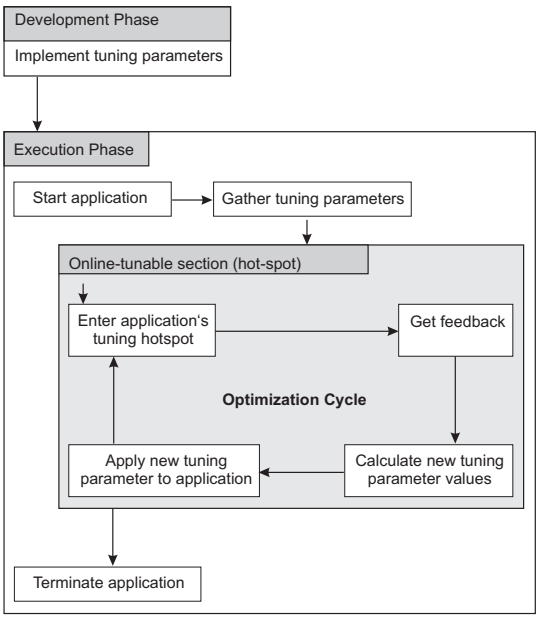

Figure 1: The optimization cycle for online tuning.

Note that the auto-tuner algorithm can adjust the tuning parameters according to the overall system workload. When two applications compete for example for cache or memory I/O, the auto-tuner aims for a cross-process optimum, as defined by the objective function of the tuning algorithm. If one application terminates and releases its resources, another application can be assigned additional resources that become available.

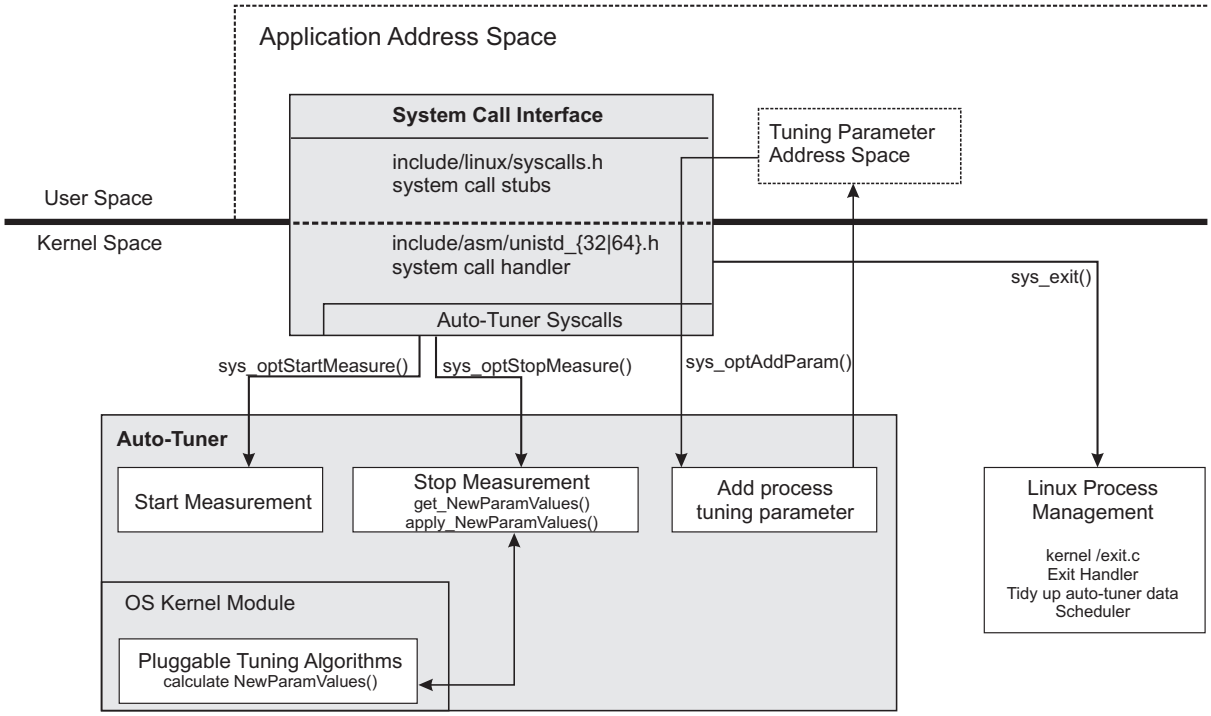

Figure 2: Overview of Perpetuum's system architecture.

3. PERPETUUM: A COOPERATIVE ONLINE TUNER

We introduce Perpetuum's details on how it's integrated into the Linux kernel, the feedback measurement mechanisms, and the tuning algorithm used to optimize application performance.

3.1 Architecture Overview

Figure 2 shows the overall system architecture of Perpetuum and how it is integrated into the Linux OS kernel. All tunable applications run in the user space. An exclusive part of each tunable application's address space is reserved for the tuning parameter address space; this is used by Perpetuum to store, read, and modify the values of the tuning parameters associated with an application.

A tunable application communicates with the auto-tuner, which is inside the OS kernel, via the system call interface. Perpetuum introduces new system calls that on the one hand allow an application to expose its tuning parameters to the OS, and on the other hand communicate the feedback information gathered from program-internal probes.

Why use system calls as a means for communication with the kernel? In Linux, there are only few methods of communication between the kernel and an application: Hardware interrupts, traps (i. e. software interrupts), and system calls. The first two are inappropriate because they are typically triggered by a hardware component or indicating a system error, so it's unreliable to use them for other types of communication. System calls are better suited because they do not interfere with other kernel communication.

The auto-tuner is an independent component in the Linux kernel that reacts to three new systems calls received via system call interface. Calling *sys_optAddParam()* registers a new tunable parameter, *sys_optStartMeasure()* may start a clock counter to measure execution time, and *sys_optStopMeasure()* may stop the clock counter. Values in the tuning parameter address space can only be changed during the *sys_optStopMeasure()* call; this call blocks the calling thread until all parameters are updated.

