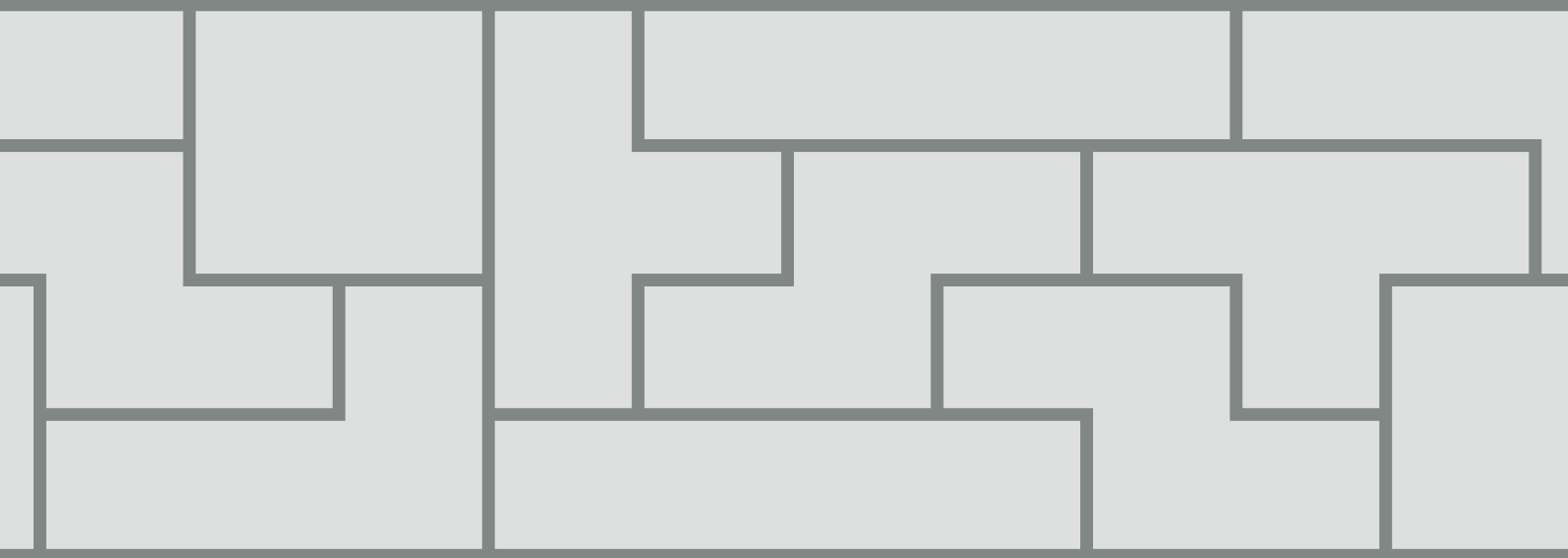
MICHAEL ROITZSCH

PRACTICAL REAL-TIME WITH LOOK-AHEAD SCHEDULING



I LOVE DEADLINES.

I LOVE THE WHOOSHING NOISE THEY MAKE AS THEY GO BY.

DOUGLAS ADAMS

THE ART OF PROPHECY IS VERY DIFFICULT,

ESPECIALLY WITH RESPECT TO THE FUTURE.

MARK TWAIN

OH DEAR! OH DEAR! I SHALL BE TOO LATE!

THE WHITE RABBIT

MICHAEL ROITZSCH

PRACTICAL REAL-TIME WITH LOOK-AHEAD SCHEDULING

DISSERTATION
ADVISOR: PROF. DR. RER. NAT. HERMANN HÄRTIG
TECHNISCHE UNIVERSITÄT DRESDEN

Dissertation zur Erlangung des akademischen Grades DOKTORINGENIEUR (DR.-ING.), vorgelegt an der technischen universität dresden, fakultät informatik, eingereicht von
DIPL.-INF. MICHAEL ROITZSCH, geboren am 15. August 1980 in Dresden.

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig, Technische Universität Dresden
Zweitgutachter:	Prof. Dr. Gerhard Fohler, Technische Universität Kaiserslautern
Fachreferent:	Prof. Dr. Christof Fetzer, Technische Universität Dresden
Statusvortrag:	12. Dezember 2011
Abgabe:	5. Juli 2013
Verteidigung:	19. September 2013

Dresden, 1. Oktober 2013

Acknowledgments

First I want to thank my advisor Prof. Hermann Härtig for his support of my work and for the inspiring research environment he creates in the Operating Systems Group at Technische Universität Dresden. Many members of the group contributed to this work. Especially, I thank Björn Döbel for his insightful ideas and helpful comments, and I thank Carsten Weinhold for teaming up on the bumpy ride toward finishing our theses. This research also greatly benefits from the master's thesis of Stefan Wächtler, whom I had the pleasure of working with. Several people proofread this text and gave valuable feedback, for which I am grateful: Prof. Gerhard Fohler, Prof. Hermann Härtig, Dr. Claude-Joachim Hamann, Björn Döbel, Stefan Wächtler, Simon Peter, and Ulf Lorenz. Finally, I want to thank my parents for always believing in me and supporting me in all my crazy endeavors.

Contents

Introduction

The evolution of computing has treated users with an impressive stream of innovation: From the mainframe era through the age of productivity computing to today's multimedia and mobile world, the capabilities of systems and thus the possibilities for users increased steadily. This sea change thrives on the exponential improvement of the underlying transistor technology as predicted by Moore's Law: The number of transistors that can be integrated cost efficiently doubles approximately every two years.[1] Software matches this exponential growth: The total body of open source software in the world doubles about every 14 months.[2]

[1] This period is often misquoted as 18 months, which is a follow-up prediction of the increase in single chip performance. (cf. *CNET News: Moore's Law to roll on for another decade*)

[2] Amit Deshpande, Dirk Riehle: *The Total Growth of Open Source.* Proceedings of the 4th Conference on Open Source Systems (OSS), pp. 197–209. Springer, September 2008

Application Necessities

Users have grown accustomed to the constant improvement of technology. What started as high-end and expert use cases will become a commodity just a few years later and will be expected to work predictably and efficiently. The previous generation iPad 2 would have been in 1994's Top 500 list of the fastest computers in the world[3] and it is arguably more intelligible today than those supercomputers were back then.

[3] John Markoff: *The iPad in Your Hand: As Fast as a Supercomputer of Yore.* Bits Blog. The New York Times, May 2011. From `bits.blogs.nytimes.com` as of December 2011

Living up to these expectations is a demanding job for developers. Users are no longer satisfied with functionality alone. Increasingly, non-functional properties separate good from great applications. Next to user interface design and visual appearance such properties include responsive and stutter-free operation,[4] perceived performance, and the useful and efficient employment of the invested resources,[5] also driven by the resource and energy constraints of today's battery-powered devices.

[4] Jason Olson: *Keeping Apps Fast and Fluid with Asynchrony in the Windows Runtime.* Windows 8 App Developer Blog. Microsoft, March 2012. From `blogs.msdn.com` as of May 2012

[5] Sharif Farag, Ben Srour: *Improving Power Efficiency for Applications.* Building Windows 8. Microsoft, February 2012. From `blogs.msdn.com` as of May 2012

My dissertation is motivated by a need to integrate non-functional requirements applications desire with system-wide decision-making. The following properties exemplify such cooperation opportunities:

Timeliness: User interface responsiveness and the smoothness of multimedia operations require that applications' timing requirements are met.[6] Such requirements arise when computers interface with the real world. Subsumed under the term real-time, these constraints tie the completion of software results to wall-clock time. A system-wide solution is needed, because time is a global resource. However,

[6] John E. Sasinowski, Jay K. Strosnider: *ARTIFACT: A Platform for Evaluating Real-Time Window System Designs.* Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS), pp. 342–352. IEEE, 1995

the majority of commodity systems today offer only weak temporal guarantees and predictability to applications, so developers have to work around those limitations. While real-time operating systems are available, they are typically applied only in special-purpose scenarios.[7]

[7] A notable exception is the BlackBerry PlayBook, a general-purpose tablet computer which runs the real-time capable QNX kernel. (cf. *QNX: Meet the Power Behind the BackBerry Tablet OS*)

Overload Detection: Overload situations occur when all ready applications collectively ask for more computation time than the machine can offer without violating any timing constraints. Real-time systems avoid such situations statically with an admission process: The system will not allow new tasks unless it can establish a formal guarantee that the total resource demand will never outgrow the available resources. On interactive systems, users will not accept rejected application launches.[8] These systems have to handle overload situations dynamically at runtime. I propose to aggregate knowledge of RESOURCE REQUIREMENTS to globally predict overload. Communicating these situations AHEAD OF TIME allows applications to prepare before the resource shortage arrives.

[8] *OS X Human Interface Guidelines.* Mac Developer Library, Chapter User Control. Apple Inc., July 2011. From `developer.apple.com` as of January 2012

Usage of Multiple Cores: Single-core performance is leveling off, so processor manufacturers rely on increasing core counts to offer higher performance. Applications thus have to employ parallel programming to benefit from multiple cores.[9] The ideas I present here build on top of a modern parallel runtime, but I leave the exploration of scheduling aspects on multiple cores for future work. Users only perceive core usage indirectly, timeliness and responsive behavior under overload are directly observable. For that reason I want to focus on the former two properties. But processors are a global resource, so assigning work to cores requires system-wide control. Therefore, I hint at extensions of my work toward contention-[10] and energy-aware[11] core placement at the end of this thesis.

[9] Herb Sutter: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, Volume 30 (3). UBM, March 2005

[10] Sergey Zhuravlev, Sergey Blagodurov, Alexandra Fedorova: *Addressing Shared Resource Contention in Multicore Processors via Scheduling.* Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 129–142. ACM, March 2010

[11] Krishna K. Rangan, Gu-Yeon Wei, David Brooks: *Thread Motion: Fine-Grained Power Management for Multi-Core Systems.* Proceedings of the 36th International Symposium on Computer Architecture (ISCA), pp. 302–313. ACM, June 2009

Non-functional properties such as timeliness and overload handling benefit from the aggregation of local application knowledge to implement a global system-wide policy. My mission is to show that a little added developer effort can be augmented by a newly designed runtime and lead to closer cooperation with the scheduler, enhancing application behavior with respect to non-functional properties. While I do not explore scheduling on multiple cores, the design does not rule it out. In this thesis, I demonstrate how communicating application's timing requirements helps overall timeliness, how knowledge on resource requirements enables overload detection and how ahead-of-time notification enables applications to react early.

DEVELOPING A MODERN APPLICATION is already a complex undertaking, because users expect intricate features, and applications have to handle new failure cases by connecting with the cloud. To face non-functional properties on top of that, developers need to rely on a foundation that helps them deliver without much development overhead. Systems research can and should help here. Libraries can mediate between the application's view and the system interface. I will now motivate my work from the system's perspective and show that the cross-cutting nature of non-functional properties calls for participation of the lower system levels.

Scheduler Knowledge

As illustrated by Figure 1.1, the scheduler is a system component responsible for accumulating information about the work applications want to perform and determine an order to execute that work. The ordering policy should serve the non-functional properties applications expect: It should fulfill timing requirements and detect overload early.

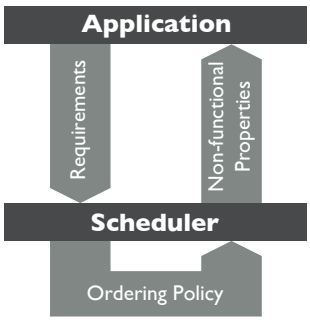


Figure 1.1: Interaction Between Application and Scheduler

Every device in a computer system has a different notion of work: sending and receiving network packets, reading and writing storage requests or the execution of graphics shader code. Consequently, every device follows a different policy for ordering its work and thus needs its own dedicated scheduler. This thesis focuses exclusively on scheduling a single CPU[12] core of a computer system. The CPU plays a central role in the system as it is the gateway for the scheduling of all other devices.[13] To submit work to a peripheral device, applications need to execute code on the CPU.

Nevertheless, taking a closer look at peripheral schedulers does help to reveal an important disadvantage of CPU scheduling: Schedulers for peripheral devices have a deeper insight into the jobs they are supposed to order.

[12] Central Processing Unit: the assembly of general purpose processors executing the operating system and application code (cf. *Wikipedia: Central Processing Unit*)

[13] Raj Rajkumar, Kanaka Juvva, et al.: *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems.* Proceedings of the 1998 Multimedia Computing and Networking Conference (MMCN), pp. 150–164. SPIE, January 1998

So how does a peripheral scheduler operate?[14] We start with an application asking for the service of a device. A write request to persistent storage shall serve as an example. The application collects the data it wants to make durable and submits a job toward the device. A write request to a file traverses file system and buffer caching code in the operating system and finally arrives at the device driver as a write request to the magnetic disk. As part of the driver, the scheduler maintains a work queue of jobs waiting for execution. Our write request is added to that queue together with concurrent requests arriving from other applications, from other threads of the same application, or even from the same thread of the same application, if the initial write executes asynchronously. Figure 1.2 depicts this situation.

[14] Robert Love: *Kernel Korner – I/O Schedulers.* Linux Journal, Volume 118. Belltown Media, February 2004

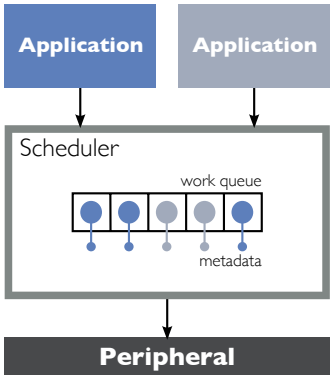


Figure 1.2: Peripheral Device Scheduling

The scheduler now orders the jobs and sends them off to the device for execution. The scheduling algorithm performs that ordering with a service goal in mind, for example to maximize device throughput. Every job carries metadata for the request. In the case of our disk write request, that metadata contains the size of the request and its location on disk.

When ordering jobs, the scheduler can inspect the metadata to decide which order best supports its service goal. The disk scheduler in the example can use the on-disk location to execute a shortest access time first policy[15] to improve drive throughput. Peripheral device schedulers enjoy the advantage that the outstanding requests are self-describing jobs. Inspecting them reveals all information the peripheral will use to execute them. This is natural, because to program the device, the driver must have all that information available anyway.

[15] David M. Jacobson, John Wilkes: *Disk Scheduling Algorithms Based on Rotational Position.* Technical Report HPL-CSP-91-7rev1, HP Laboratories, March 1991

The CPU scheduler does not share this benefit: It does not deal with self-describing jobs in a queue of outstanding work. Instead, it maintains a ready queue of runnable threads. Other than a job, a thread is not consumed after it executed, but is automatically inserted into the ready queue again until terminated by the application. Other than a job, a thread is not a self-describing aggregate of payload and metadata, but rather an opaque handle to an execution context within the application's address space. The actual behavior that unfolds if the thread is executing — whether it blocks after a short burst of code, or runs a long computation, or a periodic task — is hidden within the application's code and memory state as illustrated by Figure 1.3.

CPU schedulers in commodity operating systems support a notion of precedence, expressed with priorities or the Unix nice levels. This single numeric value[16] is a coarse abstraction of a thread's behavior, because it only indicates the importance of a thread relative to other threads. The application developer has to supply the priority without knowledge of concurrent load, which gives rise to other problems I discuss in Chapter 3.

Real-time schedulers supporting a periodic task model[17] have the benefit of implicit knowledge about a thread's upcoming behavior, but only for applications fitting into that rigid model. More details again follow in Chapter 3.

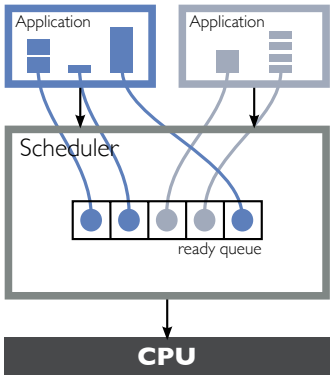


Figure 1.3: CPU Scheduling

[16] `nice` – *Change the Nice Value of a Process.* The Open Group Base Specifications, Volume 7. IEEE, 2008

[17] Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

In this thesis, I argue that it is beneficial to organize CPU scheduling using self-describing jobs similar to peripheral schedulers. Those CPU jobs represent a specific piece of code execution and are consumed once the CPU executed them. They carry metadata describing the jobs' urgency and required execution time. If the CPU scheduler receives jobs ahead-of-time, before they execute, it can build up a limited look into the applications' future and detect overload situations early. In contrast to periodic task models, this knowledge is not available implicitly, but applications share information with the scheduler by submitting jobs explicitly. I show how such a scheduling regime improves the non-functional application properties portrayed above.

Driving Insights

The walk-through of the problem area shows that the considered non-functional properties timeliness and overload detection are cross-cutting concerns. They involve local knowledge from the applications and global policy executed by the scheduler. Furthermore, many applications act as an execution environment tailored to a specific workload: a text editor manipulates documents, a photo album manages photos, a video player renders continuous media. Those applications' behavior dynamically depends on the workload they handle. This workload is what users care about after all.

A vertically integrated solution spanning from the workload through the application down to the scheduler is called for.

Refining the previous concept sketch from Figure 1.1, Figure 1.4 shows how I imagine this integration to operate. Applications should be aware of their workload and extract information to model it. They communicate a useful representation of their local knowledge down to the scheduler. The scheduler collects this knowledge from all applications and executes its global policy, consequences of this policy take effect either implicitly by executing the scheduling decisions, or explicitly by the scheduler reporting back to an application. Overload situations propagate this way, because they require an application-specific reaction to resolve them. The application in turn uses its model of the workload to decide on a quality-aware reaction.

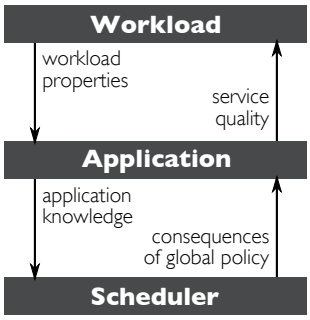


Figure 1.4: Vertically Integrated Solution

What information is relayed along those paths and what the interfaces look like remains to be discussed. But as motivated in the section on scheduler knowledge above, the scheduler should operate on self-contained jobs instead of threads which hide their execution behavior deep in the application state.

Applications should expose local knowledge to the scheduler by explicitly submitting self-describing jobs that encapsulate timing and resource requirements depending on the current workload.

To provide knowledge explicitly, applications likely have to be modified. We may get away with adapting key libraries to change the behavior of multiple applications at once, but in the context of this dissertation, modifying a library counts as modifying the application. How to reduce programming effort by architecting software so it hides this problem in library layers is outside the scope of this thesis. In scope however is to design the interfaces for ease of use:

The programming model should be approachable by matching current application development methods and by never asking the developer for parameters outside the application domain.

I hope developers can provide parameters from within the application domain with reasonable effort. An example are deadlines to describe the timing requirements of a job. Parameters requiring knowledge outside the application scope should be avoided. Developers should not need to provide estimated execution times of jobs, because these are specific to the underlying hardware platform.

An emerging trend in the development of parallel software is the use of lambdas,[18] depending on the programming language also called closures or blocks. They are used to structure code into pieces that can be executed asynchronously, which keeps applications responsive in the presence of long-running background computation or blocking operations. Because asynchronously executed code runs concurrently with other code, lambdas are also a tool to express parallelism. A more in-depth discussion of this programming style follows in Chapter 2. I will illustrate how lambdas and jobs can cooperate — both describe a bounded, self-contained piece of code execution — and how the com-

[18] Herb Sutter: *Lambdas, Lambdas Everywhere.* Session at Microsoft's Professional Developers Conference (PDC), October 2010

munication of jobs to the scheduler integrates with the use of lambdas. Big platform vendors such as Apple[19] and Microsoft[20] adopted lambda programming, so by drafting a scheduler interface that leans toward it, I hope to create a pragmatic solution that developers can adopt as part of their programming toolbox.

The asynchronous execution of lambdas gives rise to another interesting benefit: announcing future code execution ahead-of-time. Application code can specify multiple pieces of work and dispatch them for later asynchronous and potentially parallel execution. The queues in the lambda runtime therefore contain knowledge about what pieces of code will execute in the future. Again, more details on how these runtimes operate follow in Chapter 2. Here we conclude:

KNOWLEDGE OF FUTURE EXECUTION should be propagated to the scheduler.

