Run-Time Automatic Performance Tuning for Multicore Applications

Thomas Karcher and Victor Pankratius

Karlsruhe Institute of Technology, IPD
76128 Karlsruhe, Germany
{thomas.karcher,victor.pankratius}@kit.edu

Abstract. Multicore hardware and system software have become complex and differ from platform to platform. Parallel application performance optimization and portability are now a real challenge. In practice, the effects of tuning parameters are hard to predict. Programmers face even more difficulties when several applications run in parallel and influence each other indirectly. We tackle these problems with Perpetuum, a novel operating-system-based auto-tuner that is capable of tuning applications while they are running. We go beyond tuning one application in isolation and are the first to employ OS-based auto-tuning to improve system-wide application performance. Our fully functional auto-tuner extends the Linux kernel, and the application tuning process does not require any user involvement. General multicore applications are automatically re-tuned on new platforms while they are executing, which makes portability easy. Extensive case studies with real applications demonstrate the feasibility and efficiency of our approach. Perpetuum realizes a first milestone in our vision to make every performance-critical multicore application auto-tuned by default.

1 Introduction

Software developers need to create parallel applications to exploit the multicore hardware potential. Intuitive performance tuning, however, has become difficult for several reasons: (1) Different multicore platform characteristics may cause application optimizations to work on one platform, but not on others. (2) The behavior of complex parallel applications is hard to predict. (3) Applications may have several tuning parameters that impact performance. Potential parameter interdependencies can be difficult to understand. (4) The typical search space spanned by tuning parameters is large. (5) Good performance configurations that work for each application in isolation might not work when several applications run simultaneously on the same system.

In practice, programmers resort to manual and often unsystematic experiments to find program parameter configurations that lead to good performance. Auto-tuning has shown great potential to automate this process and make the search more intelligent using a feedback loop. Offline tuning approaches execute an application until it terminates, gather run-time feedback, and calculate new

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6852, Part I, pp. 3–14, 2011.

tuning parameter values that are likely to improve performance in the next run. Most of the existing solutions, however, have drawbacks. For example, [7,18,24] target domain-specific numerical programs (e.g. matrix multiply or Fourier transform). They generate on every platform a new set of executable programs and pick the best-performing one. Unfortunately, this principle does not work for general parallel programs that do not perform any of these numerical analyses. Another issue is that isolated application tuning is inappropriate in today's scenarios. A typical multicore system environment changes all the time due to dynamic resource allocation and applications that run in parallel. This requires long-running applications to be tuned at run-time.

Our paper makes several novel contributions to tackle the aforementioned problems. We introduce Perpetuum, the first auto-tuner for shared-memory multicore applications that integrates into the Linux operating system. Perpetuum's design offers unique opportunities to tune several applications simultaneously and hide the complexity of the tuning process from users and developers. Perpetuum optimizes the performance of applications while they are running, assuming that applications expose their performance-relevant tuning parameters and the associated value ranges to the operating system. Perpetuum's OS-integration reduces tuning overhead and eases portability; an application ported to a new computer that runs Perpetuum will be automatically re-tuned. Moreover, our approach is applicable beyond numerical scientific programs. Two extensive case studies demonstrate feasibility. We achieve respectable performance improvements for compression and multimedia applications in single-process and multi-process scenarios.

The paper is organized as follows. Section 2 introduces the Perpetuum auto-tuner. Section 3 discusses how to prepare applications for online tuning. Section 4 presents case studies with a reengineered parallel compression application. Several scenarios demonstrate Perpetuum's effectiveness in single-process and multi-process contexts. Section 5 presents evaluation studies for an on-line tunable parallel video-processing application written from scratch. Section 6 discusses related work. Section 7 provides a conclusion.

2 The Perpetuum Run-Time Application Tuner

Figure 1 shows the overall system architecture of Perpetuum and how it is integrated into Linux. All tunable applications run in user space. An exclusive part of each tunable application's address space is reserved for a dedicated tuning parameter address space; this space is used by Perpetuum to store, read, and modify the values of all tuning parameters associated with an application.