Tuning algorithms can be integrated into Perpetuum as OS kernel modules. It is possible to extend the system with additional plugins that implement new tuning algorithms. The tuning algorithms access within the kernel all application's tuning parameter values and feedback information to compute the new parameter values for the next iteration. In addition, they may access OS-internal data about workloads and system state – Perpetuum is the first auto-tuner to provide the means to use such information for tuning. A tuning algorithm finally updates the tuning parameter values for the next iteration directly in the tuning parameter address space of each application.

We remark that Perpetuum does not make any modifications to the Linux OS scheduler, which is part of the Linux process management module. This design decision is based on the fact that Perpetuum influences application tuning knobs that are on a higher abstraction level [4]. By contrast, the scheduler influences low-level resource management decisions such as on which core to execute a particular thread. Perpetuum currently influences the scheduler just indirectly; applications acting upon a parameter change initiated by Perpetuum may increase or reduce the number of threads managed by the scheduler.

3.2 Feedback Measurement Mechanisms

Operating systems provide access to various kinds of data that could be used for performance tuning, e.g.,:

- **Wall-clock time:** This is the time passing independently of any activity. In practice, it's the time a user spends in front of the computer while waiting for a task to finish.
- **CPU time (ticks):** This metric counts for an application how much time has been actually spent using the CPU and can differ from wall-clock time. For example, CPU time can be less than wall-clock time if an application has to wait for I/O.
- **Hardware performance counters:** Modern CPUs are capable of counting certain hardware events, such as cache misses or how many cycles the CPU spent in a specific state.

Perpetuum uses wall-clock time as the default feedback metric. We consider it the most appropriate for run-time optimizations in practical scenarios, because this is the amount of time the user has to wait for an application to finish. Calls to `startMeasurement()` and `stopMeasurement()` notify the auto-tuner to collect feedback information.

3.3 Tuning Algorithm

Perpetuum's tuning algorithm is based on the popular simplex heuristic developed by Nelder and Mead [5]. Given an n-dimensional search space, the basic idea is to generate a simplex with $n+1$ points. In our case, n is the number of application parameters to optimize. A concrete value configuration from the search space, i.e., a concrete starting value for each parameter, is needed to initialize the simplex. The simplex consists of the point of the starting configuration and n more points that are generated by adding a constant displacement to the initial values in each dimension.

The points of this simplex are iteratively moved around to explore the search space for better configurations. Nelder and Mead define specific rules and transformations (see [5]); for example points can be "reflected", moved to expand or contract the simplex, or the entire simplex may be scaled down to promising areas.

Our implementation has to take care of several technical details. For example, our search space consists of discrete integers, so we have to compute the closest feasible points to the algorithm solutions. No floating point operations are allowed within the Linux kernel, so all numbers have to be mapped on integers.

4. CASE STUDIES ON PARALLEL COMPRESSION

Application Description:

The Bzip2 program is a popular sequential file compression tool for everyday use [6]. The compression algorithm divides a file stream into independent blocks that pass a pipeline of algorithms. At the end of the pipeline, compressed blocks are concatenated in the right order to a compressed file.

We start with a parallel Bzip2 version presented in [7], which has two command line parameters: The first parameter specifies the number of threads t that are used for parallel compression, and the second parameter specifies the block size b. Thread numbers can be chosen from an interval between 3 and 64 (a reasonable upper limit chosen for the experiment), whereas blocks can have sizes of $i * 100$ kilobytes with $i \in \{1, 2, \ldots, 9\}$. The parallel Bzip2 application has an option to compress all files in a given directory, so it is a potentially long-running application. If it executes for example on a server, it is realistic that several instances work simultaneously to process different directories.

Application Reengineering for Online-Tuning:

This case study exemplifies how to modify an existing parallel application to enable it for online tuning. Conceptually, the hot-spot is the compression code that executes for each file. It is located in the `handle_compress()` function in the `bzlib.c` file that encapsulates the core algorithm implementation. We added in this file two feedback information system calls to the auto-tuner that measure the wall-clock time of the hot-spot; the rest of the program has very minimal performance impact. The parameters that influence compression performance are t and b, and are defined as variables in the `bzip2.c` file. We added in that file two system calls to inform Perpetuum that those variables are tunable. In total, we added less than 10 lines of code.

If the application processes a directory of files, the hot-spot is called over and over again until all files are compressed. The reengineered parallel implementation can change upon request the values for t and b after finishing the compression of the current file.

Experimental Environment:

We conduct the experiments with the parallel Bzip2 application on an Intel Core 2 Quad Q6600 machine, clocked at 2.40GHz. The machine runs our customized Linux kernel version 2.6.34 with Perpetuum included.

Our experiments are carried out in a controlled environment; we deactivated the graphical user interface and any other interfering applications. We use for all scenarios the same collection of 50 files, each having a file size of 2MB, so the reengineered Bzip2 will execute the hot-spot 50 times. Having all files the same size is not a constraint of the auto-tuner. This setup makes results comparable exposes sources of bias.

Scenario 1: Tuning a Single Process:

This scenario evaluates that Perpetuum successfully tunes a single application while that application is running. Perpetuum controls parallel Bzip2's t and b parameters in order to reduce the run-time of the hot-spot.

First of all, the search space of this scenario is small enough, so we could exhaustively benchmark all parameter configurations for a single Bzip2 process without auto-tuning. In total there are $9 \times 61 = 558$ configurations. The execution time for each configuration was measured 3 times to avoid bias.