I expect many applications to have knowledge on their future execution available, but they currently lack a way to expose this information. Any operation that runs autonomously after being triggered by a system event or user interaction is fully determined at the moment it is triggered. If a user clicks "play" to watch a video, the application knows that it will now be fetching, decoding and displaying video frames.

Like video, some of these chains of actions may be long-running, others may be short, like the reaction to a user's mouse click in a graphical interface. Highly interactive applications like games may have almost no knowledge of what will happen next, because the user can change the course of action at any moment. However, applications that do have knowledge of their future should be able to tell the scheduler about it early. Such insights enable the system to perform LOOK-AHEAD SCHEDULING: It can make ahead-of-time, anticipating decisions rather than exercising post-mortem, reactive control.

Thesis Goals

In this dissertation, I target two scheduling-related problems:

- timeliness and
- overload detection.

I chose these problems because I believe they constitute important non-functional properties, which applications should offer to their users. Therefore, I investigate these problems in the context of interactive end-user systems, running a commodity operating system[21] as an application platform. This category of systems includes classical desktop computers, notebooks and the ballooning family of smartphones and tablets. It does not include servers, although I am confident that my ideas generalize to this class of systems due to the large amount of shared technology.

[19] *Introducing Blocks and Grand Central Dispatch*. Mac Developer Library. Apple Inc., August 2010. From `developer.apple.com` as of January 2012

[20] Kenny Kerr: *Visual C++ 2010 and the Parallel Patterns Library*. MSDN Magazine. Microsoft, February 2009. From `msdn.microsoft.com` as of July 2011

[21] *Top 5 Operating Systems in 2011*. StatCounter Global Stats. StatCounter, 2011. From `gs.statcounter.com` as of January 2012

I DO NOT CONSIDER REACTIVE SYSTEMS[22] that continuously supervise sensors and actuators, for example in an industrial control environment or within other deeply embedded systems. Graphical and touch user interfaces however are subject to timing constraints dictated by the physical reality. Here, human-machine-interfaces and reactive systems overlap.[23] Beyond these common timeliness demands, I ignore reactive systems in this work.

Similarly, I do not consider offline scheduling, where a precomputed schedule is reenacted at runtime. In a dynamic, interactive system, advance knowledge on the executed task set is generally not available. Therefore, I exclusively research online scheduling, where scheduling decisions happen while the system runs.

Real-time literature[24] distinguishes between systems with hard, firm and soft deadlines. Figure 1.5 shows the delineating characteristics. Hard deadlines must never be missed and the system has to guarantee this invariant with a formal analysis. For firm and soft deadlines, weaker guarantees apply. Deadlines may be missed, but with predictable consequences. Jobs missing a firm deadline are aborted, whereas results arriving after a soft deadline are still useful.

I think an interactive system should not reject the user's instruction to start an application. Therefore, a task admission is not appropriate and consequently, the system cannot handle hard deadlines. Overload situations may occur and my solution provides applications the means to handle them. It is up to the application to decide on aborting the job or continuing.

[22] David Harel, Amir Pneuli: *On the Development of Reactive System.* Logics and Models of Concurrent Systems, pp. 477–498. Springer, 1985

[23] Nicholas Halbwachs, Paul Caspi, et al.: *The Synchronous Data Flow Programming Language LUSTRE.* Proceedings of the IEEE, Volume 79 (9): pp. 1305–1320. IEEE, September 1991

[24] Jane W. S. Liu: *Real-Time Systems.* Prentice Hall, Edition 1, April 2000

Figure 1.5: Types of Real-Time Deadlines

THREE LAYERS HAVE TO BE BRIDGED for a comprehensive solution:

- workload,
- application, and
- CPU scheduler.

I presume that combining application-specific knowledge on workload and execution behavior with system-wide knowledge on urgency and overall load is beneficial. Any solution that handles any of these aspects in isolation will be incomplete. Instead, applications and the scheduler have to collaborate[25] to integrate application-local and global scheduling mechanisms. I propose self-describing jobs as this integrative device. To encourage adoption, application developers should not face different programming paradigms for different non-functional properties, but one wholesale solution.

[25] Simon Peter, Adrian Schüpbach, et al.: *Design Principles for End-to-End Multicore Schedulers.* Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar). USENIX Association, June 2010

INTEGRATION across the three layers is necessary, because no layer alone has enough knowledge to provide the desired non-functional property. The following Table 1.1 summarizes the thesis goals:

	Thesis Goal	Chapter	Enabling Part of ATLAS
Workload	derive workload metrics to predict execution times	Chapter 4: Execution Time Prediction	Auto-Training
Application	expose deadlines and execution time estimates ahead of time	Chapter 3: Real Simple Real-Time	Look-Ahead
CPU Scheduler	globally order jobs to meet deadlines, detect misses ahead of time	Chapter 5: System Scheduler	Scheduler

Table 1.1: Overview of the Solution Developed in This Thesis

A large body of individual research results is available covering subsets of my three goal components. In this dissertation I provide a comprehensive, end-to-end solution whose design is rooted in the needs of applications. The discussion therefore starts with the application layer in Chapter 3, continues with workload modeling in Chapter 4 and finishes with a matching kernel scheduler in Chapter 5.

The scheduling system I present is dubbed ATLAS, the Auto-Training Look-Ahead Scheduler.[26] I now give an overview of the work and summarize the key contributions. I want to point out again that the work as presented here only explores scheduling for a single CPU core. I sketch an extension for multiple cores, but I have not implemented it.

[26] Like its namesake, the Greek titan who supports the celestial globe (cf. *Wikipedia: Atlas*), I hope the ATLAS system can support applications and developers.

Timeliness and Overload

Timeliness is the primary property of real-time scheduling. The scheduling policy considers secondary service goals only when timeliness is not jeopardized. Time is a global resource and is therefore managed by a system-wide scheduler. Imposing a separation of concerns as in the resource kernels concept,[27] applications specify their timing requirements to the scheduler, which then exercises global management. This functional separation must be accompanied by an integration of knowledge: Only the application knows its workload and should expose this knowledge to the scheduler using appropriate interfaces.

[27] Raj Rajkumar, Kanaka Juvva, et al.: *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems.* Proceedings of the 1998 Multimedia Computing and Networking Conference (MMCN), pp. 150–164. SPIE, January 1998

ATLAS does not enforce an admission process. I think rejecting application launches or new subtasks within an application would be too surprising for the user and the developer. Consequently, the system can experience overload situations. Lacking an admission that can analytically prevent overloads, ATLAS instead offers mechanisms to handle overload gracefully.

Overload can occur for two reasons: First, an application can lie about its CPU demand. It announces timing and resource requirements that the scheduler can meet, but at runtime, the application's jobs run longer than reported. The scheduler handles such situations by ensuring that lying applications never degrade the service of honest applications.

The second overload condition is more interesting: Applications specify their demands correctly, but collectively ask for more CPU

resources than available. In such a situation, service degradation is unavoidable. The best the system can do is detect the overload and warn applications so they may apply custom strategies to adapt.[28] Interaction between scheduler and applications is therefore needed.

Developers can implement custom degradation strategies like load-shedding or imprecise computation[29] that take an application-specific notion of quality into account. I published a quality-aware adaptation technique for video playback in the Journal of Visual Communication and Image Representation.[30]

I CONTRIBUTE A SCHEDULER INTERFACE and an accompanying task model and runtime infrastructure that allow applications to express timing and resource requirements. Other than the majority of related work, I do not employ a periodic task model.[31] Explicit submission of future jobs substitutes the implicit knowledge on future execution that periods provide.

The ATLAS interface as seen by the application developer only asks for parameters from the application domain. Deadlines specify timing requirements. Resource requirements are specified using WORKLOAD METRICS: parameters from the workload that describe its computational weight. ATLAS uses machine-learning to automatically derive execution time estimates from the metrics before the actual execution. The estimated execution time of each job is therefore known ahead of time. I presented the prediction method employed by ATLAS on the 27th IEEE Real-Time Systems Symposium[32] and the overall ATLAS architecture on the 19th IEEE Real-Time and Embedded Technology and Applications Symposium.[33]

The ATLAS runtime has been designed with current programming trends in mind. It explores the interaction between real-time jobs and asynchronous lambdas. I believe this combination has not been investigated yet. I demonstrate with code examples that the task model is easy to program against. However, a formal usability analysis of the programming interface is not part of this thesis.

I CLAIM THAT THE FLEXIBILITY OF THIS TASK MODEL allows to inform the scheduler more accurately of application behavior than alternative approaches. I validate this claim by demonstrating that ATLAS can predict the future execution of applications precise enough to anticipate deadline misses before they occur. Outside the implicit clairvoyance of periodic task systems, no other scheduling system I am aware of features a comparable look-ahead capability.

Literature mentions look-ahead together with scheduling in the areas of constraint satisfaction problems,[34] factory scheduling,[35] and server management for on-demand video.[36] For constraint satisfaction problems and factory scheduling, look-ahead improves the traversal of the solution search space. Thus, the scheduling does not look ahead along the time axis into the future, but along the search tree into the solution space. In the on-demand video context, the server tries to batch multiple viewers of the same video to save disk requests. Look-ahead and buffering help the server to satisfy multiple users from the same disk

[28] Damir Isović, Gerhard Fohler: *Quality Aware MPEG-2 Stream Adaptation in Resource Constrained Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS), pp. 23–32. IEEE, June 2004

[29] Kwei-Jay Lin, Swaminathan Natarajan, Jane W. S. Liu: *Imprecise Results: Utilizing Partial Comptuations in Real-Time Systems.* Proceedings of the 8th IEEE Real-Time Systems Symposium (RTSS), pp. 210–217. IEEE, December 1987

[30] Michael Roitzsch, Martin Pohlack: *Video Quality and System Resources: Scheduling Two Opponents.* Journal of Visual Communication and Image Representation, Volume 19 (8): pp. 473–488. Elsevier, December 2008

[31] Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

[32] Michael Roitzsch, Martin Pohlack: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video.* Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 271–280. IEEE, December 2006

[33] Michael Roitzsch, Stefan Wächtler, Hermann Härtig: *ATLAS: Look-Ahead Scheduling Using Workload Metrics.* Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–10. IEEE, April 2013

[34] Andrea Schaerf: *Combining Local Search and Look-Ahead for Scheduling and Constraint Satisfaction Problems.* Proceedings of the 15th International Joint Conference on Artifical Intelligence (IJCAI), Volume 2, pp. 1254–1259. Morgan Kaufmann, 1997

[35] Kenji Itoh, D. Huang, Takao Enkawa: *Twofold Look-Ahead Search for Multi-Criterion Job Shop Scheduling.* International Journal of Production Research, Volume 31 (9): pp. 2215–2234. Taylor & Francis, September 1993

[36] Philip S. Yu, Joel L. Wolf, Hadas Shachnai: *Design and Analysis of a Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications.* Multimedia Systems, Volume 3 (4): pp. 137–149. Springer, September 1995

stream, even if they independently pause and resume playback. Patterson et al. investigated the submission of future jobs to a peripheral scheduler to improve prefetching for input devices.[37]

I also evaluate the timeliness delivered by Atlas due to its compliance with the requested timing requirements. I compare with the behavior under conventional fair share scheduling without timing constraints. This evaluation serves to convince the reader of the scheduler's basic functionality with respect to the timeliness property. I do not claim to improve the state of the art in this aspect.

I describe the real-time task model in Chapter 3, discuss the execution time prediction in Chapter 4 and evaluate the scheduler in Chapter 5. These chapters also comprehensively explain the architecture of Atlas. I use a complete implementation stack from application runtime down to an in-kernel scheduler to experiment with timeliness and overload detection.

[37] R. Hugo Patterson, Garth A. Gibson, et al.: *Informed Prefetching and Caching.* Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), pp. 79–95. ACM, December 1995

Core Placement

I do not make a formal contribution in this area, but I want to convince the reader that the Atlas design is compatible with parallel execution and scheduling. Managing multiple cores adds another dimension to the scheduling problem: Not only does the scheduler order jobs along the time axis, but it also has to decide on a placement of the work onto cores. To this end, I sketch an extension toward core placement in Chapter 7.

Serving the timeliness goal, applications structure their execution into jobs that carry a timing requirement. To enable parallel processing, applications must also structure their work into independent pieces that can execute simultaneously.[38] To simplify development, I intend to decouple these two structures by allowing for parallelism within an individual job. Collette et al. presented a similar task model before,[39] but without showing a programming environment to present it to developers. By building on top of an asynchronous lambda runtime, Atlas hands developers a powerful tool for parallel programming.

[38] Michael Roitzsch: *Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding.* Proceedings of the 7th International Conference on Embedded Software (EMSOFT), pp. 269–278. ACM, October 2007

[39] Sébastien Collette, Liliana Cucu, Joël Goossens: *Integrating Job Parallelism in Real-Time Scheduling Theory.* Information Processing Letters, Volume 106 (5): pp. 180–187. Elsevier, May 2008

Demo Application

The primary example workload in this thesis is video playback. I chose video because I think it is representative for a number of real-time and throughput applications due to its combination interesting properties:[40]

- Video playback is subject to deadlines that derive naturally from the frame rate specification or the presentation timestamps in the video stream. Timing requirements are tight, because even small delays in the frame display will be visible to the user.[41]
- Although decoding and displaying video frames are repetitive tasks, they do not adhere to the classical periodic task model where the deadline of one job coincides with the release of the next job. Due to

[40] Veronica Baiceanu, Crispin Cowan, et al.: *Multimedia Applications Require Adaptive CPU Scheduling.* Proceedings of the RTSS Workshop on Resource Allocation Problems in Multimedia Systems Scheduling. IEEE, December 1996

[41] The telecine conversion of 24 frames/s cinematic content to 30 frames/s for NTSC television causes timing errors smaller than 20 ms, which are visible to some viewers as jerky motion, especially in scenes with slow and steady camera movement. This error is called telecine judder (cf. *Wikipedia: Telecine*).

buffering in the video player, the jobs' scheduling windows between release and deadline can overlap.

- Video playback can be CPU intensive, especially with high resolutions and modern coding schemes. Figure 1.6 visualizes the single-thread decoding times of the 4096×1744 pixel Sintel-4k video[42] on a 2.4 GHz Intel Core i5. The high utilization is challenging for schedulers because it increases the likelihood that misdirection of CPU time leads to deadline misses. In the figure, the marked decoding time equivalent to 24 frames per second shows that a significant portion of the frames need more time for decoding than available between the display instants of two consecutive frames.

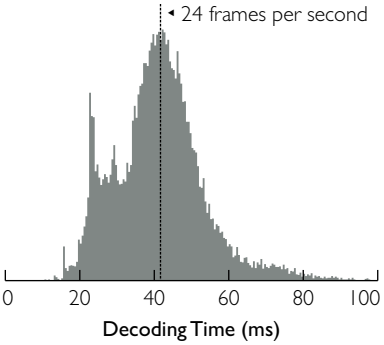


Figure 1.6: Histogram of Sintel-4k Decoding Times

- The CPU load is also highly dynamic. Figure 1.7 illustrates the short-term and long-term variations. The execution times of jobs are therefore difficult to predict and worst-case estimates would prohibitively over-allocate resources.

[42] Such video dimensions may not be common today, but they are already used in cinematic applications. With high resolution displays coming to market, such videos may appear on desktops soon. The properties of this video are summarized in Table 4.3 on page 60.

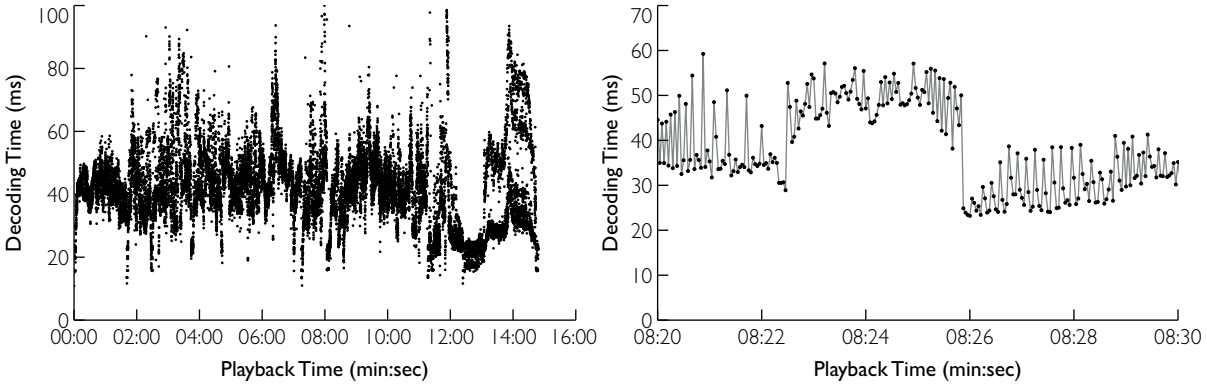


Figure 1.7: Total and Zoomed Views of Sintel-4k Decoding Times

- When overloaded, a video player conventionally adapts by dropping frames. However, more sophisticated degradation options like low-quality decoding fallbacks are also available and can improve quality per invested CPU cycles.

By showing that Atlas can support video playback in these aspects I hope to convince the reader that it also handles other applications with a subset of the stated properties. Figure 1.8 condenses typical behavior of selected desktop real-time tasks. Although video is the running example throughout this thesis, I complement it with other experiments to substantiate the evaluation.

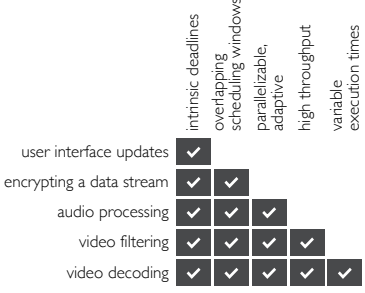


Figure 1.8: Typical Properties of Selected Real-Time Applications

Before we dive into the details of the timeliness and overload detection aspects of Atlas, the next chapter gives an overview of the emerging use of lambdas as the programming model for concurrent desktop applications. The intimate vertical integration proposed by Atlas requires developer endorsement, so taking a closer look at programming innovation is valuable. Aligning my task model with a future-proof paradigm helps simplifying the use of Atlas for developers. Building Atlas around a parallel runtime keeps the door open for future extensions toward scheduling multiple cores.

Anatomy of a Desktop Application

2

For Atlas, I envision a tight integration of application, runtime libraries and the CPU scheduler. The following pages explain the programming style of modern desktop applications, demonstrate how Atlas complements this style and motivate the benefits this combination offers. Apart from guiding ideas framed in the context of real-world programming, I do not claim a distinct scientific contribution, but rather give an understanding of my target applications and explain basic terminology.

Three major paradigm shifts changed the way desktop software is written, with the latest of these transitions still in progress. Historically, all applications started out single-threaded. The traditional Unix design only knows about processes that implicitly host only a single activity.[1] There was no notion of multiple threads per address space. The resulting applications had no internal parallelism, which simplified programming and freed libraries from the worries of reentrant calls.

However, this model made the developers' lives easier at the price of user irritation: When an application synchronously performs an input or output operation, for example reading a large file from disk or waiting for data from the network, its only thread is stuck in a blocking system call. Consequently, that thread is now unavailable for user interactions. The application appears dead for a while. The visual cue for such a mishap are redraw artifacts that tail other applications' windows when the user drags them over the blocked application. They appear like in Figure 2.1 because the blocked application underneath no longer updates its window content and thus never clears the ghost copies of the window on top.