The auto-tuner is an independent component within the Linux kernel. A tunable application communicates with the auto-tuner using the system call interface. There are three new system calls: (1) The `sys_optAddParam()` call registers a new tunable parameter; (2) `sys_optStartMeasure()` starts a wall clock time counter to measure execution time; (3) `sys_optStopMeasure()` stops the clock counter. Values in the tuning parameter address space can only be changed

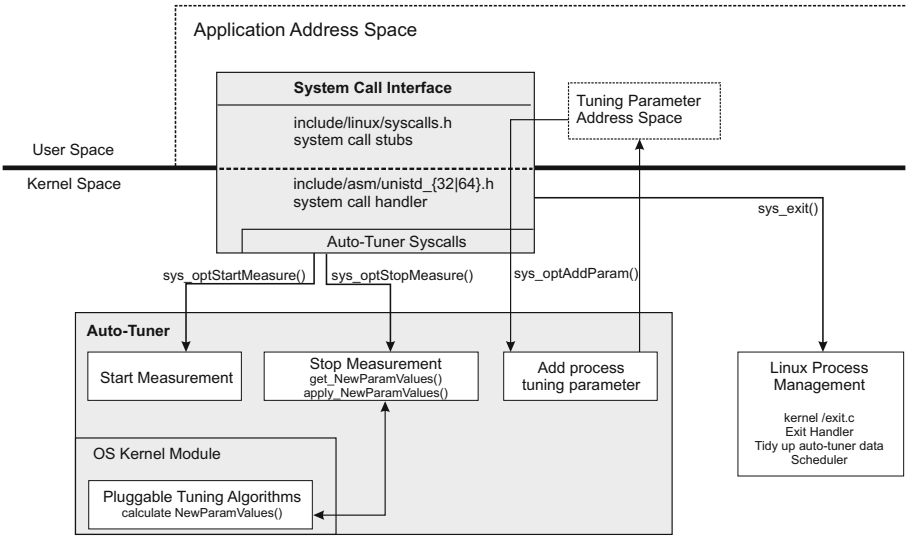

Fig. 1. Overview of Perpetuum's system architecture

during the `sys_optStopMeasure()` call, which blocks the calling thread until all parameters are updated.

Perpetuum uses one system-wide, application-independent tuning algorithm. However, this algorithm can be easily exchanged by system administrators (e.g., a simplex-based algorithm [16] can be replaced by another optimization algorithm). The algorithm is implemented in a plugin style as a Linux kernel module. In contrast to other auto-tuners (see Section 6), our architecture allows plugins to access operating system data, e.g., on workloads and system state. The tuning algorithm is called in a loop by every executing program. The tuning parameters of each application are updated (in its tuning parameter address space) with values that the tuner considers promising for the next iteration.

Perpetuum's current tuning algorithm is based on an adapted version of [16] that works on a discrete integer space and in a scenario with multiple applications that are tuned simultaneously. The number n of application parameters to optimize spans our n-dimensional search space. We generate a simplex with $n+1$ points. Our simplex consists of a starting configuration point and n more points that are obtained by adding a constant displacement in each dimension. When searching for better configurations, we move simplex points based on application execution time feedback and the rules defined in [16].

Perpetuum does not make any modifications to the Linux 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 [19]. By contrast, the scheduler influences low-level resource management decisions, e.g., on which core to execute a certain thread. Perpetuum influences, however, the scheduler in an indirect way: Applications

reacting to a change caused by Perpetuum may increase or reduce the number of threads that the scheduler controls.

We remark that even though Perpetuum has been developed for shared-memory multicore machines, it could also be run on every node of a cluster to automatically improve single-node multithreaded performance of work assigned to the node. Fine-granular performance tuning in clusters thus becomes easier as well.

3 Preparing Applications for Online Tuning

We assume that every application to be tuned at run-time has one compute-intensive "hot-spot", i.e., a modular part of code that is executed in a repetitive manner. Applications should have a longer run-time so that the auto-tuner gets a chance to execute several iterations, adapt parameter values, and observe the effects. The programmer is responsible for developing an application with such a hot-spot or identify one in existing code. To establish an auto-tuning feedback loop, the programmer inserts measurement probes that determine the execution time of the parameterized hot-spot, as shown in the C code example below:

```
int threadCount = 1;
addParam(&threadCount, 1, 16); //tunable degree of parallelism
while (calculationRunning) {
  startMeasurement();          //tuning feedback probe
  doCalculation(threadCount);  //hot-spot
  stopMeasurement();           //tuning feedback probe
}
```

The auto-tuner will automatically set `threadCount`'s values to a number between 1 and 16. It is the responsibility of the programmer that such changes produce consistent results. Our case studies further illustrate in more complex examples that the adaptation of applications for online tuning is not difficult to do.