Our exploration indicates that the block size b has a significant performance impact. In addition, we made the following observations. For any given block size, $t < 5$ may slow down performance. The execution times tend to be better than the median execution time if $b < 5$ and $t \geq 5$. If $b \in \{1, 2\}$ and $t \geq 5$, the execution time is within the best 20%. We thus expect Perpetuum to reduce the block size

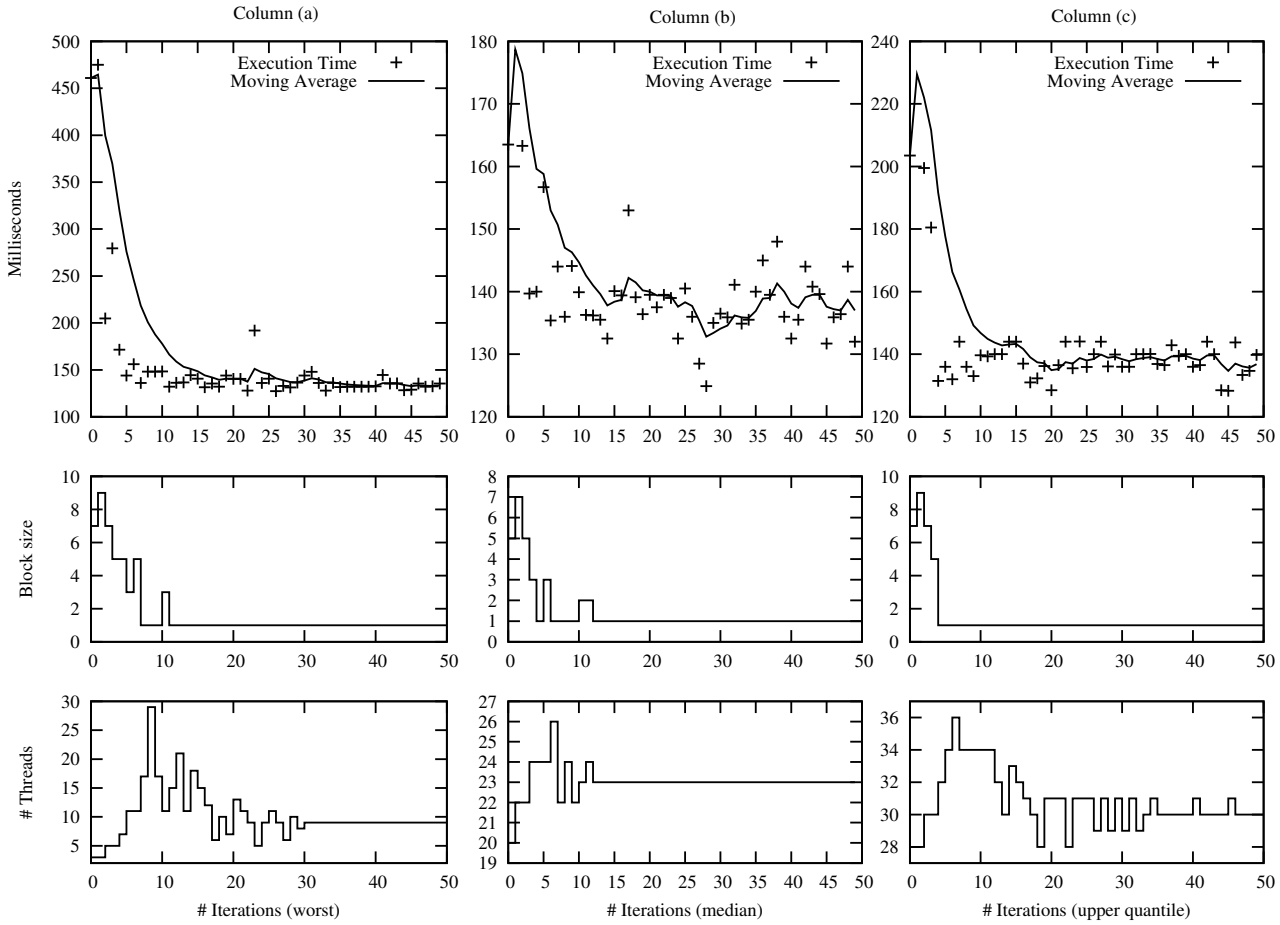

Figure 3: Online tuning of parallel Bzip2, scenario 1. Each column shows the results of one experiment with a single Bzip process. The first row illustrates hot-spot execution times, the other rows the tunable parameter values.

(ideally to $b = 1$) and increase thread count to $t \geq 5$.

With the best configuration, the entire program executes in 6.5 seconds, whereas the worst configuration takes 22.9 seconds. This is the range in which Perpetuum can optimize the application's execution time.

Three experiments. As Perpetuum needs a starting configuration, we conduct three experiments with configurations belonging to the median, upper quartile, and worst case execution times (known by exhaustive search). The results are illustrated in Figure 3. The Figure shows the execution times of the hot-spot (which accounts for almost the entire program execution time), the block size, and thread count as they change among iterations. For visualization, we also plot an exponential moving average of execution times using

$$a_0 = x_0, \quad a_i = 0.75a_{i-1} + (1 - 0.75)x_i$$

where a_i is the moving average value and x_i the execution time measured at the end of iteration i. Older values are included with exponentially lower weight into the current average value.

(1) Worst case performance: In Figure 3 column (a), the tuner starts with the configuration $b = 7$ and $t = 3$, which has an execution time of $458ms$. Without tuning, the application would have taken $458ms \times 50 = 22.9$ seconds to finish. Perpetuum reduces the average execution time to a total of 8 seconds, which is 2.9x faster. Note that this is not the classical speedup measure in comparison to the sequential program, but a performance boost in comparison to the parallel program; speedup compared to the sequential time of 24.5 seconds is even higher, namely 3.1. The final tuning result is just 23% worse than the best attainable execution time.

The tuning parameter graphs in the second and third row illustrate how Perpetuum works. Both values increase at first, but while the auto-tuner considered the direction of change to be good for the thread count, it realizes that a smaller block size is better to reduce execution time. The block size quickly converges to 1, while other thread counts are tried out. The step for t is doubled until iteration #8, and t finally converges to 9 threads after some oscillation.