Recognizing the importance of responsiveness, developers started to migrate long-running or blocking work off the main thread. Two mechanisms to support this change were asynchronous interfaces and the use of multiple threads. Both have their share of problems. Asynchronous interfaces split the logical code flow into two parts: The setup of context and arguments prepares the asynchronous invocation. Results come in as the asynchronous operation finishes. Often, a callback by the underlying framework executes this second part, which riddles the code with additional functions just for callback purposes. This callback-soup[2] complicates code understanding. Furthermore, because the callback executes in a separate function, the context

[1] Andrew Josey (Editor): *Go Solo 2 – The Authorized Guide to Version 2 of the Single UNIX Specification*, Chapter 10: Thread-Safety and POSIX.1. Prentice Hall, May 1997

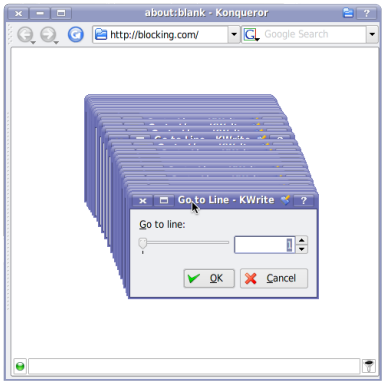


Figure 2.1: Blocked Window Redraws

[2] Francisco Sant'Anna: *The Problem with Events*. The Synchronous Blog. Private Blog, August 2008. From `thesynchronousblog.wordpress.com` as of March 2012

of local variables is different from when the asynchronous processing started, a problem known as stack ripping.[3]

On the other hand, it is also difficult for programmers to orchestrate multiple threads correctly. Threads do not suffer from stack ripping, because they individually follow a single control flow with a dedicated stack. However, as the number of concurrent threads increases, their interleaving becomes increasingly complex to manage. Programmers unknowingly make wrong assumptions on the execution order of code in different threads.[4] As mainstream computers featured only a single CPU core, those assumptions used to hold. But when multiple CPU cores made their way into commodity machines, many multithreaded programs suddenly broke.

A NEW APPROACH TO CONCURRENCY IS NEEDED. The current shift in parallel programming[5] leads us away from threads and closer to the ideas behind asynchronous invocation, but without inheriting their historical callback-soup and stack ripping downsides. Today, developers are discouraged from using threads directly, but should instead use new programming language constructs called lambdas, closures,[6] or blocks that allow them to envelop a piece of code and dispatch it for asynchronous invocation.

The name for this paradigm changes with the chosen programming language and runtime environment. The term task-based programming[7] has been coined, but also covers traditional callback-style models. Given that the term "task" is hopelessly overloaded already, I will avoid it and call the programming style ASYNCHRONOUS LAMBDAS instead.

Figure 2.2 shows four pseudo-code versions — from the historical serial version to the modern lambda-style — of a user-initiated long-running computation, which updates the user interface when finished:

[3] Atul Adya, Jon Howell, et al.: *Cooperative Task Management Without Manual Stack Management.* Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC), pp. 289–302. USENIX, June 2002

[4] Shan Lu, Soyeon Park, et al.: *Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics.* Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 329–339. ACM, March 2008

[5] Herb Sutter: *Lambdas, Lambdas Everywhere.* Session at Microsoft's Professional Developers Conference (PDC), October 2010

[6] Jaakko Järvi, John Freeman, Lawrence Crowl: *Lambda Expressions and Closures: Wording for Monomorphic Lambdas.* Technical Report N2550-08-0060, C++ Standards Committee, February 2008

[7] Ron Fosner: *Task-Based Programming – Scalable Multithreaded Programming with Tasks.* MSDN Magazine. Microsoft, November 2010. From `msdn.microsoft.com` as of May 2012

```
void mouseClick(event)
{
  view = event.target;
  result = longWork();
  view.update(result);
}
```

Serial Version

```
void mouseClick(event)
{
  view = event.target;
  context = { view };
  async(longWork,
    cb, context);
}

/* callback */
void cb(result, context)
{
  view = context[0];
  view.update(result);
}
```

Asynchronous with Callback

```
void mouseClick(event)
{
  view = event.target;
  context = { view };
  thread_create(work,
    context);
}

/* worker thread */
void work(context)
{
  view = context[0];
  result = longWork();
  view.update(result);
  thread_exit();
}
```

Multiple Threads

```
void mouseClick(event)
{
  view = event.target;
  async(^{
    result = longWork();
    view.update(result);
  });
}
```

Asynchronous Lambda

Figure 2.2: Pseudo-Code Examples for Long-Running Mouse Click Processing

The Serial Version is the shortest of the four and the easiest to read, because it describes the logical order of processing steps without much distraction by language clutter: First, the user interface element `view` is fetched by evaluating where the user clicked. The long computation follows and its result updates the representation on screen.

The Callback Version illustrates the stack ripping problem: What used to be a continuous control flow representing a single conceptual job is now broken across two functions. The programmer must explicitly transfer local state from the first function, if the second function needs access to it. After registering `longWork` for execution, control returns to the runtime, which invokes the callback as the result is available. The callback function and the context act as a continuation[8] to complete the job.

Multiple Threads similarly require manual passing of contextual state between the invoker and the worker thread. Processing is again split across two functions. A new problem introduced in this version is the updating of the view from a background thread. Many user interface frameworks limit updates to the main thread to avoid dealing with concurrent invocation.[9] Such a limitation further complicates the use of multiple threads.

The Lambda Version on the very right invokes a block of code asynchronously, but without using a callback or manually managing separate threads. In fact, using lambdas, the code regains a lot of the simplicity of the serial version, which is the main reason why this programming style is so attractive. It may appear as syntactic sugar in this toy example, but such assistance encourages new paradigms for concurrent and responsive program design.[10] The pseudo-code lambda in the figure uses the block syntax from Apple's Grand Central Dispatch (GCD) system,[11] where a caret ^ marks the lambda code. The notation is not simplified, the example shows all typing the programmer has to do. The `view` variable, which was assigned outside the lambda, is transparently available inside the lambda without any manual state transfer. If libraries impose concurrency limitations, GCD allows for seamless dispatching of work back to the main thread.[12]

Threads are still used behind the scenes to implement the asynchronous execution of lambdas, but the programmer does not need to bother. This avoids explicit thread management which often leads to an inflexible hard-coded assignment of work to threads that may not be optimal on the end-user's machine. Instead, the runtime environment can manage the number of active threads, taking into account the number of cores and the current load on the machine.

[8] John C. Reynolds: *The Discoveries of Continuations.* LISP and Symbolic Computation, Volume 6 (3): pp. 233–247. Kluwer, 1993

[9] *Threading Programming Guide.* Mac Developer Library, Chapter Thread Safety Summary. Apple Inc., April 2010. From `developer.apple.com` as of May 2012; and *Threading Basics.* Qt Reference Documentation, 4.7. Digia, 2010. From `doc.qt.digia.com` as of May 2012

[10] Greg Cowin: *Why Is the Next Major Paradigm Shift in Software Design About to Happen?* In Pursuit of Great Design. Private Blog, November 2011. From `www.pursuitofgreatdesign.com` as of February 2013

[11] *Introducing Blocks and Grand Central Dispatch.* Mac Developer Library. Apple Inc., August 2010. From `developer.apple.com` as of January 2012

[12] *Concurrency Programming Guide.* Mac Developer Library, Chapter Performing Tasks on the Main Thread. Apple Inc., July 2012. From `developer.apple.com` as of August 2012

Lambdas Far and Wide

Asynchronous lambdas combine a programming pattern based on asynchronous invocation and a lambda language feature to deliver a unique combination of properties.[13]

Asynchronous invocation originates from event-based programming.[14] High-throughput network servers experience scalability bottlenecks when assigning each incoming request to a dedicated thread. Performance improves when organizing the server such that queues collect pieces of work and execute them asynchronously.[15] Queuing work instead of explicitly spawning a thread allows the runtime layer to control the number of threads, employing strategies like transparent thread pooling.

Lambdas automatically transfer state by capturing variables from the enclosing scope. This relieves the programmer from manually collecting contextual data. Lambdas also preserve the logical locality of the code, because they appear inline and not as separate functions. The code looks like a serial flow of instructions and is easier to read than code written for threaded or callback parallelism.

The compiler and runtime combine the lambda's code block and the necessary state to an object that can be passed as an argument to functions, making this language feature a natural fit for the pattern of asynchronous invocation. Herb Sutter believes lambdas are "The Holy Grail" for developing a library of parallel programming patterns.[16] Early academic research pioneered many concepts of parallel language runtimes[17] or work queue management strategies.[18] Now, these systems appear as mature industrial solutions.

Existing implementations of asynchronous lambdas cover all or a significant subset of the four aspects of the programming paradigm:

- logical coherence of the code,
- automatic state capturing,
- asynchronous execution, and
- automatic thread management.

To show that this concept is now gaining traction, the following Table 2.1 presents a selection of popular complete and partial implementations, including their name for the unit of code execution, for which the different solutions are unfortunately not in agreement.

Two complete solutions are available from commercial operating system vendors: Microsoft offers the Parallel Patterns Library,[19] a library on top of standard C++ lambdas, and Apple is pushing for Grand Central Dispatch,[20] a combined C language extension and work queue library. Both offer fully featured lambdas that can be dispatched for asynchronous execution with transparent thread management by the underlying framework.

Goroutines in Google's Go language[21] are functions that execute

[13] Marcus Völp, Michael Roitzsch: *Elastic Manycores – How to Bring the OS Back Into the Scheduling Game?*, June 2013. In submission to the 1st Workshop On Runtime and Operating Systems for the Many-core Era

[14] Maxwell Krohn, Eddie Kohler, M. Frans Kaashoek: *Events Can Make Sense.* Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC). USENIX, June 2007

[15] Matt Welsh, David Culler, Eric Brewer: *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.* Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 230–243. ACM, October 2001

[16] Herb Sutter: *Heterogeneous Computing and C++ AMP.* Keynote at AMD Fusion Developer Summit, June 2011

[17] Stephan Murer, Jerome A. Feldman, et al.: *pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation.* Technical Report TR-93-028, International Computer Science Institute, December 1993

[18] Robert D. Blumofe, Christopher F. Joerg, et al.: *Cilk: An Efficient Multithreaded Runtime System.* Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 207–216. ACM, July 1995

[19] Kenny Kerr: *Visual C++ 2010 and the Parallel Patterns Library.* MSDN Magazine. Microsoft, February 2009. From `msdn.microsoft.com` as of July 2011

[20] *Introducing Blocks and Grand Central Dispatch.* Mac Developer Library. Apple Inc., August 2010. From `developer.apple.com` as of January 2012

[21] *The Go Programming Language Specification.* Google Inc., June 2012

Table 2.1: Complete or Partial Implementations of Asynchronous Lambdas

Implementation	Code Unit	Logical Coherence	State Capture	Async. Execution	Thread Management
Apple Grand Central Dispatch	Block	✓	✓	✓	✓
Microsoft Parallel Patterns Library	Lambda	✓	✓	✓	✓
Google Go	Goroutine	✓	✓	✓	✓
X10	Activity	✓	✓	✓	✓
C++11 (no additional libraries)	Lambda	✓	✓	✓	
Intel Threading Building Blocks	Task	✓	✓		✓
OpenMP	Iteration	✓			✓
Galois	Iteration	✓			✓
Qt Signals and Slots	Event			✓	✓
CoreManager	Task			✓	✓

asynchronously, with the Go runtime managing threads automatically. Using anonymous inline functions, developers can write Goroutine code in place, without branching off to a separate function. Like C++ lambdas, Go closures can access variables from the surrounding context and capture them by value or by reference.

IBM developed X10, a new language for parallel programming,[22] which also offers a full implementation of asynchronous lambdas. In addition, X10 also introduces a PLACES concept to represent locality in a partitioned address space. Places host activities, which is the X10 name for lambdas.

C++11 offers lambdas as part of the language,[23] but thread management for parallel execution is incomplete[24] and needs to be supplied by an additional library like Intel's Threading Building Blocks.[25] Its primitives however are tailored for parallelizing iterative and pipelined work, not for general asynchronous execution.

OpenMP[26] employs a model of strict fork-join-concurrency. The programmer marks intermittent parallel sections in otherwise serial code. Figure 2.3 on the next page shows how the master thread fans out into a number of worker threads to execute parallel work. The focus is on parallelizing loops, with an automatic assignment of iterations to threads.

Like OpenMP, the Galois research runtime parallelizes iterative work. However, it extends the traditional parallel loop to a dynamic set iterator, where executing one iteration can add new iterations to a work set.[27] While parallel loops suffice to implement array processing or dense matrix operations, the generalization to set iterators helps with irregular algorithms like traversing graphs or other pointer-based data structures. For the latter class of algorithms, parallelism is data-dependent and builds up at runtime. The runtime is therefore unable to split the iteration space among threads at the beginning of the iter-

[22] Philippe Charles, Christian Grothoff, et al.: *X10: An Object-Oriented Approach to Non-Uniform Cluster Computing.* Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 519–538. ACM, October 2005

[23] *ISO/IEC 14882:2011: Programming Languages – C++.* ISO, Edition 3, September 2011

[24] Bartosz Milewski: *Async Tasks in C++11: Not Quite There Yet.* Bartosz Milewski's Programming Cafe. Private Blog, October 2011. From `bartoszmilewski.Com` as of July 2012

[25] *Intel® Threading Building Blocks.* Intel Corporation, October 2011

[26] *OpenMP Application Program Interface.* OpenMP Architecture Review Board, Edition 3.1, July 2011

[27] Milind Kulkarni, Keshav Pingali, et al.: *Optimistic Parallelism Requires Abstractions.* Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 211–222. ACM, June 2007

ation, but must dynamically assign work while the iterator progresses. Figure 2.4 illustrates the resulting irregular parallelism.

Among the application frameworks popular on Linux, the Qt library provides a mechanism called Signals and Slots to flexibly dispatch function invocation. This mechanism works across threads[28] and allows for asynchronous execution of work in the background. Automatic thread management is available by way of a thread pool facility.

Even hardware is trending towards asynchronous programming models, forecasting the relevance for future heterogeneous manycore chips.[29] CoreManager[30] is a scheduler that dispatches tasks to processing elements on a heterogeneous multiprocessor. The Tomahawk[31] research chip is based on a hardware implementation of the CoreManager concept. Allocation of processing elements is transparent to the programmer and the main application processor continues executing while a task is running asynchronously on a processing element. Although input and output data of a task must be marked explicitly, the memory content is automatically transferred by a memory controller. Similar hardware-implemented scheduling like Nvidia's Gigathread engine[32] operates in today's GPUs.

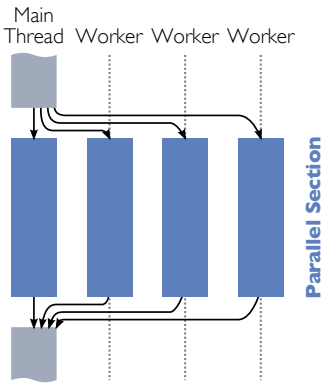


Figure 2.3: OpenMP Fork-Join-Model

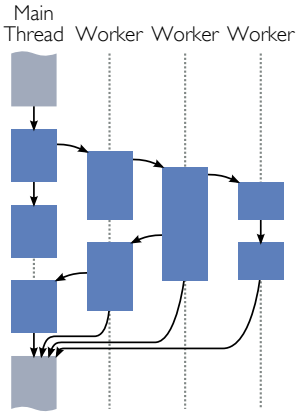


Figure 2.4: Galois Set Iterator

The Lambda Style

Throughout this thesis, I use the Grand Central Dispatch (GCD) implementation of asynchronous lambdas to illustrate examples and to demonstrate the ATLAS scheduling concepts. However, I believe my work does not rely on specifics of GCD and generalizes to any complete asynchronous lambda system. The two reasons for siding with GCD are:

First, GCD is a mature and fully-featured implementation of asynchronous lambdas that enjoys the backing of a major company. It is actively used by developers writing applications for Apple's OS X and iOS. GCD is becoming an essential technology on these platforms, with older frameworks being rewritten to use it and newer interfaces relying on it exclusively.[33]

Second, unlike Microsoft's Parallel Patterns Library or Intel's Threading Building Blocks, GCD does not require C++ and is therefore compatible with existing C-code. GCD consists of a C language extension called blocks and the libdispatch runtime library. The clang compiler[34] implements the blocks extension and libdispatch is available from Apple.[35] Both are open-source licensed and have been ported to Linux,[36] turning GCD into a suitable research platform with complete source code available.

THE TWO BASIC PRIMITIVES OF GCD are blocks and dispatch queues. Blocks implement the lambda part of asynchronous lambdas. They turn a piece of executable code into a language object, but other than a function they are written inline and automatically capture surrounding variables as needed within the block. The example in Figure 2.5 demonstrates these properties:

[28] *Threads and QObjects.* Qt Reference Documentation, 4.7, Chapter Threads and QObjects. Digia, 2010. From `doc.qt.digia.com` as of October 2012

[29] Rob Knauerhase, Romain Cledat, Justin Teller: *For Extreme Parallelism, Your OS Is Sooooo Last-Millennium.* Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar). USENIX Association, June 2012

[30] Oliver Arnold, Gerhard Fettweis: *Power Aware Heterogeneous MPSoC with Dynamic Task Scheduling and Increased Data Locality for Multiple Applications.* Proceedings of the 2010 International Conference on Embedded Computer Systems (SAMOS), pp. 110–117. IEEE, July 2010

[31] Torsten Limberg, Markus Winter, et al.: *A Fully Programmable 40 GOPS SDR Single Chip Baseband for LTE/WiMAX Terminals.* Proceedings of the 34th European Solid-State Circuits Conference (ESSCIRC), pp. 466–469. IEEE, September 2008

[32] *NVIDIA's Next Generation CUDA Compute Architecture: Fermi.* NVIDIA Corporation, Edition 1.1, 2009

[33] *AV Foundation Programming Guide.* Mac Developer Library. Apple Inc., October 2011. From `developer.apple.com` as of July 2012

[34] *clang: a C language family frontend for LLVM.* The LLVM Compiler Infrastructure. `clang.llvm.org`

[35] *libdispatch.* `libdispatch.macosforge.org`

[36] *OpenGCD: Portable implementation of Grand Central Dispatch*

```
1  #include <stdio.h>
2  int main(void) {
3    int multiplier = 7;
4    void (^multiplication)() = ^{
5      int result = 6 * multiplier;
6      printf("%d", result);
7    };
8    multiplier++;
9    multiplication();
10   // prints 42
11   return 0;
12 }
```

Figure 2.5: GCD Blocks Capture State

In lines four to seven, a block is defined[37] and assigned to the local variable `multiplication`. The block multiplies a variable by six and prints the result. The `multiplier` variable is not part of the block's local scope, but is imported from the surrounding scope. The value of this variable is captured at definition time of the block. When the block is finally executed in line nine, it still uses that captured value, even though the multiplier has been changed in line eight. The result printed in line nine is therefore 42.

[37] Refer to Apple's documentation *Blocks Programming Topics* for an in-depth explanation of the blocks syntax.

Figure 2.6 shows that blocks can be passed to functions as arguments and they can be defined anonymously right in the function call, omitting the assignment to a local block variable:

```
1  #include <stdio.h>
2  static void run(void (^block)()) {
3    block();
4  }
5  int main(void) {
6    int multiplier = 7;
7    run(^{
8      int result = 6 * multiplier;
9      printf("%d", result);
10   });
11   // prints 42
12   return 0;
13 }
```

Figure 2.6: Anonymous GCD Block Passed to Function

Again, when the block is defined in lines seven to ten, it captures the variable `multiplier` from the surrounding scope. The block is passed as an argument to the `run()` function, which invokes it in line three, where it prints 42 as the result. In this example, the block is executed immediately, but `run()` is also free to store the block for later invocation, in which case the original `multiplier` variable may already be destroyed, leaving only the captured copy in the block.

Here, libdispatch enters the stage. It implements dispatch queues to which a programmer can submit asynchronous work items in the form of blocks. GCD conveniently provides synchronization primitives as a natural part of its programming idioms.[38] Dispatch queues come in two flavors: serial and parallel queues. Blocks submitted to a serial queue execute one after the other, never running two of them at the same

[38] *Concurrency Programming Guide.* Mac Developer Library, Chapter Eliminating Lock-Based Code. Apple Inc., July 2012. From `developer.apple.com` as of August 2012

time. This guarantee not only implies protection for critical sections, it also allows developers to reason about the order of block execution. Blocks on a parallel queue execute concurrently and may complete in any order. Figure 2.7 restates our running example using libdispatch functions:

```
1  #include <dispatch/dispatch.h>
2  #include <stdio.h>
3  int main(void) {
4    dispatch_queue_t queue = dispatch_get_global_queue(0, 0);
5    int multiplier = 7;
6    dispatch_async(queue, ^{
7      int result = 6 * multiplier;
8      printf("%d", result);
9    });
10   // prints 42
11   dispatch_barrier_sync(queue, ^{});
12   return 0;
13 }
```

Figure 2.7: Asynchronous Execution with libdispatch

Line four obtains a reference to the default global dispatch queue. The block is passed in line six to a libdispatch function, which enqueues it for asynchronous execution. Figure 2.8 pictures the libdispatch runtime automatically spawning a worker thread behind the scenes to process the block from the queue. Line eleven waits for background work to finish before the program exits. Of course the main thread could perform other duties before waiting, resulting in two-way parallel execution.