After each iteration of the application's tuning hot-spot, the auto-tuner collects feedback information. Based on the elapsed execution time, it calculates new values for all tuning parameters before the next iteration begins. The optimization cycle repeats until the application terminates.

Note that the auto-tuner algorithm can adjust the tuning parameters according to the overall system workload that indirectly influences the run-time of the hot-spot. When two applications compete for example for cache or memory I/O, the auto-tuner aims for a cross-process optimum, which is obtained based on the objective function of the tuning algorithm. If an application terminates and releases its resources, another application can be assigned the newly available resources. We now show in two case studies how Perpetuum adapts application parameters in an automated fashion.

4 Perpetuum in Action: Automated Online-Tuning in Parallel Compression

This case study exemplifies how to reengineer an existing parallel application and make it tunable at run-time. We illustrate Perpetuum's online tuning in three scenarios.

4.1 Environment

The sequential Bzip2 file compressor divides a file stream into independent blocks and passes them through a pipeline of algorithms [20]. At the end of the pipeline, compressed blocks are concatenated in their original order and stored in an output file.

We employ the parallel Bzip2 version of [17] that has two command line parameters: the number of compression threads t and the block size b in hundred kilobytes. In our scenarios, we use $t \in \{3, 4, \ldots, 64\}$ and $b \in \{1, 2, \ldots, 9\}$. The tuning hot-spot is the compression code that is applied to each file, located in Bzip's `handle_compress()` function. We reengineered the application and added two system calls to measure the wall-clock time of the hot-spot. In addition, we added two system calls to make t and b tunable. When a directory of files is compressed, the hot-spot is executed in a repetitive fashion. Our implementation can update t and b with new values after finishing the compression of the current file.

We conducted the experiments in a controlled environment on the following machine: Intel Core 2 Quad Q6600 machine, 2.40GHz, running Linux 2.6.34 with Perpetuum. We deactivated the graphical user interface and all other interfering applications. All scenarios use the same collection of 50 files (each with a size of 2 MB), so the compression hot-spot executes 50 times. The fact that all files have the same size is not a constraint of the auto-tuner; this setup was chosen to make results comparable and identify sources of bias more easily.

4.2 Scenario 1: Tuning a Single Process

This scenario shows that Perpetuum successfully tunes one application while that application is running. Perpetuum controls the t and b parameters and aims to reduce the run-time of the hot-spot.

To evaluate tuning effectiveness, we exhaustively benchmarked all parameter configurations for a single Bzip2 process without auto-tuning, for the total of $9 \times 61 = 558$ configurations. The execution time for each configuration was measured 3 times to avoid bias. These results allow us to compare Perpetuum's results with the real optimum.

The exhaustive measurements show that 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 (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 be expected to optimize the application's run time.

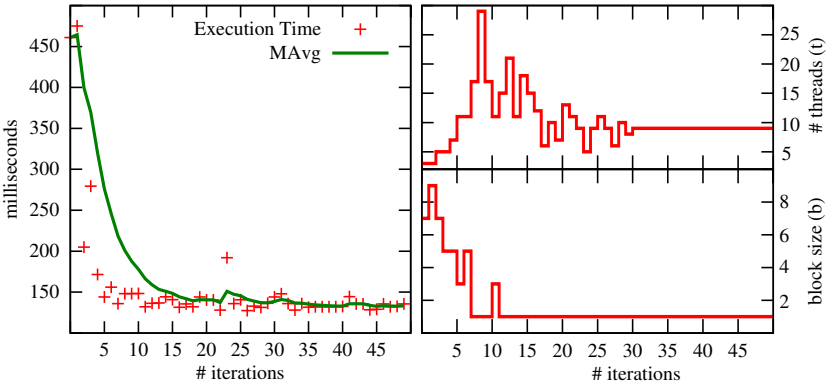

Fig. 2. Online tuning of parallel Bzip2. Left graph: hot-spot execution times after each iteration. Right graphs: values of the tunable parameters.