Our exhaustive measurements exploring the search space show that a configuration with a $b = 1$ and $t = 9$ is within the best 1% of all configurations.

(2) Median performance: The tuner starts in Figure 3 column (b) with the configuration $b = 5$ and $t = 20$. Without auto-tuning, finishing all iterations would have taken 8.6 sec-

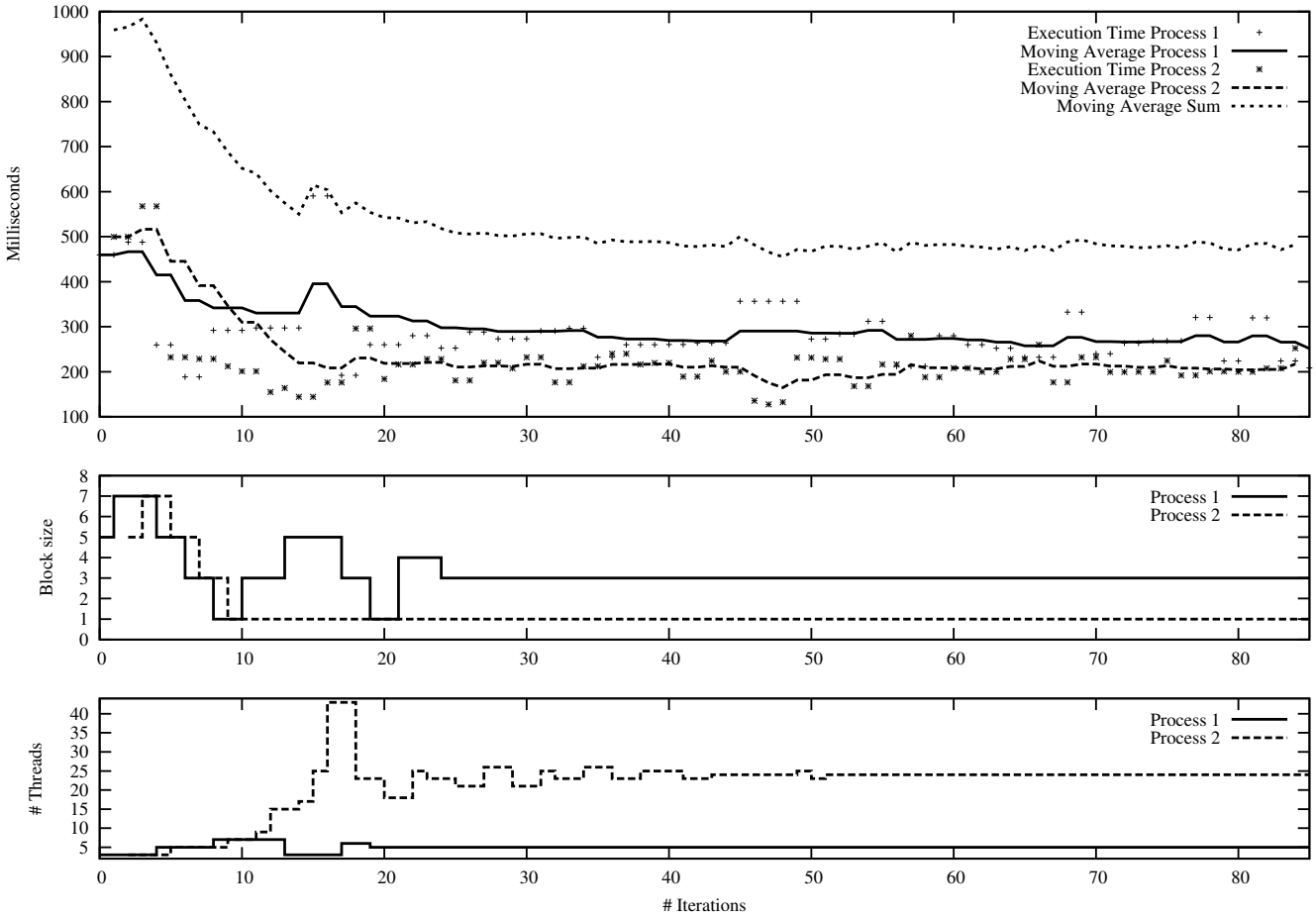

Figure 4: Online tuning of parallel Bzip2, scenario 2. Two instances of Bzip2 are started at the same time and tuned simultaneously.

onds, whereas the auto-tuned version finishes in 7.2 seconds – this is still 1.2x faster.

(3) Upper quartile performance: The experiment in Figure 3 column (c) has a starting configuration between the median and worst case performance, with $b = 7$ and $t = 28$. Without auto-tuning, finishing all iterations would have taken 10.4 seconds, whereas the auto-tuned version finishes in 7.4 seconds, so the online-tuned version is 1.4x faster.

If the starting configuration has already good values, Perpetuum tries to tune the application but is not able to significantly improve performance, so we omit these graphs.

Scenario 2: Simultaneously Auto-tuning Two Processes:

In this scenario we evaluate how Perpetuum simultaneously tunes two processes that are started at the same time.

The experiment executes two instances of the parallel Bzip2 application that work on individual copies of the file benchmark used in scenario 1. Each instance starts with the same configuration $b = 5$ and $t = 3$ which is within the worst 10% of execution times. The graphs also show a higher execution time variance with two processes, which is due to increased system activity on CPU, RAM, and hard disk.

Without auto-tuning, starting both instances at the same time and waiting for the last one to finish takes 26.5 seconds. With auto-tuning, it takes just 13.5 seconds. This boosts performance by a factor of 1.96.

Fig. 4 shows the execution time results for both Bzip processes and how Perpetuum adapted block size and thread count for each process. The auto-tuner reduces the block size for both processes. Process 2 reaches $b = 1$, which was the optimum in scenario 1. Process 1 also is assigned $b = 1$ for some iterations, but the auto-tuner finds out that it can reduce execution time by increasing block size to 3, which differs from the single-process scenario. The sum of the moving averages of the two processes decreases, which shows that Perpetuum indeed improves overall system performance.