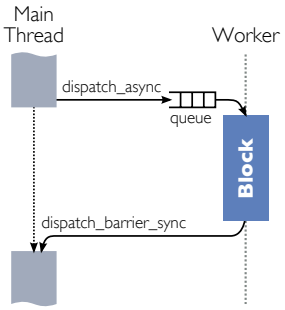


Figure 2.8: GCD Asynchronous Dispatch

GCD CAN PARALLELIZE LOOPS by dispatching all iterations at once with the `dispatch_apply()` convenience function. The dispatch queue will suddenly contain a number of blocks, so the runtime library starts multiple threads to process iterations concurrently, leading to OpenMP-style fork-join parallelism.

But GCD also allows dispatching additional blocks at any time during the iteration, enabling dynamic parallelism like the Galois set iterator. In fact, while the set iterator is restricted to adding more runs of the same iteration, GCD can also dispatch different blocks of code. It therefore generalizes the Galois single-program multiple-data concept to a multiple-program multiple-data system.

THE PROGRAMMING PARADIGM of asynchronous lambdas relieves the developer from explicitly managing parallelism with threads. Instead, the code expresses latent parallelism: The developer states explicitly which code pieces can potentially run in parallel. The runtime library later decides dynamically, which pieces do run concurrently. It translates the latent parallelism to actual parallelism.

An Asynchronous World

If developers follow the rules and ensure responsiveness by keeping all non-trivial work off the main thread, the bulk of execution will be funneled through dispatch queues. Especially for autonomous computing tasks, where the user triggers a long-running operation with a single mouse click, the queues can fill with substantial amounts of work. At the time of the click, no further input from the user or the environment influences the result of the computation, so only the application's code and data determine the following execution. Therefore, when the user triggers the operation, the application can inspect its internal state and expose its knowledge about the upcoming computation to the CPU scheduler ahead of time.

Media playback is an example for such an operation that is one-shot triggered, but then runs autonomously for an extended period of time. As the user clicks play and sits back, all the following calculations are predetermined and known to the application. This also holds true for interactive applications such as games, albeit with a much smaller time horizon. In an application, which primarily reacts to user input with short latency, the dispatch queues will only contain short bursts of work. The application can still foresee and report to the scheduler what it is going to do, but the time horizon shortens compared to media playback use cases.

In the next chapters I demonstrate the benefits of reporting future application behavior to the scheduler early: it can anticipate deadline misses before they occur. I also evaluate the different time horizons of applications. Here, I want to emphasize that such ahead-of-time knowledge and introspection capabilities are a natural consequence of the programming style of modern applications:

Blocks expose static program structure to the compiler to enable automatic state capturing. Dispatch queues expose dynamic program structure to the runtime library to enable automatic thread management. Asynchronous execution by definition announces future computation before it begins. As shown above, the asynchronous lambda style is gaining traction. Thus, modern applications increasingly have knowledge about their future execution behavior at hand. My goal is to expose future program behavior to the CPU scheduler ahead of time to improve global management of timeliness and overload.

Blocks package execution into discrete portions, introducing boundaries the runtime library can recognize and act upon. Individual block instances can be amended with metadata, either manually by the developer or automatically by the system through machine learning. As I show in the next chapters, metadata can convey timing requirements in the form of deadlines and resource requirements in the form of execution times. When queueing such augmented blocks and exposing the resulting structure, applications form a precedence graph of real-time jobs.[39]

Aggregating multiple blocks helps, when a single block is too small to warrant the overhead of metadata handling. GCD already features dispatch groups, which collect multiple blocks and track their joint completion. To delineate these notions, I use the following terminology:

[39] Jane W. S. Liu: *Real-Time Systems.* Prentice Hall, Edition 1, April 2000

Block or lambda designates a combined portion of code and captured variable state that is submitted to a dispatch queue. Block is the name for the GCD implementation, but I use the term lambda synonymously. The runtime library manages blocks within the application. By executing independent blocks on different threads, blocks become the UNIT OF CONCURRENCY.

Job denotes a work item described by metadata like a deadline or an execution time requirement. The application exposes jobs and the scheduler manages them globally. The term is similar in meaning to the one from the real-time field, although the latter is typically defined in the context of a task model. A formal definition of my task model follows in the next chapter. Jobs constitute the UNIT OF SCHEDULING.

Both concepts overlap considerably, especially if one job from the scheduler's perspective is implemented by one block in the application. But as soon as jobs are aggregates of a group of blocks, it is useful to have two separate terms. Note that threads — although an important implementation detail — are not part of the essential vocabulary, because developers do not need to reason about them.

TO THE BEST OF MY KNOWLEDGE, the combination and mutual benefits of asynchronous lambdas and real-time jobs have not been explored yet. A different example of connecting real-time and programming concepts was the Comquad project, which introduced timing contracts[40] into component-based development.

I propose to exploit the code's block structure to attach real-time metadata and expose jobs to the scheduler for global management. As illustrated in Figure 2.9, the scheduler allocates CPU time for each job. The application fills those reservations with life by running the associated code blocks.

[40] Hermann Härtig, Steffen Zschaler, et al.: *Enforceable Component-Based Realtime Contracts.* Real-Time Systems, Volume 35 (1): pp. 1–31. Springer, January 2007

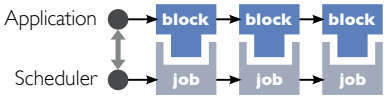


Figure 2.9: Cooperation Between Blocks and Jobs

Inside a Video Player

Studying FFplay — a simple yet fully featured video player — allows us to substantiate the abstract insights discussed above. FFplay is a command line video player, which is part of the FFmpeg[41] project and based on SDL[42] for graphical output and sound. I use version 0.11.1 of FFmpeg,[43] released in June 2012.

I conduct all experiments on a 64-bit Ubuntu Linux 12.04.2 LTS, running on a 2010 15" MacBook Pro[44] equipped with a 2.4 GHz Intel Core i5-520M Arrandale and 4 GiB of 1066 MHz DDR3 SDRAM. This configuration is representative for a mid-range consumer notebook at the time of this writing. Reproducing the experiments is fully automated and all source code and scripts are available in a public source code repository.[45]

[41] *FFmpeg* is a media processing library with comprehensive support for decoding all major video formats.

[42] *Simple DirectMedia Layer* is a library for portable access to audio, video and input devices.

[43] Git commit *39fe8033*

[44] Model identifier: MacBookPro6,2

[45] `https://os.inf.tu-dresden.de/~mroi/git/thesis`

FFPLAY IS DESIGNED according to the traditional multithreading paradigm. At the beginning of playback, it initializes a pipeline with

three stages. A dedicated thread services each stage in a loop, as illustrated in Figure 2.10. This architecture offloads the bulk of work to background threads, keeping the main thread responsive. Thus, it typifies a well-behaving graphical user interface application that performs data processing.

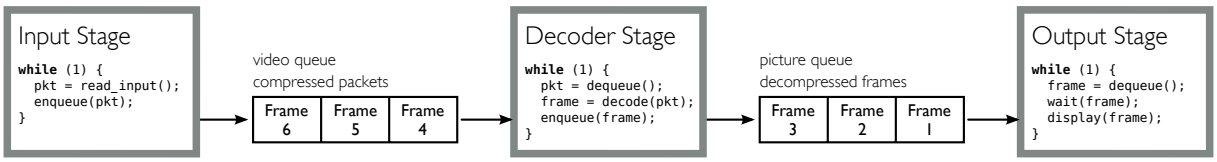


Figure 2.10: FFplay Multithreaded Video Player Design

The input stage reads the video file from disk, parses the stream and splits it into packets of compressed data, with one packet holding the data for one resulting frame of video. This stage is input-bound and requires little CPU time. When the input stage becomes delayed, buffering within the player helps the following stages to continue working.

The decoder stage receives individual packets of compressed video data and decodes them to obtain the finished frames. This process is compute-intensive, especially for high resolution video streams. The computing time needed to decode one frame is dynamic and changes as the video progresses. A buffer to the final stage can compensate timing fluctuations, but when the system experiences high competing CPU load, catching up after a delay is difficult.

The output stage triggers the display of decoded video frames at their proper presentation time. This stage interacts with physical reality, because it determines when the user sees which frame. Therefore, this stage has strict timing requirements, but does not generate much CPU load. To ensure portability, the SDL documentation discourages calling graphics functions from background threads.[46] Therefore, execution of the output stage alternates between the dedicated per-stage thread and the main thread.

[46] *Can I Call SDL Video Functions from Multiple Threads?* SDL Documentation Wiki. SDL Project, April 2008. From `libsdl.org` as of September 2012

Two more threads deal with audio and subtitle decoding, but are ignored here because they do not add interesting new properties to the mix. The main thread runs the event handling loop responsible for reacting to user input and executing display updates. It forms the backbone of the application by giving the user control of the player and coordinating the pipeline to achieve the desired result. To always remain responsive, this thread never executes a long-running or blocking operation.

Data travels through queues, which help even out the varying execution time of decoding video frames and mitigate temporary interruptions in the threads' operation. The queues connect the stages and allow the threads to synchronize: The input stage acquires a free slot on the video queue and fills it with data it has read from the video file. The decoder stage consumes one packet from the video queue and fills

an empty slot on the picture queue with a finished frame. The output stage removes a frame from the picture queue and waits until it needs to be displayed. Unlike the other stages, the output stage therefore self-suspends by sleeping. When the output stage is momentarily suspended and the queues fill up, the preceding stages will eventually block, when they do not find a free slot on their outgoing queues. The queues represent a dependency between the threads.

EACH ITERATION OF A STAGE REPRESENTS A REAL-TIME JOB, for which we can calculate property measures. Figure 2.11 illustrates the necessary terminology. An individual iteration of a stage starts by consuming an item from its incoming queue. This item originated from the previous stage, which inserted it into the queue earlier. We call this insertion point the SUBMISSION TIME for this job. When the item has made it to the front of the queue, the beginning of the iteration marks the RELEASE TIME of the job. When the iteration ends, the job COMPLETES.

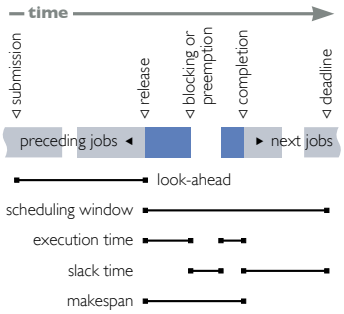


Figure 2.11: Terminology for the Course of Events Concerning a Job

Classical real-time terminology[47] uses a slightly different meaning for the term release time. Here, it is a specific point in the flow of the running code. A job is released when execution reaches the next iteration. In the real-time sense, the release time is part of a formal task model and is a constraint or an assumption for the scheduler to consider.

Additionally, real-time literature often uses the terms arrival time and release time synonymously. But originating from queueing theory, arrival time intuitively refers to the submission instant, because at this time the job arrives at the queue. To prevent confusion, I avoid the term arrival time altogether.

While the submission, release and completion times are names for important points in the actual code execution, a job's DEADLINE is a non-functional target property: The job should complete before its deadline. For FFplay, each data item ultimately results in a frame. The time when that frame must be ready for display marks the deadline of that data item.

After defining important events during the life of a job, we also assign names to interesting time intervals: The time between submission and release of a job is the LOOK-AHEAD TIME, the time between release and completion is the job's MAKESPAN. Preemption or blocking may occur during the makespan, resulting in an EXECUTION TIME less than the makespan. The interval between release and deadline is the SCHEDULING WINDOW and the difference between the scheduling window and the execution time is SLACK TIME. Except for the look-ahead term, these definitions are in line with common real-time terminology.

TO CONFIRM THE DIFFERENT CHARACTERISTICS of the three threads involved in executing the stages, Table 2.2 contains an evaluation of FFplay showing the Hunger Games video[48] with a resolution of 1920×816 pixels. The CPU load column shows the share of CPU time spent in that stage. Because FFplay idles for a portion of time, the loads

[47] Jane W. S. Liu: *Real-Time Systems*. Prentice Hall, Edition 1, April 2000

[48] This video is a representative high definition movie trailer. For detailed video properties, see Table 4.1 on page 51.

do not sum up to 100%. The table continues with the median execution time and its interquartile range (IQR) — the difference between the 25% and the 75% quantiles. These values help judge the weight and variability of the stages, while the average slack time indicates how time critical the stage is. A lower value signifies more critical work, because any delay greater than the slack time will lead to a deadline miss.

Stage	Behavior	Load	Median	IQR	Slack
input	input-bound	0.38%	0.12 ms	0.057 ms	195 ms
decoder	CPU-intensive	28.7%	11.9 ms	3.752 ms	64.1 ms
output	self-suspending	0.15%	0.04 ms	0.014 ms	41.4 ms

Table 2.2: Characteristics of FFplay Stages

The results emphasize the different properties of the three stages: Input exhibits low execution times because it merely shuttles data from disk to the queue. Decoding is compute-intensive and the execution time is varying considerably, as indicated by the interquartile range. The decode and output stages experience low slack times, illustrating their tight timing requirements.

These differences in behavior and the interdependencies between the stages are only incidentally exposed to the CPU scheduler. The preceding measurement of thread characteristics require modifications to FFplay. Intimate knowledge about its inner workings guided the placement of custom taps in the player, which record the necessary job data. The scheduler otherwise only sees a bunch of threads that occasionally sleep or block, followed by smaller or larger bursts of CPU activity. No notion of individual stage iterations, no dependency information of a thread waiting for the work of another, and no knowledge of timing requirements progresses down to the scheduler.

A key concept of Atlas is the explicit handover of such data to the scheduler. However, the solution should not be specific to video players in general or FFplay in particular. Instead, I modify FFplay to organize its work using the asynchronous lambdas programming style outlined earlier in this chapter. The integration between application and scheduler can then be implemented generically in the runtime library managing the asynchronous execution of the lambdas.

Processing pipelines such as FFplay's engine lend themselves well to the queue-based dispatching of work in the asynchronous lambda style. Figure 2.12 on the next page outlines how I adapted FFplay's loop-based implementations of the player stages to block-based execution using Grand Central Dispatch. FFplay's main logic resides in a single file called `ffplay.c` and I needed to modify or add 75 source lines to adapt the programming style, mostly converting thread management to queue handling and restructuring stage initialization code. This change constitutes less than 5% of the 2100 line file. Additionally, the tens of thousands of source lines from FFmpeg libraries which FFplay

```
void input_stage() {
  initialize_input();
  while (1} {
    pkt = read_input();
    enqueue(video_queue, pkt);
  }
}

void decoder_stage() {
  initialize_decoder();
  while (1) {
    pkt = dequeue(video_queue);
    frame = decode(pkt);
    enqueue(pict_queue, frame);
  }
}

void output_stage() {
  initialize_output();
  while (1) {
    frame = dequeue(pict_queue);
    time = display_time(frame);
    wait_until(time);
    display(frame);
  }
}

void start_playback() {
  thread_create(input_stage);
  thread_create(decoder_stage);
  thread_create(output_stage);
}
```

loop-based implementation

```
void input_stage() {
  pkt = read_input();
  enqueue(video_queue, pkt);
  dispatch_async(^{ decoder_stage(); });
  dispatch_async(^{ input_stage(); });
}

void decoder_stage() {
  pkt = dequeue(video_queue);
  frame = decode(pkt);
  enqueue(pict_queue, frame);
  dispatch_async(^{ output_stage(); });
}

void output_stage() {
  frame = dequeue(pict_queue);
  time = display_time(frame);
  wait_until(time);
  display(frame);
}

void start_playback() {
  initialize_input();
  initialize_decoder();
  initialize_output();
  dispatch_async(^{ input_stage(); });
}
```

block-based implementation

Figure 2.12: Changing FFplay to Asynchronous Lambdas

uses for video decoding remain unchanged, making this a small change overall.

DEVELOPERS ARE NATURALLY RELUCTANT to rewriting applications and would rather spend their time adding new features. However, in today's world of scaling down to mobile devices on one end and scaling out to massive parallelism and the cloud on the other, applications are already under constant pressure to evolve.[49] This chapter has motivated the move to asynchronous lambdas as a recent trend, because they simplify development of parallel and responsive code.

ATLAS strives to simplify development of real-time code by using the same programming model. To that end, it exploits the program structure exposed by lambdas to provide information to the scheduler. The next chapter explains the real-time task model and how the interaction between blocks in the application and jobs in the scheduler is orchestrated. The scheduler interface expects assistance from applications, but delivers non-functional timeliness properties in return.

[49] Herb Sutter: *Welcome to the Parallel Jungle!* Dr. Dobb's Journal. UBM, January 2012

Real Simple Real-Time

3

Many of today's applications have intrinsic real-time characteristics. The term real-time describes a predictable relation between observable system behavior and physical time.[1] Therefore, such requirements naturally surface when computers interact with the real world, which most application do through their user interface. In this work, I target interactive systems, so examples operating along this cyber-physical boundary are:

[1] Hermann Kopetz: *Real Time Systems: Design Principles for Distributed Embedded Applications.* Springer, Edition 2, April 2011

Timely delivery of system output, such as

- speedy updates of user interface elements,
- smooth user interface animations, and
- uninterrupted video and sound playback.

Timely reaction to human input, such as

- classical keyboard and mouse interaction,
- gesture tracking with touchscreen or video camera, and
- speech recognition.

I evaluate Atlas with user interaction, video playback, and gesture tracking workloads.
I contribute a task model and a layered scheduler interface which allows applications to express timing and resource requirements. The developer-facing part of the interface only asks for parameters from the application domain, which simplifies programming. The scheduler-facing part provides look-ahead information on future application behavior. A runtime library employs machine learning to mediate between the two layers.
I claim that Atlas is uniquely able to anticipate deadline misses before they occur.

Within this thesis, I restrict all discussions and experiments to a single core scenario. My evaluation platform is a homogeneous multicore machine, so I always pin the relevant threads to the same core. An outlook toward a multicore extension of Atlas follows in the chapter *The Road Ahead.*

On the following pages, I will first motivate the need for real-time programming by explaining pervasive time-critical operations. I continue with insights on existing scheduler interfaces and their shortcomings, before describing the design of Atlas.

The Need for Real-Time

Timing requirements in end-user systems emerge for one of two reasons, illustrated in Figure 3.1

- the system autonomously executes a workload that requires timely output of results or
- the user interacted with the system and expects a timely response.

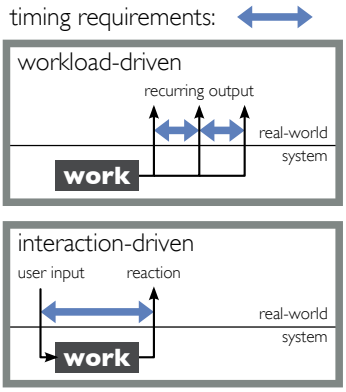


Figure 3.1: Sources of Timing Requirements

CONTINUOUS MEDIA is a real-time application with natural deadlines dictated directly by the processed workload. Video playback exhibits typical frame rates between 24 and 60 frames per second, so frames must be displayed at 16.7 to 41.7 ms intervals. Because users are accustomed to stutter-free movie experiences from cinema and television, providing high-quality video playback on general-purpose computers has been a subject of previous research regarding task models[2], execution time analysis[3] and quality of service.[4]

I designed ATLAS with video in mind and it solves the problem of providing temporal guarantees to high-throughput dynamic applications such as video decoding. However, ATLAS is not a video-specific solution, but also applies to other workloads.

USER INTERFACE CODE is of the interaction-driven type. Timing requirements are driven by usability considerations and follow a 0.1/1/10 second cadence:[5]

Users explore applications by scrolling through data views or operating controls. Visual feedback like updating the scroll view or rendering a button in pressed state should feel immediate, giving the user confidence that he is in charge. Interface changes may involve updating data representations, revealing interface panels, or opening popup-windows. The user perceives those changes as immediate, if the response time stays under 100 ms. This limit becomes even tighter for touchscreen devices, where users operate items on the screen without the indirection of a mouse. Any interface lag will break the user's perception, that the app "sticks to his finger."[6] Animation can accompany the interface state change to support the user's mental transition. Such aids should run timely and smoothly.