Figure 2 shows the execution time of the hot-spot (which accounts for almost the entire program execution time) for the block size and thread count chosen by Perpetuum in each iteration. We also plot an exponential moving average (MAvg) of execution times using $a_0 = x_0$ and $a_i = 0.75a_{i-1} + (1-0.75)x_i$, where a_i is the moving average value and x_i the execution time measured at the end of iteration i. Optimization starts at the worst-case configuration $b = 7$ and $t = 3$ where compression needs $458ms$. Without tuning, the entire 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, which is not bad considering the quadcore machine and almost no programming effort. The final tuning result is just 23% worse than the best attainable execution time.

The graphs for thread count t and block size b illustrate how Perpetuum works. Both values increase at first. The auto-tuner then realizes that increasing thread count alone is not too effective and that a smaller block size reduces execution times more significantly. 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 the finally obtained configuration of $b = 1$ and $t = 9$ is within the best 1% of all performance configurations.

We remark that a starting configuration can be randomly generated. If optimization had started, for example, with another configuration (e.g., $b = 5, t = 20$), finishing all iterations without tuning would have taken 8.6 seconds, and with tuning 7.2 seconds (which is still 1.2x faster). In general, if a starting configuration already has good values, Perpetuum tries to tune the application but will not be able to significantly improve performance, so it will stop tuning after some time.

4.3 Scenario 2: Simultaneously Auto-tuning Two Processes

This scenario evaluates how Perpetuum simultaneously tunes two processes that are started at the same time (see Figure 3 (a)). We execute two instances of the parallel Bzip2 application that work on individual copies of the file benchmark from 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 execution time variance with two processes is higher than in a single-process scenario, due to increased CPU, RAM, and hard disk activity.

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 of the parallel application by a factor of 1.96. Fig. 3 shows how Perpetuum adapted block size and thread count for each process. First, the auto-tuner reduces the block size for both processes. Process 2 reaches $b = 1$, which was the optimum in scenario 1. Process 1 is also assigned $b = 1$ for a few 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 moving averages (MAvg Sum) of the two processes decreases, which shows that Perpetuum globally improves 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 $t = 24$.

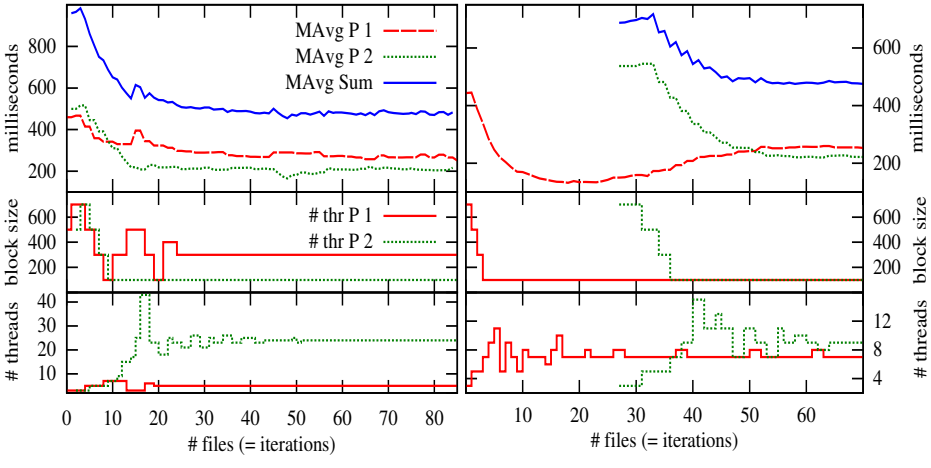

Fig. 3. Online tuning of parallel Bzip2. Graphs in first row show hot-spot moving average execution times; graphs in other rows show tuning parameter values. Scenario 2 (left): two instances are started at the same time and tuned simultaneously. Scenario 3 (right): two instances are started with a time lag.