Perpetuum automatically finds the critical point around $t = 5$ after 10 iterations, which we manually identified ourselves in the exhaustive exploration of the search space in the single-process scenario (the single process was significantly slower when $t \leq 4$). As a result, Perpetuum increases the thread counts for both processes. Process 1 converges to $t = 5$ and process 2 to $t = 24$.

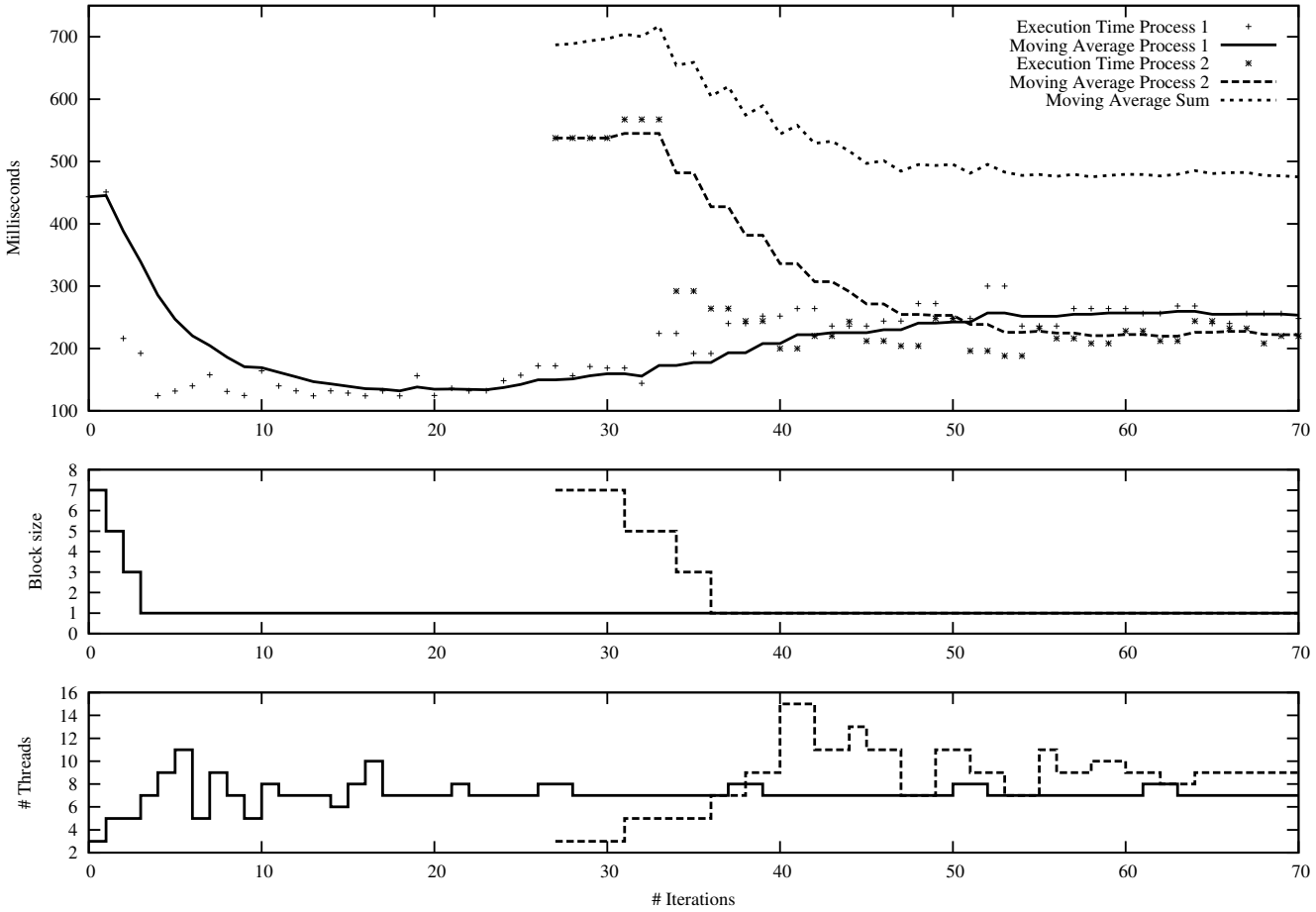

Figure 5: Online tuning of parallel Bzip2, scenario 3. Two instances of Bzip2 are started with a time lag and tuned simultaneously.

Scenario 3: Simultaneously Auto-Tuning Two Processes with a Time Lag:

This scenario is similar to scenario 2, except that the second process is started 4 seconds after the first one. Figure 5 shows the timeline: The first process starts off solo, as in scenario 1. The block size converges again quickly to $b = 1$ while the thread count roughly converges to $t = 7$. Then, the second process starts. While the tuning parameters of process 2 are modified as expected, the execution time of process 1 increases due to interference with process 2. The auto-tuner does not change the block size in both processes, but assigns process 2 more threads, possibly to hide latency. Apparently, this strategy improves overall performance, because the moving average sum decreases.

Lessons Learned from Scenarios 1–3:

The auto-tuner improves performance significantly in multiple application scenarios. Parameters such as the block size, which had more impact on performance variance, were adjusted earlier than the others. We remark that a heuristic used by many programmers is to set the number of threads to the number of cores. However, in all our scenarios, configurations having 4 threads and any block size were in the worst 10% performance interval. Perpetuum could not be fooled into this false assumption and quickly converged to better values within the first 10 iterations.

5. CASE STUDIES ON PARALLEL VIDEO PROCESSING

Application Description:

This is an online-tunable application that is designed from scratch [8]. It applies a parallel edge detection algorithm to an input video stream, and produces an output video stream of images that just show the edges of objects. This is an important application in computer vision, as edge detection is the basis for other algorithms that are used for example to identify or track objects. A fast online-tunable version can be employed in many areas, such as robotics, security, or human-computer interaction.