Whenever an operation runs longer than one second, the application should show a progress indicator. Otherwise, users start to wonder whether they accidentally missed the button and may click again. After about ten seconds, users get bored waiting and switch to a different task.

VIRTUALLY ALL DESKTOP APPLICATIONS perform user interface activities. But even though developers are aware of the need for responsiveness, timing requirements are hidden deep within the application and are not expressed to the scheduler. Typical frameworks for graphical user interfaces perform all screen updates from the main thread, so

[2] Michael Ditze, Peter Altenbernd: *A Method for Real-Time Scheduling and Admission Control of MPEG-2 Streams.* Proceedings of the 7th Australasian Conference on Parallel and Real-Time Systems (PART). Springer, November 2000

[3] Peter Altenbernd, Lars-Olof Burchard, Friedhelm Stappert: *Worst-Case Execution Times Analysis of MPEG-2 Decoding.* Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS), pp. 73–80. IEEE, June 2000

[4] Damir Isović, Gerhard Fohler: *Quality Aware MPEG-2 Stream Adaptation in Resource Constrained Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS), pp. 23–32. IEEE, June 2004

[5] Stuart K. Card, George G. Robertson, Jock D. Mackinlay: *The Information Visualizer, an Information Workspace.* Proceedings of the 1991 SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 181–186. ACM, April 1991

[6] Jason Olson: *Keeping Apps Fast and Fluid with Asynchrony in the Windows Runtime.* Windows 8 App Developer Blog. Microsoft, March 2012. From `blogs.msdn.com` as of May 2012

all the previously mentioned deadline classes conflate on one thread, impossible to distinguish for the scheduler.

I believe real-time programming can help with these problems and should become the default rather than the rare special case it is today. ATLAS can help pave the way by providing a simple scheduler interface.

Talking to Schedulers

Real-time scheduling is about ordering work so that its execution meets previously negotiated timing constraints. Figure 3.2 illustrates, how order matters with time: Two jobs may come with different deadlines such that one potential order violates the earlier deadline of job B, while another order satisfies both deadlines. The scheduler should recognize this advantage of the second ordering and pick job B first. To make this decision, it needs prior knowledge on both jobs' deadlines and execution times. The design of the scheduler determines how applications communicate their timing and resource requirements to the system and what guarantees they receive in return.

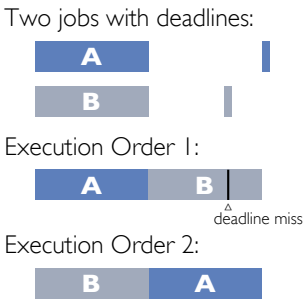


Figure 3.2: Order Matters

EXISTING SOLUTIONS cover a spectrum from strictly regulated to loosely managed platforms. As depicted in Figure 3.3, the strict regulations are harder to program against, but yield strong guarantees, whereas the loosely organized systems trade weaker guarantees for simpler programming. To motivate the ATLAS design, I describe the two extremes of this spectrum and discuss their downsides. A comparison of ATLAS with other points in this solution space follows in a section on related work at the end of the *Timely Service* chapter.

Figure 3.3: Real-Time Solution Spectrum

FAIR CPU SHARING sits at the simplicity end of the spectrum. All of today's commodity operating systems run almost exclusively on this policy. The Completely Fair Scheduler is the default scheduler on Linux[7] and is also used pervasively on Android.[8] The advantage of fair sharing is the lack of a complicated interface or task model. In fact, a developer has to do nothing to get decent behavior for his application. The scheduler distributes the CPU evenly across all ready threads to ensure that all of them make equal progress.

Users however do not care whether their applications receive equal shares, they simply want the right thing to happen at the right time. The fair share model is a good abstraction for performance isolation, but it may be the wrong abstraction for applications with timing requirements. The scheduler disregards timing requirements, because developers have no meaningful way to express them.

An experiment shows this disadvantage: While playing the Hunger Games video[9] in FFplay, a competing load of ten CPU-hogging processes[10] starts one minute into playback. The CPU hoggers simulate a compute-intensive background activity. Linux' default scheduler is used, but all threads are pinned to the same core of the 2.4 GHz Intel Core i5 processor. Due to the scheduler's fairness policy, each thread receives an equal share, squeezing the player down to $\frac{1}{10+1} = 9\%$ of CPU

[7] M. Tim Jones: *Inside the Linux 2.6 Completely Fair Scheduler*. DeveloperWorks. IBM, December 2009

[8] Cláudio Maia, Luís Nogueira, Luís Miguel Pinho: *Evaluating Android OS for Embedded Real-Time Systems.* Proceedings of the 6th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 62–69. Instituto Politécnico do Porto, July 2010

[9] This video is a representative high definition movie trailer. For detailed video properties, see Table 4.1 on page 51.

[10] Their complete source code:
```
int main(void) { while (1); }
```

time.[11] This share is insufficient to decode the video, so frames are delayed. Figure 3.4 plots the time interval between adjacent frames over the runtime of the video, illustrating the user-perceived smoothness of the playback. The grey areas mark the competing background activity. Ideally, the graph should show a flat line at 41.7 ms, corresponding to a rate of 24 frames per second.

[11] Counting FFplay as one thread is reasonable, because only its decoder thread is CPU intensive.

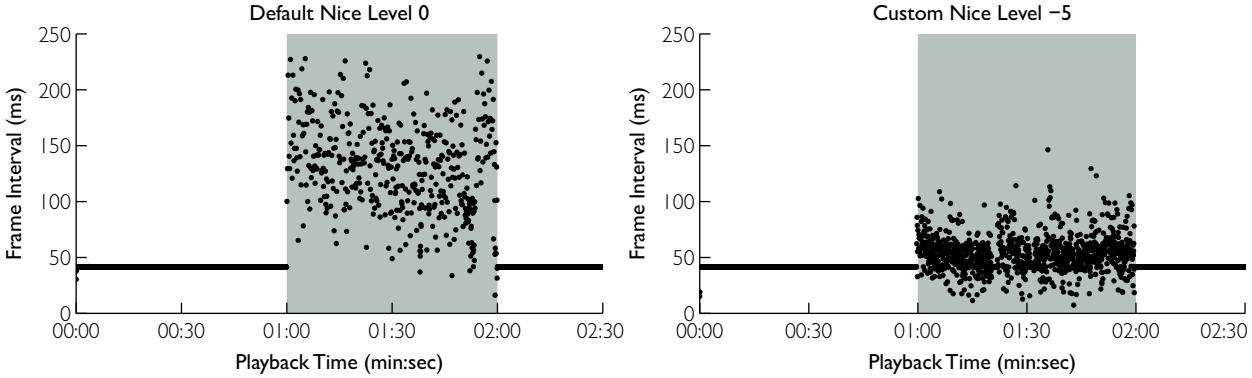


Figure 3.4: Video Smoothness with Competing Background Load

As soon as the background load starts, video quality suffers, because its timing requirements are no longer satisfied. Nice levels[12] alleviate the problem, but cannot fundamentally solve it. A thread with a lower nice level receives a larger CPU share, but the actual time available to each thread depends on the number of competing threads and their nice levels. Figure 3.4 includes an experiment running FFplay at nice level −5.[13] The situation improves, but video quality is still not satisfying. To choose the perfect nice level, the developer of FFplay would need global knowledge of all threads in the system and their execution behavior. Therefore, nice levels are an inadequate interface to specify application timing requirements, because they rely on information an individual application does not have.

[12] `nice` – *Change the Nice Value of a Process.* The Open Group Base Specifications, Volume 7. IEEE, 2008

[13] Only the root user is allowed to run applications with negative nice levels.

For the design of Atlas, we learn that applications want to specify their timing needs in a way that requires no contextual knowledge of the surrounding system. Atlas should meet those timing requirements even at the expense of fairness. The biggest strength of fair sharing — it is simple because it has no interface and it is practical because it exhibits useful default behavior — is also its biggest weakness: There is nothing the developer must do to use it, but at the same time, there is nothing the developer can do if things go wrong.

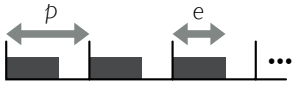


Figure 3.5: Periodic Task Model

The classical periodic task model is diametrically opposed to the fair sharing model in our spectrum of solutions. Developers have to follow a strict programming regime, but the system's runtime behavior will be predictable. Periodic tasks as illustrated in Figure 3.5 describe a chain of jobs with equal execution time *e*. Individual job instances must respect a minimum inter-arrival separation time, more commonly referred to as their period *p*. Each job meets an implicit deadline at the beginning of the next period, so jobs cannot overlap. This serial arrangement is the only inter-job dependency the model

allows. Jobs from different tasks have to be independent, jobs may not block or self-suspend and must be fully preemptible.

Following these rigid rules enables strong guarantees of system behavior. Aggregating all the information and assumptions about the jobs, an admission test can be performed. This test determines — either offline at design time or online while the system runs — whether the task set can be scheduled such that all tasks receive their requested CPU share of e/p and jobs always finish before their deadline, even in worst-case conditions. The task set and resulting schedule are then called FEASIBLE. The Rate-Monotonic Scheduling algorithm[14] provides such a feasibility test for periodic tasks on a single CPU. At runtime, it only needs a fixed priority scheduler to enforce the timeliness guarantees.

The strictness of the periodic task model limits application development. Complex real-time loads in interactive systems may not easily adhere to a minimum inter-arrival distance or a constant execution time of jobs.[15] Varying execution times have to be admitted with their theoretical worst case. For a dynamic workload like video playback, this worst-case planning leads to infeasible CPU reservations way higher than the actual demand.

EXTENSIONS TO THE PERIODIC TASK MODEL as in Quality Rate-Monotonic Scheduling[16] allow to admit tasks with less than their worst-case execution time, resulting in a portion of deadline misses at runtime. Figure 3.6 plots the expected deadline misses for the Rear Window video[17] over the execution time reserved per period. The video plays at 50 frames per second, so the period is a constant $p = 20$ ms. The execution time directly corresponds to a CPU reservation e/p, given in percent. Buffering in the player is disregarded, because the task model disallows overlapping scheduling windows. The right end of the graph shows a worst-case admission, which would result in no deadline misses, but its required CPU share of more than 150% is impossible to accommodate on a single CPU. In fact, all reservations greater than 100% are infeasible, so even dedicating the entire CPU will miss 2% of the deadlines.

At runtime, the player will of course consume much less than the worst-case most of the time, so a slack reclaiming strategy[18] can redistribute the extra time to other threads. However, such slack is not guaranteed and threads with a reservation less than their worst-case may not receive enough time when they need it.

We learn that the periodic task model is inflexible in describing timing requirements and pessimistic in describing resource requirements. ATLAS should offer more flexible reservations than periods with a constant job execution time. Instead, we want ATLAS to accommodate jobs dynamically according to their actual need, thus improving support for highly dynamic applications like video decoding. I show in Chapter 6 how ATLAS successfully schedules playback of the Rear Window video.

The main advantage of the periodic task model is its infinite clairvoyance. The scheduling algorithm can make strong assumptions on the

[14] Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

[15] Veronica Baiceanu, Crispin Cowan, et al.: *Multimedia Applications Require Adaptive CPU Scheduling.* Proceedings of the RTSS Workshop on Resource Allocation Problems in Multimedia Systems Scheduling. IEEE, December 1996

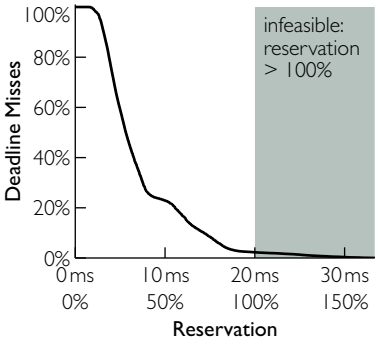


Figure 3.6: Periodic Admission Below Worst-Case

[16] Claude-J. Hamann, Michael Roitzsch, et al.: *Probabilistic Admission Control to Govern Real-Time Systems Under Overload.* Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS), pp. 211–222. IEEE, July 2007

[17] This video is CPU intensive to decode. For detailed video properties, see Table 4.3 on page 60.

[18] Marco Caccamo, Giorgio C. Buttazzo, Deepu C. Thomas: *Efficient Reclaiming in Reservation-Based Real-Time Systems with Variable Execution Times.* IEEE Transactions on Computers, Volume 54 (2): pp. 198–213. IEEE, February 2005

tasks' future behavior and can use them to provide an admission test. Once admitted, jobs are guaranteed to receive their reservations before the deadlines, but with worst-case reservations, only few tasks can be admitted. Developers must take responsibility for adhering to the task model and not violating its assumptions.

Designing Atlas

The discussion of existing scheduling interfaces and their advantages and shortcomings condenses into a set of design constraints for Atlas. We want to find a middle ground between simplicity for programmers and strong real-time guarantees.

1. Atlas shall allow developers to express timing requirements, otherwise we cannot ask the scheduler to respect them.
   Contrast this to fair sharing, which does not convey any timing requirements and thus fails to meet applications' timeliness needs.

2. Atlas shall allow a context-independent specification of resource requirements.
   Fair sharing offers nice levels, which alter a thread's sharing weight relative to the surrounding load. Periodic tasks ask the developer for an execution time, which depends on the target hardware and the user-provided workload. Both thus require information the developer does not have.

3. Atlas shall put timeliness before fairness.
   Fair sharing tries to assign the CPU equally to all ready threads. As shown by the video player experiment, this behavior is not necessarily in the user's best interest.

4. Atlas shall not refuse work.
   Scheduling algorithms for hard deadlines must reject new work when they cannot guarantee real-time performance. Atlas targets firm and soft deadlines and can afford to overload the system if the user demands too much.

5. Atlas' task model and programming interface shall be flexible and expressive.
   Fair sharing offers perfect flexibility, because it does not impose any rules on application behavior. The periodic task model is rigid, but offers infinite clairvoyance of the task's future behavior.

I will now tick off these properties by explaining the architectural building blocks of Atlas.

Deadlines are a natural way to express timing requirements. Developers know about timing constraints, because they are part of the application's problem domain. The same thread may execute real-time jobs with different deadlines, so the scheduler cannot generally guess deadlines by observing the thread. Black-box heuristics may improve unmodified applications,[19] but programmatically exposing deadlines provides added value to the scheduler.

[19] Tommaso Cucinotta, Fabio Checconi, et al.: *Self-Tuning Schedulers for Legacy Real-Time Applications.* Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys), pp. 55–68. ACM, April 2010

Execution times allow the scheduler to allocate time for jobs,[20] calculate a schedule, and anticipate deadline misses. However, execution times are hard to obtain for developers because they depend on the end-user hardware and the workload. ATLAS resolves this impedance mismatch with an indirection: An estimator between the application and the system-level scheduler decouples the interface as seen by the developer from the lower-level interface of the scheduler. Developers provide WORKLOAD METRICS to the estimator, which uses machine learning to infer execution time predictions.

Urgency is the primary criterion for task selection by the scheduler. Fairness is strictly a secondary concern. I postulate that users do not care about fairness as long as all applications meet their timing requirements. In underloaded systems, applications may receive disproportionate CPU shares to meet deadlines.

Overload Detection replaces traditional real-time admission, which uses a priori information to prevent overload. I object to overruling a user decision, so ATLAS will never reject work. However, I demonstrate later in this thesis that ATLAS can detect overload situations before they occur.

Look-Ahead replaces the infinite clairvoyance offered by periodic tasks. ATLAS opts for an explicit submission of future jobs without constraining the inter-arrival distance or demanding non-overlapping scheduling windows. The ATLAS task model is flexible, but compromises on the guarantees it can make.

Specifying deadlines and submitting jobs are an application responsibility, because only the developer has the necessary domain knowledge. Execution times even depend on the workload the application is currently processing, like the video the user chose to play. However, distributing CPU time requires global supervision, because jobs from different applications must be ordered to respect their individual time constraints. Therefore, application components must work with the global scheduler in a unified solution.

[20] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda: *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* Proceedings of the IEEE International Conference on Multimedia Computing and Systems (MCS), pp. 90–99. IEEE, May 1994

End-to-End Integration

Figure 3.7 illustrates the cooperation between the ATLAS layers to provide timeliness properties to applications: Metrics collected from the workload help a machine learning component understand the dependency between processed data and job execution time. Execution time predictions follow from the metrics. The application propagates these predictions together with deadlines to the scheduler before the job runs, offering a limited look into the future. The scheduler orders jobs from all application to provide timely execution. Application and scheduler stay in sync about which job is currently active, allowing the application to associate the right work with each job to ensure timely processing.

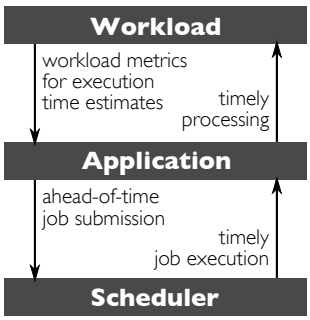


Figure 3.7: Vertical Integration for Timeliness

Throughout the following explanation of the ATLAS architecture and design, I will repeat Figure 3.7 to highlight the described component. With the exception of the asynchronous lambda layer, the architecture was previously published[21] and select text in the task model, interfaces, and prediction sections may have been copied verbatim from this paper.

[21] Michael Roitzsch, Stefan Wächtler, Hermann Härtig: *ATLAS: Look-Ahead Scheduling Using Workload Metrics.* Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–10. IEEE, April 2013

THE ARCHITECTURAL OVERVIEW in Figure 3.8 reflects the conceptual layers just discussed. Applications inspect their workload and talk to a local estimator component, which is linked to the application as a library. This component implements the ATLAS interface as seen by the application developer. It remembers submitted jobs in a local queue until they have run to completion.

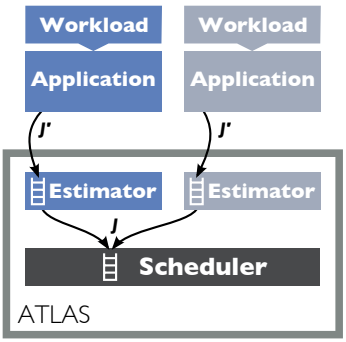


Figure 3.8: Architectural Overview

The per-application estimator also forwards all its jobs to the system-wide scheduler, which queues them for execution. Because I think the estimator interface is simpler, developers should not need to use the scheduler interface directly. However, direct access to the scheduler is possible and not harmful for system security. In the *System Scheduler* chapter I explain how the scheduler can deal with lying applications.

The estimator component queues the jobs solely to remember the workload metrics and the application code associated with the job. ATLAS does not employ hierarchical scheduling.[22] Instead, the passing of job descriptions to the system-wide scheduler enables comprehensive scheduling decisions in one place. The applications then execute according to those decisions.

[22] John Regehr, John A. Stankovic: *HLS: A Framework for Composing Soft Real-Time Schedulers.* Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pp. 3–14. IEEE, December 2001

The two interface layers use two different kinds of job descriptions $\mathcal{J}$ and $\mathcal{J}'$ to represent a job in the system. I first formalize the task model, before I explain the interface operations and their integration into the application runtime.

Task Model and Interfaces

ATLAS runs a finite set $T = \{\tau_i \mid i = 1, \ldots, n\}$ of tasks τ_i. The task set is dynamic, because applications start and stop at the user's discretion. Each task τ_i is a set $\tau_i = \{\mathcal{J}_{i,j} \mid j = 1, \ldots, m\}$ of job descriptions $\mathcal{J}_{i,j}$. While theoretically unbounded, at any given time the number of jobs per task is finite, because we do not use a periodic task model, where an infinite set of jobs is known a priori. In the following, i typically denotes the task number and j the job number within the task. At most one job per task can be running and jobs execute in arrival order. Therefore, tasks are an abstract concept to describe inter-job dependencies.

At the scheduler level, each job is described as a pair $\mathcal{J}_{i,j} = (e_{i,j}, d_{i,j})$ of an estimated execution time $e_{i,j}$ and an absolute deadline $d_{i,j}$. Applications can release jobs whenever they want, no minimum inter-release time and consequently no maximum time demand can be assumed. This design decision is in line with the goal of a flexible task model at the expense of a formal feasibility analysis.