4.4 Scenario 3: Simultaneously Auto-tuning Two Processes Starting 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 3 (b) shows the timeline: The first process starts off solo, as in scenario 1. The block size converges again 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, which likely has the effect of hiding latency. This strategy improves overall performance, as demonstrated by the decreasing moving average sum of execution times.

4.5 Summary

The auto-tuner significantly improves performance in all of our scenarios. Perpetuum adjusts parameters that have more impact on performance variance (e.g., block size) earlier than others. Another insight is that the common programmer intuition to set the number of threads to the number of cores would fail here: Configurations with 4 threads and an arbitrary block size had a performance within the worst 10% of all configurations. Perpetuum could not be fooled into this false assumption and quickly converged to better values within the first 10 iterations.

5 Automated Online-Tuning in Parallel Video Processing

This Section presents a study with an online-tunable multimedia application written from scratch [1]. The application performs parallel edge detection on a video stream. The output video stream consists of images that show the edges of objects. In computer vision, edge detection is an important basis for other algorithms, e.g., to track or identify objects in robotics, security applications, or human-computer interaction.

The application has five multithreaded filters organized in a pipeline. Each filter works in parallel within a pipeline stage on one frame of the video stream. **Stage S1 (Gauss)** performs a Gaussian blur by applying a convolution mask. **Stage S2 (Gradient)** applies a Sobel mask to compute the gradient strength and direction for each pixel. **Stage S3 (Trace)** traces the edges based on the gradients computed in the previous stage. **Stage S4 (Suppress)** suppresses pixels that are not on an edge. **Stage S5 (Non-Max)** performs some clean-ups in the picture by eliminating weaker edges that are parallel to stronger ones.

Parallelism is introduced using Intel's Threading Building Blocks [10] and assigning 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. The experiments are conducted on the same machine as in the Bzip2 case study. As an input data