In principle, the application has five multithreaded filters in a pipeline. Each filter is a pipeline stage that works in parallel on one frame of the video stream:

- Stage 1 (Gauss): Performs a Gaussian blur by applying a convolution mask.
- Stage 2 (Gradient): Applies a Sobel mask to compute the gradient strength and direction for each pixel.

- Stage 3 (Trace): Traces the edges based on the gradients computed in the previous stage.
- Stage 4 (Suppress): Suppresses pixels that haven't been identified to be on an edge.
- Stage 5 (Non-Max): Performs some clean-ups in the picture by eliminating weaker edges that are parallel to stronger ones.

The application uses Intel's Threading Building Blocks [9] and assigns a tunable number of threads to each pipeline stage. For each stage, thread count can be set from 1 to 64. The tunable hot-spot measures the execution time of every 10 frames passing the entire pipeline.

Experimental Environment:

The experiments are conducted on the same machine as in the Bzip2 case study. As an input data set, we use the first 720 frames of an open source movie [10]. Out input file has an AVI format with MPEG-4 compression, 854x480 pixels resolution, 24 frames per second, and a total size of roughly 12 MB.

Scenario 1: Tuning a Single Process:

The total search space with our five parameters consists of $64^5 = 1,073,741,824$ configurations, which can hardly be explored exhaustively. Instead, we run a few explorative tests. It appears that the first two stages have more impact on the overall application run-time than the last three stages. We thus focus on exploring the first two stages with thread counts between 1 and 16 for each stage. All measurements are repeated 3 times. The best-performing configuration have 11 threads for Gauss and 5 threads for Gradient, with a total run-time of 116.3 seconds. However, the characteristics of the distribution are interesting: The difference between the 100 fastest configurations is at most 4.3 seconds; the curve seems rather flat for at least 100 additional configurations before it increases. We also find that intuitive configurations such as assigning 1 thread per each stage end up within the worst 10% performance. Configurations with thread count of 1 for the Gauss stage are in the worst 5% of all configurations. The worst configuration has threads assigned to stages as follows: 1-16-1-1-1. The average run-time without auto-tuner is 384.4 seconds. The Gauss thread count parameter seems to have high sensitivity; when it increases from 1 to 2, performance makes a surge. Performance improvements are smaller if more than 2 threads are assigned to this stage.

Our results with the auto-tuner are illustrated in Figure 6, which exemplifies how well Perpetuum performs on the worst case in which the auto-tuner gets the start configuration 1-16-1-1-1 with bad performance. In the first iteration, hot-spot execution time is about 5.2 seconds, but the auto-tuner is able to finally reduce it to 2.7 seconds, which is a 1.9x improvement. It is also remarkable that the auto-tuner tries out tuning the thread count of the last stages but quickly realizes that they don't have much effect, so the values remain constant. By contrast, the thread counts in the first stages are tuned more often, and the tuner automatically detected what we had to find out manually: Increasing thread count of Gauss above 1 significantly improves performance.

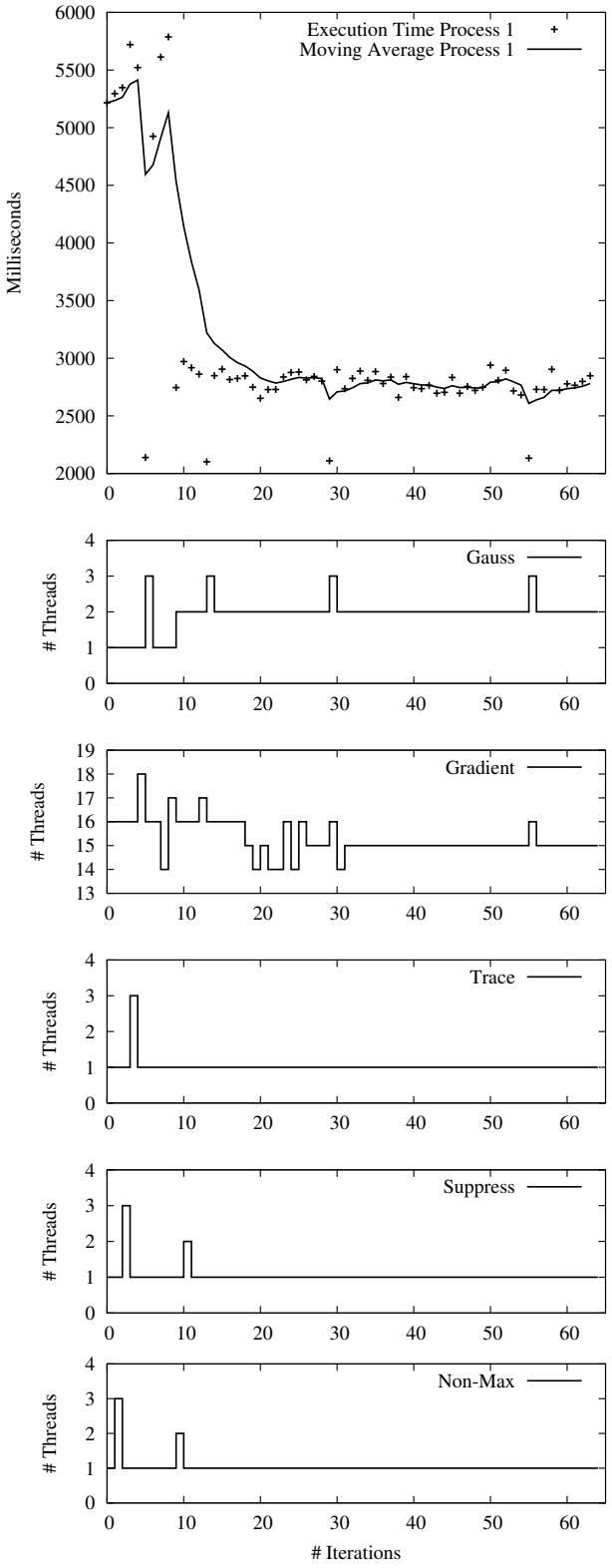

Figure 6: Online tuning of a single-process video processing application, scenario 1. The graph illustrates hot-spot execution times, the other graphs show the tunable parameter values.