While the deadlines $d_{i,j}$ are provided by the application developer directly, the execution times $e_{i,j}$ are not. The application provides its estimator with job descriptions $\mathcal{J}'_{i,j} = (\underline{m}_{i,j}, d_{i,j})$ consisting of a vector $\underline{m}_{i,j}$ of workload metrics and the deadline as seen by the scheduler. From the metrics, the estimator derives approximate execution times. It can provide these estimates before the job executes. Workload metrics are a vector of non-negative real numbers describing the computational weight of the workload. The estimator expects at least a subset of them to correlate positively with the execution time. Details on the metrics and examples for their use follow in the chapter *Execution Time Prediction*.

THE ESTIMATOR collects job descriptions in local job queues and predicts the per-job execution times. It trains itself using the application-provided metrics and a history of measured actual execution times of past jobs. With the predicted execution times, the estimator forwards for each of its jobs $\mathcal{J}'_{i,j}$ a translated job description $\mathcal{J}_{i,j}$ to the scheduler. The estimator exposes the following three interface primitives:

submit(i, $\mathcal{J}'_{i,j}$) As soon as an application learns about a job it needs to perform, it should register the job with ATLAS. The application announces an identifier i of the task this job belongs to and a job description $\mathcal{J}'_{i,j} = (\underline{m}_{i,j}, d_{i,j})$ that includes a deadline $d_{i,j}$ and a vector $\underline{m}_{i,j}$ of workload metrics. When a job is submitted, the estimator uses $\underline{m}_{i,j}$ and its internal training state to predict the job's execution time $e_{i,j}$ and forwards the resulting job $\mathcal{J}_{i,j} = (e_{i,j}, d_{i,j})$ to the scheduler. It keeps the job $\mathcal{J}'_{i,j}$ in its local queue to automatically train the estimation once the actual execution time of the job is known.

Applications can submit jobs long before actual execution begins and should do so as early as possible to allow the scheduler to look into the future. By submitting a job $\mathcal{J}'_{i,j}$ an application promises the following: As soon as the work of all previously submitted jobs $\mathcal{J}'_{i,k<j}$ within task τ_i is finished, the application will, when given CPU time, start working on the submitted job $\mathcal{J}'_{i,j}$. In other words, there is no extra work performed between $\mathcal{J}'_{i,j-1}$ and $\mathcal{J}'_{i,j}$ that the submitted jobs do not cover. This way, the estimator and scheduler are aware of all work the application performs within a task τ_i.

next() Because jobs execute back to back within a task, this primitive notifies the estimator that the current job $\mathcal{J}'_{i,j}$ within task τ_i completed and that the next job $\mathcal{J}'_{i,j+1}$ should start. The identity of the caller invoking *next* implicitly determines the task number i. The estimator is trained with the actual execution time $t_{i,j}$ of the just finished job and the metrics vector $\underline{m}_{i,j}$ stored during *submit*. Then the execution time measurement for $t_{i,j+1}$ starts. The *next* primitive also informs the scheduler of the job switch.

cancel(i, j) Applications can revoke a previously submitted job $\mathcal{J}_{i,j}$ of task τ_i. This operation mainly serves technical purposes: A video player would use *cancel* when the user stops playback midway in the

video. Because it only serves such technicalities, I do not consider job cancelation in this work and list it here only for completeness.

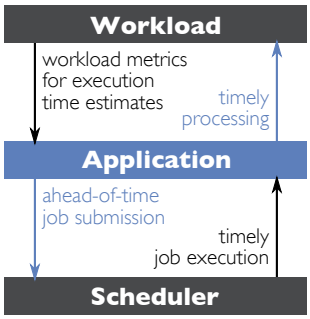
THE CPU SCHEDULER resides in the Linux kernel. ATLAS modifies the kernel's interface to add three new system calls:

atlas_submit notifies the scheduler about a new job. The *submit* primitive employs the *atlas_submit* system call to pass the estimated execution time and the application-provided deadline to the kernel. The kernel also receives the thread identifier of the submission target. This information is necessary, because a thread can submit new jobs for threads other than itself. This is useful for producer-consumer scenarios, where the producer thread will submit jobs for the consumer thread.

atlas_next is used by the *next* primitive to inform the scheduler that the thread is now ready to begin the next job. No parameters are necessary, because *atlas_next* always affects the calling thread. If the scheduler decides to run a different thread instead, the *atlas_next* system call blocks until the scheduler picks the caller again.

atlas_cancel removes a job from the kernel's schedule.

LOOK-AHEAD KNOWLEDGE can build up in the scheduler, when jobs are submitted early before their execution. This course of action aligns with the programming paradigm of asynchronous lambdas, which I described in the previous chapter. This style of organizing applications encourages developers to submit blocks of work to a queueing runtime library for asynchronous execution.



Lambdas with Deadlines

Integration of the ATLAS primitives with an asynchronous lambda runtime helps to reduce the development overhead of adding real-time support to modern desktop applications. Therefore, I implement a subset of the Grand Central Dispatch (GCD) infrastructure with built-in ATLAS support.[23]

The key GCD function call is `dispatch_async(q, b)`.[24] It submits block `b` to a queue `q`. The GCD runtime will execute the block later at its own convenience using an automatically managed background thread. When submitting to a serial queue, only the order of block execution is known. No assumptions about the completion time of blocks can be made. ATLAS integration can change that. I augmented `dispatch_async` with a job description containing three additional arguments:

- the absolute deadline,
- the size of the metrics vector, and
- an array holding the metrics.

[23] Because of its high degree of optimization, the official GCD implementation is complex. Reimplementing a GCD subset was easier than modifying the actual GCD library and suffices to show the viability of my approach.

[24] *Grand Central Dispatch (GCD) Reference.* Mac Developer Library. Apple Inc., November 2011. From `developer.apple.com` as of November 2012

The signature of the resulting new function is:

```
typedef struct {
  double deadline;
  size_t metrics_count;
  const double *metrics;
} atlas_job_t;

void
dispatch_async_atlas(dispatch_queue_t queue,
                     atlas_job_t job,
                     dispatch_block_t block);
```

The block remains the last argument, which improves readability when writing block code inline.

The implementation of `dispatch_async_atlas` enqueues the block in the dispatch queue and calls *submit* to inform the ATLAS estimator about a new job $\mathcal{J}'_{i,j}$ with the given absolute deadline and metrics. The dispatch queue corresponds to the task τ_i. The estimator predicts the execution time for the job and forwards it to the scheduler.

In my prototype, every dispatch queue comes with an associated worker thread. It runs an endless loop, where it first calls the *next* primitive, which can block until the scheduler wants the next job to start. When *next* returns, the worker dequeues the next block and executes it. After the block completes, its measured execution time trains the estimator.

Figure 3.9 summarizes the lifetime of a job from initial submission to final execution. The interplay between asynchronous runtime, estimator, and system-wide scheduler ensures, that the timely release of jobs in the kernel translates to timely processing of the application workload.

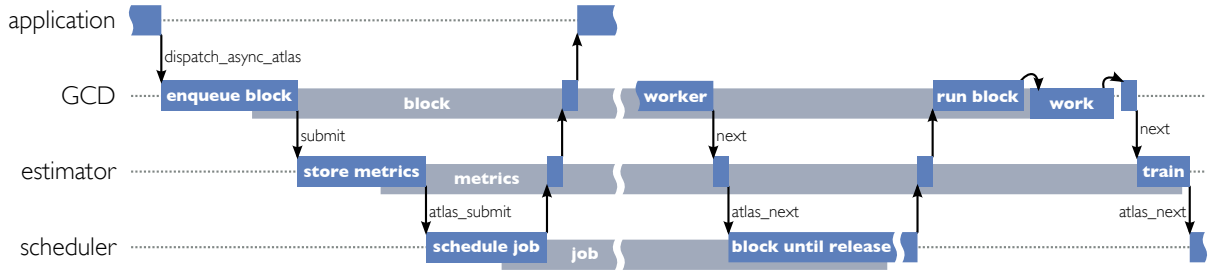


Figure 3.9: A Job's Life Through the Interface Layers

QUEUES AND ESTIMATORS are orthogonal in my implementation, allowing the programmer to submit any block to any queue. The queue is specified explicitly in the first parameter of `dispatch_async_atlas`. The correct execution time estimator is chosen automatically based on the identity of the block.[25] This separation allows developers to freely organize queues and blocks to fit the application.

[25] Blocks internally contain a function pointer to the associated anonymous function. This pointer is used to identify the block

BEING A RESEARCH PROTOTYPE, my ATLAS implementation is subject to the following limitations and assumptions:

- Other than the official GCD library, my reimplementation of the queueing infrastructure only supports independent serial queues. Parallel queues are not implemented, although I indicate an extension in the chapter *The Road Ahead.* The official GCD interface also provides functions to suspend and resume queues or organize them in parent-child relationships. The ATLAS concept does not technically prevent completing the implementation, but the implemented subset is sufficient to demonstrate my approach.
- One worker thread is statically allocated for each dispatch queue. The worker threads could be dynamically stopped when the queues are idle and restarted on demand, but my implementation is simpler and not wasteful, because the worker of an empty queue will not needlessly burn CPU time, but blocks in the kernel when calling *atlas_next*.
- Consecutive submissions to the same queue must have non-decreasing deadlines. This limitation originates from the implementation of the ATLAS kernel scheduler. The kernel sorts jobs by their deadline, which means applications have to match their job management, or the job order will differ. The semantics of serial queues dictate submit-order execution anyway, so this limitation is acceptable.

ATLAS also assumes deadlines and metrics to be available at job submission time. This assumption is not an artifact of my implementation, but is fundamental to the concept. ATLAS needs the metrics at submission time, because the estimator derives an execution time prediction from them. This estimate together with the deadline informs the scheduler about the timing and resource requirements of the job.

Applications, where deadlines or metrics emerge while the job is already running are unsuitable for ATLAS. It may help to split work into multiple jobs, starting with the first job until the deadline is known and then submitting a follow-up job with that deadline. However, this limitation was not impeding my experiments so I presume it does not hinder ATLAS' applicability. A detailed explanation of my test workloads and how I adapted them for ATLAS follows in the next chapters.

IN THIS CHAPTER, I motivated the ATLAS design from the application's point of view. The interface strives for simplicity to foster its adoption as a comprehensive real-time substrate.

Asynchronous lambdas are gaining traction because of their benefits for parallel programming. ATLAS builds on this trend by extending it: With asynchronous lambdas, the application dispatches blocks, which the runtime later executes on an automatic background thread. By passing additional job information with blocks, the ATLAS task model aims for a developer mindset of telling the runtime: "Do this work. Until then. Here's what I know about it." Deadlines specify the time constraint, workload metrics describe the resource requirements. Both are part of the application domain and can be obtained by developers with local reasoning. No global system knowledge is needed.

Next, I explain how applications model their workload and how workload metrics enable the estimator to predict job execution times.

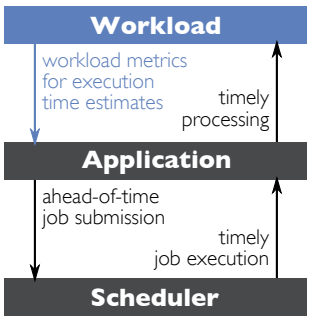
Execution Time Prediction

4

Atlas uses workload metrics from the application domain instead of calling on developers to provide actual execution times. Applications provide the metrics as part of each job description $\mathcal{J}'_{i,j}$ in a vector $\underline{m}_{i,j}$ of real numbers to their estimator. The estimator assumes a subset of metrics to have a positive linear correlation with the job execution time. Badly correlated elements will be filtered out automatically to improve numerical stability.

The developer should choose metrics on a "larger value means more work" basis. Expected iteration counts of loops or the expected number of times an expensive function is called are good candidates. Research within the performance profiling community uses similar workload metrics to characterize computational complexity of algorithms.[1]

Metrics vectors for different instances of the same job must be compatible, that is, the same metric must be delivered in the same vector element. Accordingly, the vectors also deliver the same number of metrics l_i. A linear auto-regressive predictor produces execution time estimates from the metrics. The problem formalization is based on previous work,[2] but the implementation presented here removes the need for an offline training phase. Instead, the mathematical apparatus continuously updates and stabilizes online.



[1] Simon F. Goldsmith, Alex S. Aiken, Daniel S. Wilkerson: *Measuring Empirical Computational Complexity*. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 395–404. ACM, September 2007

[2] Michael Roitzsch, Martin Pohlack: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video*. Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 271–280. IEEE, December 2006

Formalizing the Problem

Atlas maintains an independent estimator state for each kind of GCD block submitted with `dispatch_async_atlas`. However, for clarity, I omit an index discriminating different estimators on the variables below. The index would be identical for all variables and would only clutter the explanation.

The estimator conceptually collects a history of metrics vectors in a metrics matrix M, where each individual vector $\underline{m}_j$ contributes a row of the matrix.[3] All elements are real values. For jobs already executed, the estimator also knows the actual, measured execution times t_j of the jobs and collects them in a column vector $\underline{t}$. To predict execution times of future jobs, the estimator uses this history to obtain a coefficient vector $\underline{x}$ that approximates the execution times when applied to past metrics:

$$M\underline{x} \approx \underline{t}$$

The number l of metrics per job is expected to be small compared to the number of jobs k, so the linear equation has more rows than

[3] Capital letters denote matrices, underlined letters denote vectors.

columns. The equation is therefore overdetermined, so we reformulate it as a linear least squares problem:[4]

$$\|M\underline{x} - \underline{t}\|_2 \to \min_{\underline{x}}$$

Given such coefficients $\underline{x}$ and a metrics vector $\underline{m}$ of a new job description $\mathcal{J}'$, the estimator can calculate the execution time prediction as a dot product of both:

$$e = \underline{m} \cdot \underline{x}$$

This estimated time is forwarded to the system scheduler.

[4] $\|\underline{z}\|_2$ is the Euclidean Norm of $\underline{z}$.

QR DECOMPOSITION is the textbook solution[5] for linear least squares problems. Let M have k rows and l columns, with $k > l$. M is decomposed into an orthogonal real $k \times k$ matrix Q and a real $k \times l$ matrix R, such that $M = QR$ holds. R has the form

$$R = \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} \begin{matrix} \} \, l \\ \} \, k - l \end{matrix}$$

with $\hat{R}$ being a real $l \times l$ upper triangular matrix. Similarly dividing the Q-transformed vector of measured execution times

$$\underline{c} := Q^T \underline{t} = \begin{pmatrix} \underline{\hat{c}} \\ \underline{c_R} \end{pmatrix} \begin{matrix} \} \, l \\ \} \, k - l \end{matrix}$$

and applying the orthogonality of Q yields a solution for the linear least square problem:

$$\left\|M\underline{x} - \underline{t}\right\|_2 = \left\|\hat{R}\underline{x} - \underline{\hat{c}}\right\|_2 + \left\|\underline{c_R}\right\|_2 \to \min_{\underline{x}}$$

$$\text{iff}$$

$$\hat{R}\underline{x} = \underline{\hat{c}}$$

[5] Josef Stör, Roland Bulirsch: *Introduction to Numerical Analysis.* Springer, Edition 3, 2002

This equation is easily solved by back-substitution, because $\hat{R}$ is upper triangular. The resulting $\underline{x}$ is unique, if M has full rank. The residual error of the least squares fit is $\|\underline{c_R}\|_2$.[6] The transformation of M and $\underline{t}$ into R and $\underline{c}$ and thereby into $\hat{R}$ and $\underline{\hat{c}}$ can be unified into one step:

$$R = Q^T M \quad \text{and} \quad \underline{c} = Q^T \underline{t}$$

$$\text{iff}$$

$$(R, \underline{c}) = Q^T (M, \underline{t})$$

[6] $\|\underline{c_R}\|_2^2$ is also called the residual sum of squares.

Therefore, the approach to determine $\underline{x}$ is to merge metrics and measured execution times into a single two-dimensional array and transform it as a whole. The resulting array holds $(\hat{R}, \underline{\hat{c}})$ as illustrated in Figure 4.1. It represents the upper triangular system $\hat{R}\underline{x} = \underline{\hat{c}}$ yielding $\underline{x}$. Note that this system has a constant size of l rows, independent of the number of rows k collected in M and $\underline{t}$.

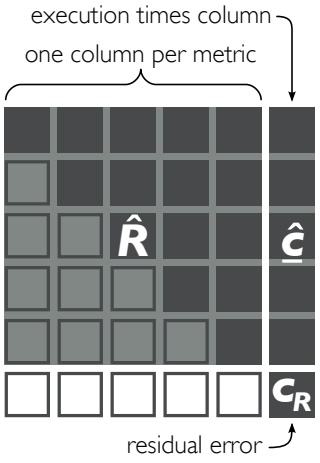


Figure 4.1: Structure of the Linear Least Squares Solution

ADVANCING FROM CONCEPT TO IMPLEMENTATION, we want an estimator that continuously calculates the solution $\underline{x}$ at runtime, without an offline training phase. A naïve implementation would remember

the entire history of metrics and measured execution times and repeatedly solve the complete linear least squares problem to obtain an updated coefficient vector $\underline{x}$. This method is called batch computation. As jobs are executed, new metrics vectors $\underline{m}$ and measured execution times t add more rows to the problem, so the computation required to update $\underline{x}$ would unboundedly increase over time.

Rotate to Fit

Literature on numerical algorithms offers updating solutions,[7] preventing this unbounded growth: Given an upper triangular system $\left(\hat{R}, \underline{\hat{c}}\right)^{(n)}$, which solves a linear least squares problem $\left(M, \underline{t}\right)^{(n)}$, a new job adds a new row to the metrics matrix and the vector of measured execution times. The resulting new problem is $\left(M, \underline{t}\right)^{(n+1)}$. The system $\left(\hat{R}, \underline{\hat{c}}\right)^{(n+1)}$ which solves this new problem can be calculated from the previous solution $\left(\hat{R}, \underline{\hat{c}}\right)^{(n)}$ and the newly added row alone, without the need for the entire $\left(M, \underline{t}\right)^{(n)}$: As shown in Figure 4.2, the new row is appended to $\left(\hat{R}, \underline{\hat{c}}\right)^{(n)}$, shifting the previous values down one row. Givens rotations are applied to zero the subdiagonal elements, turning the matrix into upper triangular form again. The coefficients $\underline{x}$ are then solved by back-substitution in the resulting system $\left(\hat{R}, \underline{\hat{c}}\right)^{(n+1)}$.

[7] Andrzej Kiełbasiński, Hubert Schwetlick: *Numerische lineare Algebra: eine computerorientierte Einführung.* Deutscher Verlag der Wissenschaften, 1988

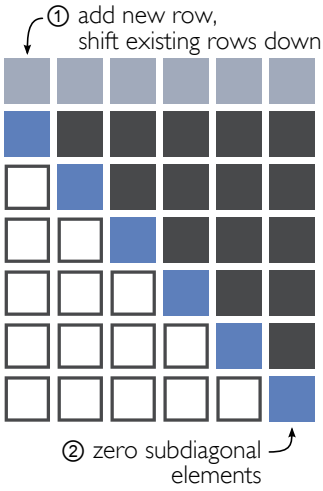


Figure 4.2: Updating the Linear Least Squares Solution

A SINGLE GIVENS ROTATION is a rotation in a plane defined by two standard basis vectors. Regarding the linear least squares problem as a set of column vectors, the solution $\underline{x}$ describes a linear combination of the first l vectors that approximates the execution times vector.

Rotating a matrix can align a selected vector with one of the basis vectors, zeroing one of its coordinates. Consequently, a Givens rotation can zero any matrix element other than diagonal elements. When zeroing the element at row i and column j $(i \neq j)$, only the rows i and j of the matrix change. Because we rotate in the plane defined by the i'th and j'th standard basis vector, all other coordinates remain the same, thus all other rows are unaffected. Figure 4.3 illustrates a single Givens rotation.