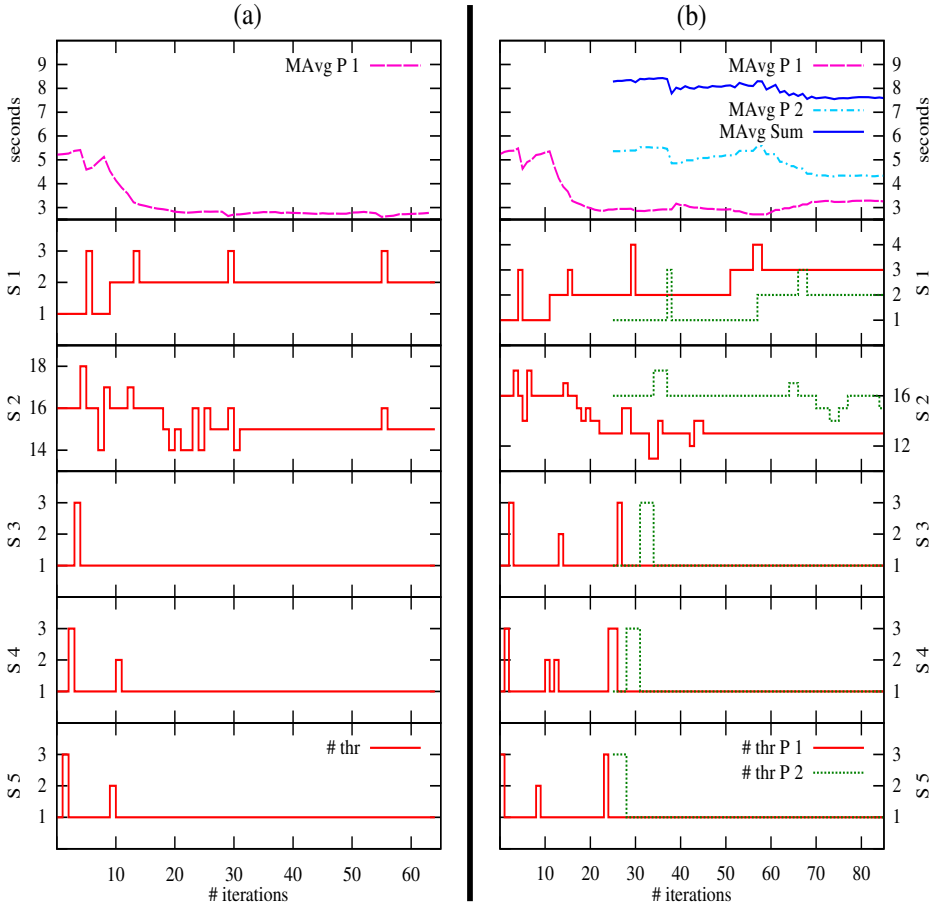

Fig. 4. Online tuning of a video processing application. The graphs in the first row illustrate hot-spot execution times, the others the tunable parameter values. (a): single-process scenario 1; (b): two-process scenario 2.

set, we use the first 720 frames of an open source movie [8]. Our 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.

5.1 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 benchmarked exhaustively, so we did explorative studies. The first two stages have more impact on the overall application run-time than the last three stages. We thus focused 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 has 11 threads for Gauss

(S1) and 5 threads for Gradient (S2), with a total run-time of 116.3 seconds. Intuitive configurations assigning 1 thread per stage end up within the worst 10% performance. Configurations with one thread for S1 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. S1's thread count parameter has high sensitivity; increasing it from 1 to 2 causes performance to surge, but performance improvements diminish if more than 2 threads are assigned to this stage.

Figure 4 (a) exemplifies how Perpetuum performs in the worst case with start configuration 1-16-1-1-1. 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 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 detects that increasing thread count for S1 above 1 significantly improves performance.

Perpetuum finally converges to the non-intuitive configuration of 2-15-1-1-1, showing that this application needs a total of 20 threads on our 4-core machine. This empirical result proves programmers wrong who assume that the number of threads must equal the number of cores.

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

Similarly to the Bzip2 online tuning case study, we start two processes of the video processing application, both with the configuration 1-16-1-1-1. Figure 4 (b) shows that this tuning scenario is more difficult. The first process is tuned similarly to scenario 1; if S1 receives more than one thread, performance improves significantly. At iteration 25, the second process starts interfering. The performance of process two finally improves after the auto-tuner finds that increasing S1's thread count 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. Finally, each application's hot-spot execution time is lower than before tuning.

6 Related Work

The advantages of integrating auto-tuners into operating systems haven been acknowledged by the operating systems community [11]. Other details on experiments with Perpetuum are summarized in a technical report [12]. Most of the related work covers online tuning with a different focus and with other techniques. *Orio* [9] focuses mostly low-level operation performance optimizations on a particular code fragment annotated with specific structured comments. *MATE* [15] provides dynamic tuning for MPI applications and is designed for distributed architectures. An adaptive task scheduler for multi-tasked data-parallel jobs is introduced in [3], however, assuming job granularity and a distributed system

environment. The work of [4] uses hardware performance counters and aims to minimize cache contention by clustering threads and assign dedicated cache regions to threads. The *CAER* environment [14] provides a run-time solution that targets a reduction of cross-core interference due to contention. *Active Harmony* [6,21,22,23] tunes one parallel program at a time in a heterogeneous, distributed environment. Each application has to obey a dedicated API to send performance feedback to a dedicated optimization server and receive new configurations via message passing. By contrast, Perpetuum is targeted at interactive use on shared-memory multicore desktops and server machines, so we have other assumptions about application characteristics and the acceptable computation and communication overhead. For example, Perpetuum is 15 seconds (30%) faster than our adapted version of *Active Harmony* running compression scenario one on our hardware. Several other approaches work on a lower abstraction levels than Perpetuum. The work of [13] combines static and dynamic binary compiler optimizations to select the best-performing variant of a program function out of multiple versions. A compiler framework that detects at run-time which code optimizations to apply is shown in [5]. Machine learning is applied in [2] to iteratively learn about program features and adapt compiler optimizations.