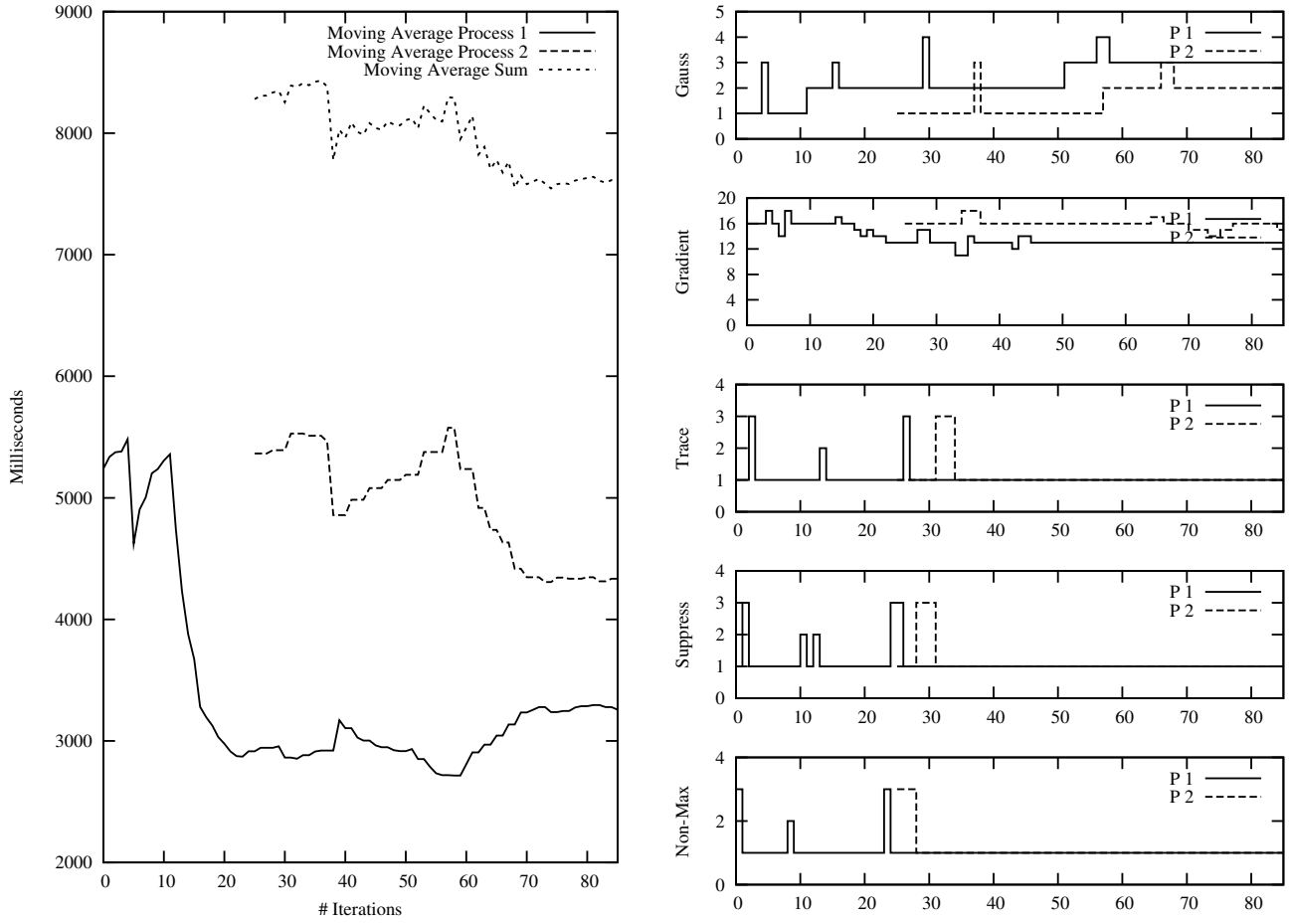

Figure 7: Online tuning of a video processing application, scenario 2. Two processes are started with a time lag and tuned simultaneously.

The final configuration that Perpetuum converges to is non-intuitive. The number of threads per stage is distributed as follows: 2-15-1-1-1, so a total of 20 threads works best on our 4-core machine!

Scenario 2: Simultaneously Tuning Two Processes with a Time Lag:

In a similar way to the Bzip2 online tuning case study we start two processes of the video processing application with the configuration 1-16-1-1-1. Figure 7 shows that this tuning scenario is more difficult. The first process is tuned in a similar way to scenario 1; if the Gauss filter has more than one thread, performance improves significantly. At iteration 25, the second process starts and interferes in the environment, which causes some disturbances in both graphs. The performance of process two finally improves after the autotuner has found that increasing the thread count of Gauss is good. Note that even though process one's run-time increases until it terminates in iteration 88, the overall system performance represented by the moving average sum still improves. In addition, the hot-spot execution time is improved for each process until the last iteration, in comparison to the first iteration.

6. RELATED WORK

Cooperative auto-tuning with operating system support has been welcomed as a new idea in the community [11]. Most of the related work, however, covers online tuning with a different focus and with other techniques.

Orio [12] focuses mostly low-level performance optimizations on a particular code fragment annotated with specific structured comments. It generates many tuned versions of the same operation and empirically evaluates the alternatives to select the best performing version for production use.

MATE [13] provides dynamic tuning for MPI applications and is designed for distributed architectures. The tuning decisions are made according to performance models and predefined tuning parameters.

An adaptive task scheduler for multitasked data-parallel jobs is introduced in [14] for distributed systems. The scheduler has a continuous feedback loop that lets the scheduler know about new processor requests.