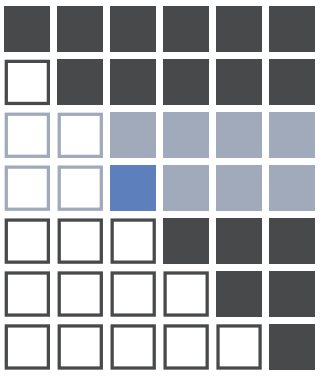


Figure 4.3: Effect of a Single Givens Rotation

Subsequently applying Givens rotations to all subdiagonal elements transforms a matrix to its upper triangular form $\left(\hat{R}, \underline{\hat{c}}\right)$, affording back-substitution calculation of $\underline{x}$. Because the l metrics columns and the rightmost execution times column $\underline{\hat{c}}$ are transformed alike, the rotations are neutral to the linear least squares solution. The scalar in the lower right of $\left(\hat{R}, \underline{\hat{c}}\right)$ contains the residual error as shown in Figure 4.1.

In summary, the estimator starts with an all-zero system $\left(\hat{R}, \underline{\hat{c}}\right)^{(0)}$ and repeatedly adds new rows $\left(\underline{m}_j, t_j\right)$ consisting of application-provided metrics $\underline{m}_j$ and measured execution times t_j to the current system. Applying $l+1$ Givens rotations keeps the system in upper triangular form.[8] With this updating estimator, only the current value of the constant-size system $\left(\hat{R}, \underline{\hat{c}}\right)$ needs to be stored, the ever-growing matrix M and vector $\underline{t}$ need not be stored, because new rows are directly merged into $\left(\hat{R}, \underline{\hat{c}}\right)$. The time complexity is in $\mathcal{O}\left(l^2\right)$, because $l+1$ Givens rotations are applied, each affecting at most $2\left(l+1\right)$ nonzero matrix elements.

[8] John M. Chambers: *Regression Updating.* Journal of the American Statistical Association, Volume 66 (336): pp. 744–748. American Statistical Association, December 1971

Numerical Stability

As I gained practical experience with the described execution time estimator, I observed two numerical problems:

- inflexibility because of a large amount of previously trained values and
- instability because of overfitting to previously trained values.

Both problems are related, because they are instances of the same fundamental conflict: The estimator accumulates knowledge of past job executions and calculates a coefficient vector $\underline{x}$ that approximates this history well. However, our intention is to use the estimator as a predictor in a Model Predictive Control[9] setup to direct scheduling. We want good approximations for future execution times; we actually do not care how well it models the past. Therefore, two independent heuristics dampen the influence of the past to mitigate the two problems mentioned.

[9] Jan M. Maciejowski: *Predictive Control with Constraints.* Prentice Hall, June 2001

THE INCREASING INFLEXIBILITY stems from the estimator always considering all past jobs equally. The solution closes in on a global fit over the entire history and considers new information with decreasing significance. Because the workload metrics never cover the full complexity of the execution time fluctuations, the estimator should maintain a moving average fit, placing more emphasis on recent application behavior than on the distant past.

ATLAS incorporates an AGING FACTOR a in the updating predictor. Possible values range from zero to one, typical values are close to zero. Before adding a new row $(\underline{m}_j, t_j)$ to the predictor, the current system is multiplied with a scalar factor $1 - a$. This factor downscales the complete system and is therefore neutral to the solution.[10] It reduces the influence of the previous knowledge and allows the solution to remain flexible and lean towards the newly acquired row. The scalar multiplication has a time complexity in $\mathcal{O}(l^2)$, so the aging factor approach does not increase the complexity of the update step.

[10] The alternative implementation of emphasizing the new row with a scalar factor $1 + a$ is not stable, because it causes unbounded growth of the matrix elements.

The aging factor is the first tunable parameter of the predictor. It adjusts the relative weight of past knowledge against new information. Because any past behavior of the application may resurface, I argue that the aging effect should be small and only allow for slow adaptation in a long-term moving average style. I use a value of $a = 0.01$ and I experiment with different factors in a video decoding case study below.

OVERFITTING occurs when the linear least squares problem contains redundant metrics. These metrics can show unsystematic correlation with previously trained values and skew the resulting coefficients. Such a solution models past values better than a solution without redundancies, but it generates unstable predictions for future behavior.

An example helps to understand this effect: Two jobs are submitted with two metrics each: $(1, 1)$ for the first job and $(3, 3.00001)$ for the second job. The two metrics correlate, so one of them can be consid-

ered redundant. The measured execution times are 1 for the first job and 3.01 for the second job. Intuitively, the execution time appears to follow both metrics with some small error. We should ignore one of the redundant metrics and use the other for prediction. But mathematically, we solve the following system of equations:

$$\begin{array}{rcrcl} x_1 & + & x_2 & = & 1 \\ 3x_1 & + & 3.00001x_2 & = & 3.01 \end{array}$$

The solution is $x_1 = -999,\ x_2 = 1000$. For a next job with the metrics $(1, 1.01)$, we would intuitively expect a prediction around 1, but the calculated coefficients say 11, an unstable prediction due to overfitting.

I approached this problem before[11] and devised a strategy based on dropping redundant metrics before solving the reduced system. This method however operates offline with full knowledge of the complete linear least squares problem $(M, \underline{t})$. ATLAS' updating predictor requires online stabilization and only has access to the aggregated upper triangular $(\hat{R}, \underline{\hat{c}})$ system. I independently developed a solution, which turned out to be similar to subset regression algorithms from statistics literature.[12]

[11] Michael Roitzsch: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video*, June 2005. Undergraduate Thesis

[12] Michael R. B. Clarke: *Algorithm AS 163: A Givens Algorithm for Moving from One Linear Model to Another Without Going Back to the Data.* Journal of the Royal Statistical Society, Series C – Applied Statistics, Volume 30 (2): pp. 198–203. Wiley, 1981

THE ONLINE METRICS DROPPING ALGORITHM runs in two phases: At first, it scans all metrics to figure out which are significant for the result and which are redundant. It then reorganizes the current $(\hat{R}, \underline{\hat{c}})$ so a reduced result $\underline{\tilde{x}}$ can be calculated, which ignores the redundant metrics and is based only on the STABLE SUBSET.

The scan for redundant metrics successively drops the rightmost metrics column and recalculates the residual error c_R. Because of the nature of a linear least squares solution, the residual error increases with each dropped column. However, a metric that does not contribute significantly will cause only a minor increase of the residual error. The predictor therefore considers a metric redundant, if its contribution increases c_R by less than 10%. This COLUMN CONTRIBUTION THRESHOLD is the second tunable parameter of the predictor. It allows trading prediction accuracy for better stability. I evaluate different thresholds in the video decoding case study below.

Because the rightmost column of $(\hat{R}, \underline{\hat{c}})$ always represents the measured execution times, dropping the rightmost metric means dropping the second column from the right. As Figure 4.4 shows, readjusting the matrix to upper triangular form requires only one Givens rotation, which changes just a single matrix element: the new residual error after dropping the column. The old residual error is zeroed by the Givens rotation. The implementation thus only touches the rightmost column when successively calculating the individual column contributions. Scanning through all metrics in this fashion applies l Givens rotations, each affecting one element. The time complexity of this step is in $\mathcal{O}(l)$.

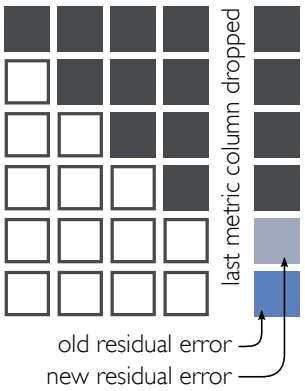


Figure 4.4: Dropping a Column to Determine Error Contribution

Dropping all of the metrics degenerates the matrix to a scalar value equivalent to the standard deviation of the measured execution times, the upper bound for the residual error of the linear least squares solution. The complete scan yields a series of non-decreasing residual

error values as illustrated in Figure 4.5. Metrics that only cause a sub-threshold increase in the residual error — like metrics two and five in the figure — are considered redundant and dropped in the second phase.

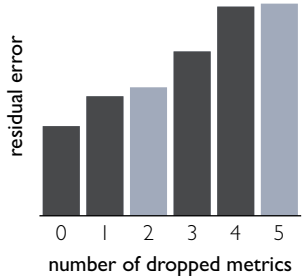


Figure 4.5: Residual Error When Dropping Columns

After the scan step, the predictor drops metrics that contribute below threshold. As we have seen, dropping from the right of the matrix is computationally lightweight. Therefore, the algorithm reorders the matrix by moving all to-be-dropped columns to the right. The column representing the measured execution times must stay in place, so all moves happen within the metrics columns.

An individual move operation pulls a column from its previous position and reinserts it further right. The first such move reinserts the column as the new rightmost metrics column. Any subsequent move reinserts the column left of the previously moved drop-column. After every move, the subdiagonal elements are zeroed with Givens rotations as shown in Figure 4.6, keeping the matrix in upper triangular form. The resulting matrix is not a temporary result, but actually replaces the previous system $\left(\hat{R}, \underline{\hat{c}}\right)^{(n)}$ and becomes the new $\left(\hat{R}, \underline{\hat{c}}\right)^{(n+1)}$. Stabilization is an integral part of every training step. The predictor remembers the column permutation, because new rows that are added to the system and solution coefficients derived from it must be equally permuted.

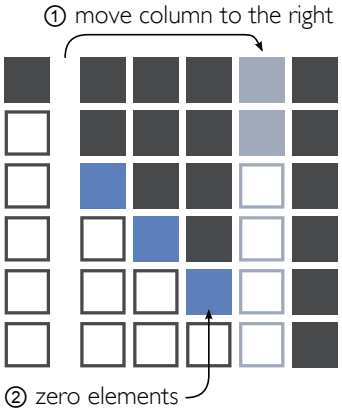


Figure 4.6: Moving a To-Be-Dropped Column to the Right

The worst-case time complexity of this reorganization is in $\mathcal{O}\left(l^3\right)$: Imagine a matrix, where the left half of the metrics columns are to be dropped and the right half are to be kept. We will move $l/2$ columns and for each move apply $l/2$ Givens rotations, each affecting up to $l+1$ elements. However, the state of an individual column hopefully does not oscillate wildly between keeping and dropping. The video decoding case study shows that the column selection settles, which reduces the average case time complexity.

In the new system $\left(\hat{R}, \underline{\hat{c}}\right)$, all drop-columns are grouped together in a single run on the right-hand side of the matrix, followed only by the execution times column. A reduced solution $\underline{\tilde{x}}$ can now be calculated by performing back-substitution in a reduced system, where the dropped columns are removed and a matching number of rows are deleted from the bottom end of the system. Figure 4.7 visualizes such a reduced system. The solution $\underline{\tilde{x}}$ stems from the subset of significant metrics, so it is expected to produce a more stable prediction. The following case study examines and confirms this proposition.

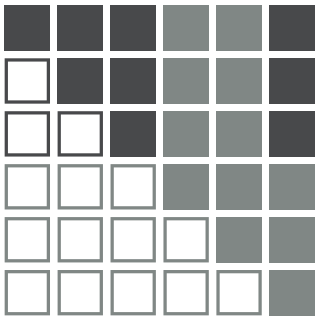


Figure 4.7: Reduced Solution with Three out of Five Metrics

Case Study: Video Decoding

After the mathematical rundown of execution time prediction, I want to take a look at the bigger picture: How would a developer with specific knowledge in his application domain adopt Atlas? I continue using video playback as the example, because it is a highly dynamic workload that is challenging to estimate.[13] The case study provides a better understanding of the workload metrics and the behavior of the estimator.

[13] Figure 1.7 on page 17 illustrates the variability of video decoding times. More examples follow later in this section.

In the following discussion, we assume the role of a developer, who is familiar with the problem domain of the application, in this case video playback. Such knowledge is helpful to formulate the timing requirements and to come up with good candidate metrics for the execution time prediction.

For a video player, the time constraint is rather obvious: frames should complete decoding and display until a time instant dictated by video metadata, commonly a fixed frame rate. Although variable frame-rate video is possible,[14] I limit this case study to constant frame rates.

I assume the video player has already adopted the asynchronous lambda style using GCD. I outlined the necessary changes to FFplay in the preceding chapter *Anatomy of a Desktop Application*. Employing Atlas is easiest when using the interface with the highest abstraction level: the `dispatch_async_atlas` extension to GCD attaches deadlines and workload metrics to code blocks. Should the application structure demand it, direct access to the lower-level primitives of the estimator and scheduler is also possible.

[14] Today's video formats like the MPEG-4 family support individual presentation time stamps (PTS) for each frame. This can improve compression efficiency for certain source material, for example cel-animated content.

Video (Source)	Content	Resolution	FPS	Length	Profile	Encoding
Black Swan (*Apple Trailers*)	movie trailer for "Black Swan" (2010)	848×352	24	126 s	Baseline	x264, CBR, 1500 kbit/s
Shore (*FastVDO*)	flight over a shoreline at dawn	740×576	25	27 s	High	FastVDO
Park Run (*Xiph Test Clip*)	man running in a park with trees, snow and water	1280×720	50	10 s	High	x264, CBR, 4 Mbit/s
Hunger Games (*Apple Trailers*)	movie trailer for "The Hunger Games" (2012)	1920×816	24	155 s	High	x264, CRF, rate factor 23
Charlie (DVB recording)	clip from "Charlie's Angels" (2000)	1920×1088	25	10 s	Main	by TV station

Table 4.1: Properties of Test Videos for Experiments

To demonstrate the metrics selection, I picked the five demo videos in Table 4.1. They cover a typical range of resolutions, given in pixels, frame rates, given as frames per second (FPS), and coding profiles, which declare a set of decoder features required by the video. I experiment with the H.264 video decoding standard,[15] because it is widely deployed and powers the majority of modern media applications from mobile video to HD and 3D Blu-ray discs. Three of the test videos have been encoded by me with the x264 encoder.[16] The Shore video was encoded by the manufacturer of the commercial FastVDO encoder and the Charlie video was recorded directly from the DVB broadcast signal. I ignore audio, because its computational load is two orders of magnitude smaller than video decoding and its timing requirements are less strict, because the audio hardware takes care of sample buffering and timely output.

[15] Thomas Wiegand, Gary J. Sullivan, et al.: *Overview of the H.264/AVC Video Coding Standard*. IEEE Transactions on Circuits and Systems for Video Technology, Volume 13 (7): pp. 560–576. IEEE, July 2003

[16] *x264* is a highly competitive open-source H.264 encoder. At the time of this writing, it has won the previous six installments of the Annual MSU H.264 *Video Codec Comparison*. The x264 version used here is git commit *37be5521* from July 2012.

VIDEO DECODING is the biggest spender of CPU time during playback, as shown in Figure 4.8. Therefore, an accurate prediction of the decoder jobs is vital for informed planning by the scheduler. Figure 4.9 provides an overview of the decoding time per video frame, illustrating the variability within and across the test clips. However, when we run the same video twice, Figure 4.10 demonstrates that decoding times of the two runs are strongly correlated, with a Pearson correlation of 0.998.

We conclude that the high variability of video decoding times does not stem from random influences, but is determined by the input data. This is not surprising given that modern video compression standards like H.264 have a bitstream format so complex we can consider it a domain-specific language to describe video frames. The video decoder is then a special-purpose interpreter for this language whose execution time inherently depends on the video bitstream it is processing.

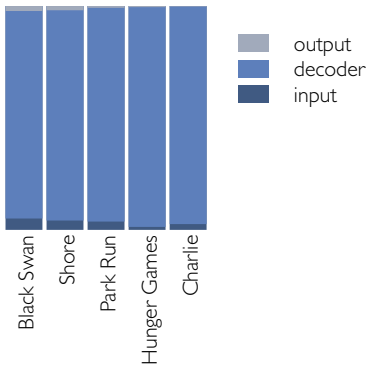


Figure 4.8: Relative Execution Time of the Player Stages

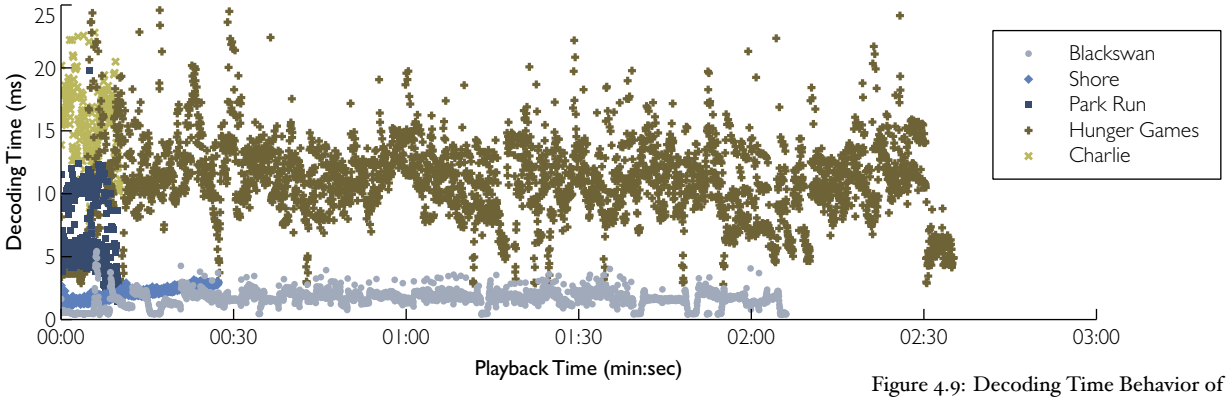


Figure 4.9: Decoding Time Behavior of Test Videos

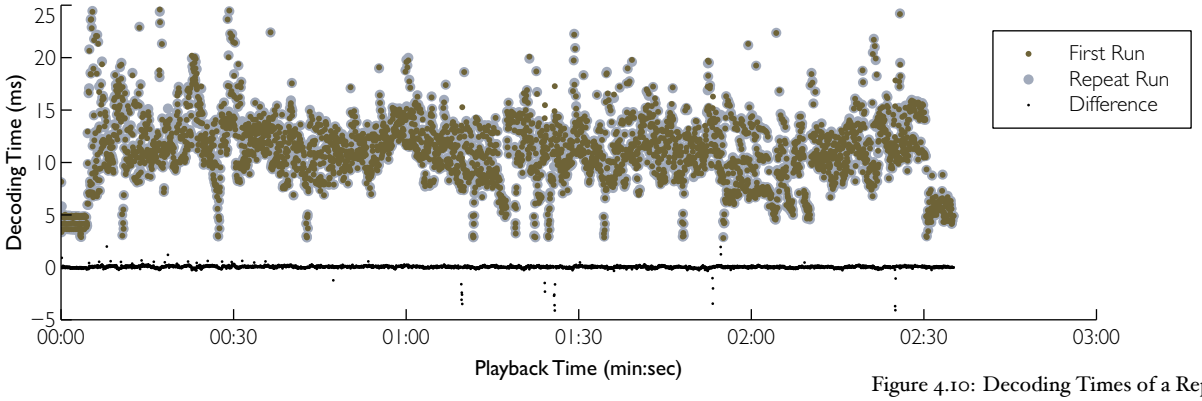


Figure 4.10: Decoding Times of a Repeat Run

As a developer trying to benefit from ATLAS, a key obligation is to find workload metrics that allow the estimator to derive good predictions of the decoding time spent per frame. Intuitively, each metric should grow linearly with part of the computation. For example, if the computation involves a loop, the number of iterations is a good metric candidate. I resort to a manual analysis here, but tool support is conceivable.[17]

In the video case, the metrics should be acquired from the bitstream, because the decoding times strongly depend on the video and because the predictor needs access to the metrics of a frame before decoding begins. Metrics only available after decoding has finished are not helpful for prediction.

[17] Dmitrijs Zaparanuks, Matthias Hauswirth: *Algorithmic Profiling.* Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 67–76. ACM, June 2012

To determine candidate metrics we need inside knowledge of the decoding process. I developed and published an approach for earlier video standards,[18] which I subsequently also applied to H.264.[19] The method divides the decoding work into smaller steps that are easier to examine individually. I summarize the line of thought in the following paragraphs. Text may have been copied verbatim from the two mentioned papers.

First, we need an overview of the decoding steps and how they fit together. A domain expert knows that the building blocks connect as shown in Figure 4.11. The entire chain of components executes once per frame. An inner loop iterates over the macroblocks within a frame. A macroblock is a 16×16 pixel image tile that is stored consecutively in the bitstream. To form the final video frame the decoder arranges the macroblocks in raster scan order and applies post processing.