7 Conclusion

Perpetuum's infrastructure presented in this paper is the first OS-based approach to allow automatic performance tuning at runtime for simultaneously executing multicore applications. Our approach works well beyond scientific numerical programs and can be used in standard desktop PCs and servers. We are also the first to integrate such an auto-tuner into the Linux operating system, which has several advantages: (1) The performance optimization algorithm can access system information to compute the global performance optimum for all active applications in a cooperative way; (2) OS integration allows fast response times for online tuning; (3) Auto-tuning as a standard service in the OS allows programmers to outsource tuning logic from their code and make their code base easier to maintain; (4) Tedious and intuitive manual tuning (which might not even find optimum performance) becomes obsolete; (5) Parallel application portability is improved, as applications are automatically re-tuned on each platform. Overall, Perpetuum paves the way towards making auto-tuning a standard approach in multicore application development.

Acknowledgements. We thank the Excellence Inititative and the Landesstiftung Baden-Württemberg for their support.

References

1. Abudiab, I.: Online-tunable parallel edge detection in video streams. Student project thesis. Karlsruhe Institute of Technology (2010)
2. Agakov, F., et al.: Using machine learning to focus iterative optimization. In: CGO 2006, p. 11 (2006)

3. Agrawal, K., et al.: Adaptive scheduling with parallelism feedback. In: PPoPP 2006, p. 1 (2006)
4. Azimi, R., et al.: Enhancing operating system support for multicore processors by using hardware performance monitoring. SIGOPS Oper. Syst. Rev. 43(2), 56 (2009)
5. Cavazos, J., Moss, J.E.B., O'Boyle, M.: Hybrid optimizations: Which optimization algorithm to use? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 124–138. Springer, Heidelberg (2006)
6. Ţăpuş, C., et al.: Active harmony: towards automated performance tuning. In: SC 2002, p. 44 (2002)
7. Frigo, M., Johnson, S.: FFTW: an adaptive software architecture for the FFT. In: Proc. IEEE ICASSP 1998, vol. 3, p. 1381 (1998)
8. Goedegebure, S., et al.: Big buck bunny. An open source movie (April 2008), `http://www.bigbuckbunny.org` (last accessed May 2011)
9. Hartono, A., Ponnuswamy, S.: Annotation-based empirical performance tuning using Orio. In: IPDPS 2009, p. 1 (2009)
10. Intel: Threading building blocks (August 2006), `http://www.threadingbuildingblocks.org`
11. Karcher, T., et al.: Auto-tuning support for manycore applications: perspectives for operating systems and compilers. SIGOPS Oper. Syst. Rev. 43(2), 96 (2009); Special Iss. on the Interaction among the OS, Compilers, and Multicore Processors
12. Karcher, T., Pankratius, V.: Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner. Karlsruhe Reports in Informatics 2011-4 (February 2011)
13. Mars, J., Hundt, R.: Scenario based optimization: A framework for statically enabling online optimizations. In: Proc. CGO 2009, p. 169 (2009)
14. Mars, J., et al.: Contention aware execution: online contention detection and response. In: Proc. CGO 2010, p. 257 (2010)
15. Morajko, A., et al.: Mate: Monitoring, analysis and tuning environment for parallel & distributed applications: Research articles. Concurr. Comput.: Pract. Exper. 19(11), 1517 (2007)
16. Nelder, J.A., Mead, R.: A simplex method for function minimization. The Computer Journal 7(4), 308 (1965)
17. Pankratius, V., et al.: Parallelizing bzip2: A case study in multicore software engineering. IEEE Software 26(6), 70 (2009)
18. Puschel, M., et al.: Spiral: code generation for dsp transforms. Proceedings of the IEEE 93(2), 232 (2005)
19. Schwedes, S.: Operating system integration of an automatic performance optimizer for parallel applications. Master's thesis, Karlsruhe Institute of Technology (2009)
20. Seward, J.: Bzip2 (2011), `http://www.bzip.org`
21. Tabatabaee, V., Hollingsworth, J.K.: Automatic software interference detection in parallel applications. In: SC 2007, vol. 1, p. 14 (2007)
22. Tabatabaee, V., et al.: Parallel parameter tuning for applications with performance variability. In: SC 2005, p. 57 (2005)
23. Tiwari, A., et al.: Tuning parallel applications in parallel. Parallel Comput. 35(8-9), 475 (2009)
24. Whaley, C.R., et al.: Automated empirical optimizations of software and the atlas project. Parallel Computing 27(1-2), 3 (2001)