The work of [15] uses hardware performance counters in a multicore-aware operating system to gather fine-grained information about run-time application behavior. This information is needed in a feedback loop that performs dynamic program optimizations. The system aims to minimize cache

contention by clustering threads and assign dedicated cache regions to threads.

The *Contention Aware Execution Runtime (CAER) environment* [16] provides a run-time solution that minimizes cross-core interference due to contention, while maximizing utilization. CAER leverages performance monitoring features of multicore processors to infer and respond to contention.

Active Harmony [17, 18, 19, 20] is a suite of compiler and run-time tools to support parameter tuning of parallel programs in heterogeneous distributed environments. We measured the overhead of this approach on multicore to be significantly higher than in Perpetuum. Moreover, our approach favors low response times over throughout, which makes it more suitable for multicore desktops.

Scenario Based Optimization [21] combines static and dynamic optimization techniques for online tuning. In a static phase, the system generates multiple versions of some programming language function. Each version's performance is evaluated at run-time, and for future evaluations the control flow is directed to the best-performing version.

In the compiler domain, [22] present a compiler framework that can detect at run-time which code optimizations to apply. The work of [23] employs machine learning to iteratively learn about program features and adapt compiler optimizations accordingly.

7. OUTLOOK

The OS-based auto-tuner developed in this paper provides a platform for novel research in auto-tuning to make the optimization process transparent for the user. Moreover, tuning algorithm can efficiently access a variety of OS-internal information about global system state.

A promising extension is to add semantics to the application parameters that are exposed to the OS. If the auto-tuner knows that a certain parameter represents for example a number of threads or buffer sizes, it can make more sophisticated tuning decisions.

We now have a working prototype, and future versions of Perpetuum can provide more advanced means to gather other run-time feedback information to detect congestions, bandwidth overload, etc. This will help better identify and factor out influences on application run-time and make tuning decisions that lead to faster convergence to an optimum run-time. Online tuning algorithms such as ours based on Nelder-Mead need to be improved to escape local minima.

The combination of online tuning and offline tuning has great potential. Perpetuum needs a starting configuration for tuning, and it would be advantageous to store information from earlier program runs, so when the same application is started again, online tuning may draw upon insights from historical executions.

8. CONCLUSION

Perpetuum is the first cooperative online auto-tuner that is integrated into the Linux OS kernel. To our knowledge, this is the first paper to present a working prototype of such a system. Our case studies show that the performance of multicore applications can be significantly improved while they are running. The entire optimization process is transparent for users and developers, thus relieving them of heavy burdens of manual performance experiments and performance portability issues. The platform we developed offers a very fertile new ground for future research because performance optimization can access global system state that is only visible within the OS. Our long-term vision is that every multicore application will be auto-tuned by default.

Acknowledgments. We thank the Excellence Initiative and the Landesstiftung Baden-Wuerttemberg for financial support. We also thank Sascha Schwedes for his initial prototype implementation of the kernel auto-tuner and Iskandar Abudiab for the implementation of the video processing application.

9. REFERENCES

[1] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Proc. IEEE ICASSP'98*, vol. 3, 1998, pp. 1381–1384.

[2] C. R. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, January 2001.

[3] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "Spiral: code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[4] S. Schwedes, "Operating system integration of an automatic performance optimizer for parallel applications," Master's thesis, Karlsruhe Institute of Technology, 2009.

[5] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, 1965.

[6] J. Seward, "bzip2," http://www.bzip.org/, October 1 2010. [Online]. Available: http://www.bzip.org/

[7] V. Pankratius, A. Jannesari, and W. Tichy, "Parallelizing bzip2: A case study in multicore software engineering," *Software, IEEE*, vol. 26, no. 6, pp. 70 –77, nov.-dec. 2009.

[8] I. Abudiab, "Online-tunable parallel edge detection in video streams," Student project thesis. Karlsruhe Institute of Technology, 2010.

[9] Intel, "Threading building blocks," August 2006. [Online]. Available: http://www.threadingbuildingblocks.org/

[10] S. Goedegebure, A. Goralczyk, E. Valenza, N. Vegdahl, W. Reynish, B. van Lommel, C. Barton, J. Morgenstern, and T. Roosendaal, "Big buck bunny. an open source movie." April 2008, last accessed October 1, 2010. [Online]. Available: http://www.bigbuckbunny.org/

[11] T. Karcher, C. Schaefer, and V. Pankratius, "Auto-tuning support for manycore applications: perspectives for operating systems and compilers," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 96–97, April 2009.

[12] A. Hartono and S. Ponnuswamy, "Annotation-based empirical performance tuning using Orio," in *23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS) Rome, Italy*, May 2009.

[13] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, "Mate: Monitoring, analysis and tuning environment for parallel & distributed applications: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 11, pp. 1517–1531, 2007.

[14] K. Agrawal, Y. He, W. J. Hsu, and C. E. Leiserson, "Adaptive scheduling with parallelism feedback," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 100–109.

[15] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 56–65, 2009.

[16] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "Contention aware execution: online contention detection and response," in *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2010, pp. 257–265.

[17] V. Tabatabaee and J. K. Hollingsworth, "Automatic software interference detection in parallel applications," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[18] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth, "Parallel parameter tuning for applications with performance variability," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 57.

[19] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: towards automated performance tuning," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11.

[20] A. Tiwari, V. Tabatabaee, and J. K. Hollingsworth, "Tuning parallel applications in parallel," *Parallel Comput.*, vol. 35, no. 8-9, pp. 475–492, 2009.

[21] J. Mars and R. Hundt, "Scenario based optimization: A framework for statically enabling online optimizations," in *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 169–179.

[22] J. Cavazos, J. E. B. Moss, and M. F. P. OâĂŹBoyle, "Hybrid optimizations: Which optimization algorithm to use?" in *5th International Conference On Compiler Construction (CC 2006)*, 2006.

[23] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 295–305.