[18] Michael Roitzsch, Martin Pohlack: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video.* Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 271–280. IEEE, December 2006

[19] Michael Roitzsch: *Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding.* Proceedings of the 7th International Conference on Embedded Software (EMSOFT), pp. 269–278. ACM, October 2007

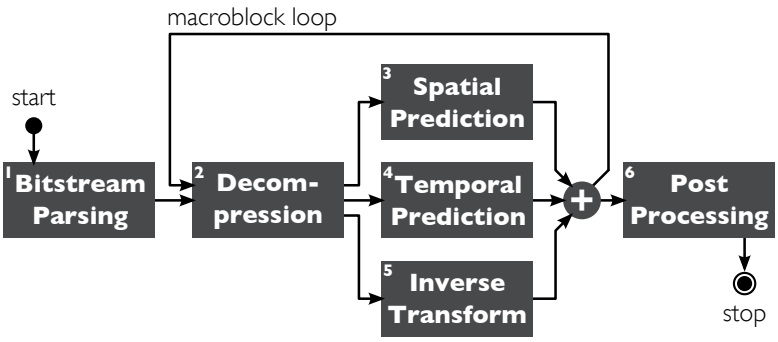


Figure 4.11: Decoding Steps
The numbering matches the description in the text.

Figure 4.12 visualizes the relative weight of the execution times spent in the decoding steps. We observe that decompression and temporal prediction are the two heaviest contributors, followed by post processing. It is therefore most important to find highly correlating metrics for these parts. Interestingly, the Charlie video as encoded by the TV broadcaster does not employ any post processing.

I now visit each decoding step and illustrate how a domain expert selects appropriate workload metrics. A detailed correlation breakdown follows after the discussion.

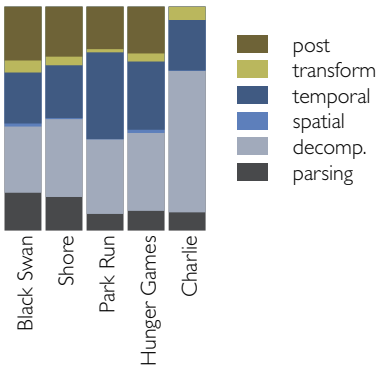


Figure 4.12: Relative Execution Time of the Decoding Steps

1. Bitstream Parsing
   The decoder reads and parses the bitstream representing the next frame and processes header information to prepare the following steps. The parsing effort scans the input data, so it obviously depends on the bitstream length of the compressed frame. In addition, each output pixel is represented in the bitstream and processed by the downstream steps, so the number of pixels in the final frame is another candidate metric. Figure 4.13 shows that a linear fit of the per-frame pixel and bit counts sufficiently matches the execution time of the bitstream parsing step. Remember that this step accounts for a small fraction of total decoding time.

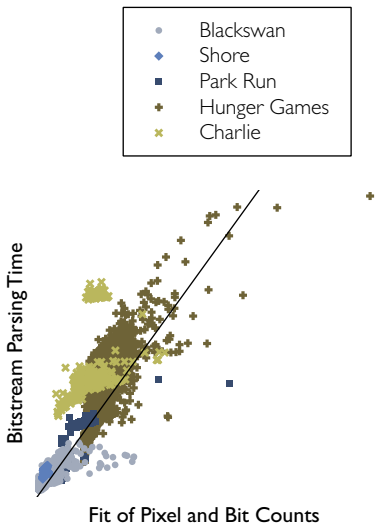


Figure 4.13: Metrics for Bitstream Parsing

2. Decompression
   H.264 offers two different entropy coding schemes: the faster CAVLC and the more efficient CABAC,[20] the latter is only available in the Main and High codec profiles. The decompression step operates on the bitstream to expand the payload and obtain the macroblock data. Thus, the bit count correlates with this step's execution time as Figure 4.14 illustrates. Fortunately the match is visually tight, as this step is responsible for a large share of the overall decoding time.

3. Spatial Prediction
   Both prediction steps try to algorithmically fill the current macroblock by extrapolating from already decoded pixels of either the same frame or a previous frame. The bitstream only stores the residual error between prediction and actual image. Spatial prediction takes patterns from decoded pixels of the same frame and bleeds them into the area of the current macroblock. This process can use sub-block sizes of 4×4, 8×8, or 16×16 pixels, so we separately count occurrences of these block sizes. Figure 4.15 demonstrates that a linear fit of those counts adequately matches the execution time.

4. Temporal Prediction
   This step is the second big contributor to the total decoding time. It extrapolates from previously decoded frames, called reference frames, compensating for any motion by shifting image content along a motion vector. Finding good metrics for this step is difficult, because its execution is exceptionally diverse. Not only can motion compensation operate with square and rectangular sub-blocks of different sizes, each block can also be shifted by a motion vector of full, half or quarter pixel accuracy. In addition, bi-predicted macroblocks target two reference frames. They employ two motion vectors for each sub-block and can apply arbitrary weighting factors to the contributions. The key idea to untangle these options is to consider memory accesses to the reference frames. The filter operations for full, half and quarter pixel vectors read different amounts of reference pixels according to the filter taps in the H.264 standard. Treating bi-predicted blocks as two blocks and rectangular sub-blocks as two smaller square blocks, we infer reference pixel accesses for three sub-block sizes: 4×4, 8×8, and 16×16. Figure 4.16 shows the resulting fit.

5. Inverse Transform
   For spatially or temporally predicted macroblocks, this step compensates the residual error between the prediction and the actual image. For self-contained macroblocks, the bitstream stores the image data directly. To exploit visual redundancy, the bitstream keeps all image data in a DCT-like[21] two-dimensional frequency domain, which this decoding step transforms to the pixel domain. The transform can operate on two sub-block sizes of 4×4 and 8×8. Alternatively, a PCM encoding is available, where the bitstream contains the macroblock

[20] Detlev Marpe, Heiko Schwarz, et al.: *Context-Based Adaptive Binary Arithmetic Coding in JVT/H.26L.* Proceedings of the 2002 International Conference on Image Processing (ICIP), Volume 2, pp. 513–516. IEEE, September 2002

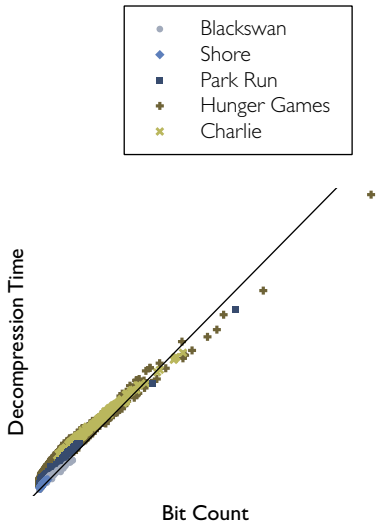


Figure 4.14: Metrics for Decompression

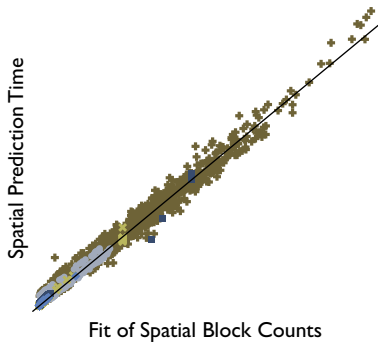


Figure 4.15: Metrics for Spatial Prediction

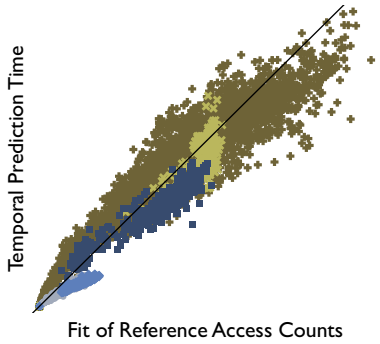


Figure 4.16: Metrics for Temporal Prediction

[21] Nasir Ahmed, T. Natarajan, K. R. Rao: *Discrete Cosine Transfom.* IEEE Transactions on Computers, Volume 23 (1): pp. 90–93. IEEE, January 1974

in a spatial encoding that does not need to be transformed. Figure 4.17 illustrates how the sub-block occurrence counts fit execution time. The remaining imprecision indicates a non-linear component, which we ignore because this step has a small influence on overall decoding time.

6. Post Processing
   The mandatory[22] post processing step tries to reduce block artifacts by selective blurring of macroblock edges. Counting the number of treated edges results in the fit displayed in Figure 4.18.

In summary, we have identified the following set of candidate metrics:

- number of pixels per frame,
- number of CABAC-compressed bits,
- number of CAVLC-compressed bits,
- number of spatial sub-blocks of size 4×4,
- number of spatial sub-blocks of size 8×8,
- number of spatial sub-blocks of size 16×16,
- reference frame accesses for temporal sub-blocks of size 4×4,
- reference frame accesses for temporal sub-blocks of size 8×8,
- reference frame accesses for temporal sub-blocks of size 16×16,
- number of PCM-encoded macroblocks,
- number of block transforms of size 4×4,
- number of block transforms of size 8×8, and
- number of deblocked edges.

To judge the quality of the selected metrics, Table 4.2 analyses the Pearson correlation of the metrics with the measured execution times for each frame over all test videos. The table lists the decoding steps and their relative share of total decoding time. The selected metrics regression calculates a fit of the metrics we identified as reasonable for the respective step. This column numerically presents the correlation we have visually examined in the previous figures. The rightmost column regresses all thirteen metrics against the individual decoding steps and also the total decoding time per frame. As expected, the fit correlates better when more metrics are considered.

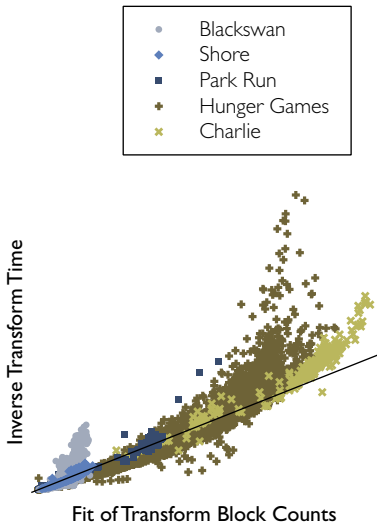


Figure 4.17: Metrics for Inverse Transform

[22] Post processing was optional with previous MPEG-4 coding standards. H.264 uses an in-loop deblocking filter that cannot be skipped.

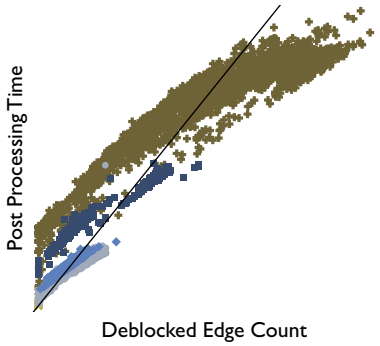


Figure 4.18: Metrics for Post Processing

Step	Relative Execution Time	Selected Metrics Regression	Global Regression
parsing	9.07%	0.920	0.963
decompression	38.1%	0.957	0.997
spatial	1.41%	0.995	0.997
temporal	28%	0.971	0.985
transform	4.15%	0.946	0.994
post	19.3%	0.925	0.996
complete decode	100%	—	0.996

Table 4.2: Pearson Correlation of Metrics Fitted to Per-Frame Execution Times

The strong correlation with the decoding time for complete frames proves that the metrics are well-chosen. However, the regression so far

relies on perfect a-priori knowledge of all metrics and decoding times. The predictor in our video player however has to learn this information while the video is playing.

After analyzing the most time consuming player component in detail, I now explain how these insights practically apply to FFplay.

FFplay Integration

The video playback pipeline consists of three stages: the input stage reads the compressed video from disk, the decoder stage converts it to frames, which the output stage displays. These stages are connected by queues[23] with a limited number of slots where elements wait for processing by the next stage.

[23] These queues contain application data and should not be confused with GCD's work queues.

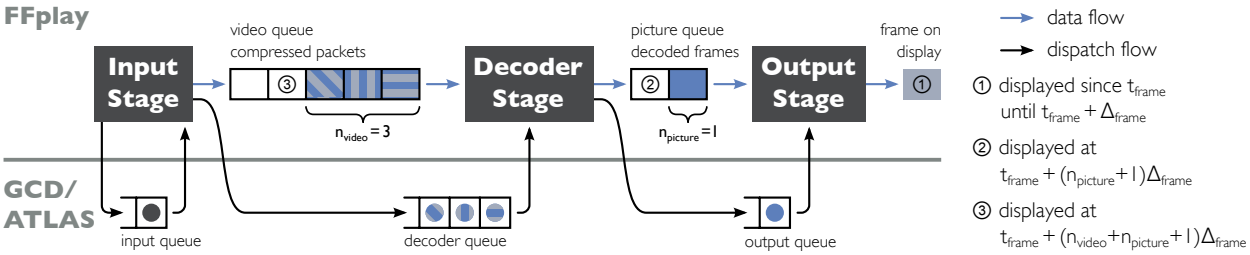


Figure 4.19: FFplay Stage Structure

Figure 4.19 shows one possible queue situation: the current length of the video queue n_{video} is three, the picture queue holds $n_{picture} = 1$ decoded frame. FFplay's default setup uses a five-slot video queue and a two-slot picture queue. The idea of these queues is to compensate for heavyweight frames which need more execution time in a stage than the frame interval allows. We therefore try to keep the queues filled by calculating deadlines accordingly. Remember that all ATLAS deadlines are absolute deadlines.

To model how the pipeline advances, consider a situation where all queues are filled. Whenever the currently displayed frame expires, the next frame to show is taken from the picture queue. This free slot unblocks the decoder which can now enqueue a decoded frame and consume one packet of video bitstream from the video queue, thereby unblocking the input stage.

Our simple model of the pipeline progression therefore rests upon the absolute presentation time t_{frame} of the currently displayed frame and the time interval Δ_{frame} between two consecutive frames. The former is available in the `frame_timer` variable in FFplay. The latter is determined by the video frame rate, which dictates how long each frame shows up on screen. For simplicity, we assume that the queues advance by one frame in every Δ_{frame} interval.

With these prerequisites, I modified the stages of FFplay as follows to enable ATLAS scheduling:

The input stage continuously reads packets off the video file from disk and enqueues them into the video queue. Thus, the input stage must submit new ATLAS jobs for itself to read the next video packet and for the decoder stage to process the packets. The input stage needs

a free slot on the video queue to proceed, so we set $t_{frame} + \Delta_{frame}$ as absolute deadline for the submitted input job.

The video packet associated with the decoder job first travels through the video queue and begins decoding after advancing n_{video} slots. The decoder stage then requires one free slot on the picture queue. Thus, we establish $t_{frame} + (n_{video} + 1)\Delta_{frame}$ as the deadline of the decoder job. However, because we enqueue the packet before submitting the job, the video queue is already extended to $n'_{video} = n_{video} + 1$, so the deadline is actually set to $t_{frame} + n'_{video}\Delta_{frame}$.

Due to their small execution time relative to decoding, no metrics are supplied to the ATLAS estimator for the input jobs. The decoder jobs carry metrics as discussed above, which the input stage obtains by parsing the video bitstream. At the end of this chapter I evaluate design alternatives for the metrics and their extraction and the consequences for prediction accuracy.

Figure 4.20 shows a pseudo-code representation of the input stage with ATLAS modifications on top of the asynchronous lambda version. Remember that `dispatch_async_atlas` sends a new block to the dispatch queue designated by the first argument and also informs the scheduler about this new work by submitting the job given in the second argument.

Figure 4.20: FFplay Input Stage Using the ATLAS Interface

```
void input_stage() {
  /* instruct input stage to read next packet after this one */
  atlas_job_t input_job = {
    .deadline = frame_timer + frame_interval
  };
  dispatch_async_atlas(input_queue, input_job, ^{ input_stage(); });

  packet = read_input();
  enqueue(video_queue, packet);

  /* instruct decoder stage to process the video packet */
  atlas_job_t decoder_job = {
    .deadline = frame_timer + video_queue->length * frame_interval,
    .metrics_count = 13,
    .metrics = extract_metrics(packet)
  };
  dispatch_async_atlas(decoder_queue, decoder_job, ^{ decoder_stage(); });
}
```

The decoder stage consumes compressed packets from the video queue, decodes them and places the finished frames onto the picture queue. A newly enqueued frame has to wait for the currently displayed frame to expire and for the $n_{picture}$ frames preceding it in the picture queue. Thus, its output stage invocation receives a deadline of $t_{frame} + (n_{picture} + 1)\Delta_{frame}$. Again, enqueueing the frame before submitting the job increases the length of the picture queue to $n'_{picture} = n_{picture} + 1$, resulting in an actual deadline of $t_{frame} + n'_{picture}\Delta_{frame}$. Because of the insignificant execution time of the output stage, the estimator receives no metrics. Figure 4.21 lists the resulting pseudo-code:

```
void decoder_stage() {
  packet = dequeue(video_queue);
  frame = decode(packet);
  enqueue(picture_queue, frame);

  /* instruct output stage to display the frame */
  atlas_job_t output_job = {
    .deadline = frame_timer + picture_queue->length * frame_interval
  };
  dispatch_async_atlas(output_queue, output_job, ^{ output_stage(); });
}
```

Figure 4.21: FFplay Decoder Stage Using the ATLAS Interface

The output stage at the end of the video player pipeline finally displays the frames. It does not issue any new work and therefore remains unmodified.

WITH THESE MODIFICATIONS to FFplay, any block submitted to any GCD queue is backed by an ATLAS job and therefore real-time scheduled. On top of the changes discussed in Chapter 2 to enable asynchronous lambdas, the ATLAS adoption touches 33 lines, less than 2% of `ffplay.c`. The modifications are straightforward for a developer familiar with the codebase. The FFmpeg libraries remain unchanged.

Note that the application workload dictates only the final display deadline of the frames. Considering the application structure we manually inferred the intermediate deadlines to make maximum use of the queues, ensuring that later stages operate without being delayed. When I evaluate the overall scheduling behavior of ATLAS, I include results with later intermediate deadlines, allowing the queues to deplete in high load situations. ATLAS provides smooth playback with both deadline placements.

Before evaluating the prediction quality of FFplay execution times, I want to bring up the following implementation details:

- Because SDL limits calls to graphics functions to the main thread,[24] the output stage contains two parts: the first part runs from the GCD output queue on a background thread and manages the timing of frame display, the second part runs on the main thread and puts the image on screen. Therefore, the decoder stage emits two ATLAS jobs, one for each part of the output stage. I ignore this technicality to simplify the evaluation.

- Due to inter-frame referencing, the display order of video frames can be different from their stream order. The decoder stage of FFplay calls a frame decode function from FFmpeg's `libavcodec` library, which handles frame reordering in an internal pipeline. There are pipeline ramp-up and ramp-down effects at the beginning and end of a video, but in its steady state, the decoder emits one decoded frame for each delivered bitstream packet.

- As a consequence of absolute deadlines the application and the kernel scheduler must agree on a common time reference. ATLAS uses

[24] *Can I Call SDL Video Functions from Multiple Threads?* SDL Documentation Wiki. SDL Project, April 2008. From `libsdl.org` as of September 2012

the `CLOCK_MONOTONIC` facility which is available on Linux as part of the `clock_gettime`[25] system call.[26] One second elapsing on this clock always corresponds to one second in physical reality. The clock offers nanosecond granularity and is unaffected by clock adjustments, leap seconds or time zone changes. The starting point is at an unspecified time in the past, so the clock does not bear a meaningful relation to wallclock time. This limitation is unproblematic for the intended use.

- Training the predictor requires measuring the execution time of individual jobs. However, simply subtracting the job's release time from its completion time calculates the makespan instead of the execution time, the difference being interruptions due to blocking or preemption. To measure time spent within a job, `clock_gettime` provides the `CLOCK_THREAD_CPUTIME_ID` flavor. This thread-specific clock only ticks, when the corresponding thread executes. The estimator internally performs time measurements with this clock.
- Thanks to the numerical enhancements presented earlier, the Atlas predictor automatically selects a stable subset from the metrics given to it. Therefore, we can always supply additional metrics without harming the prediction quality. As a convenience to the programmer, the Atlas runtime library always adds a constant metric of one before passing the metrics to the linear solver. This one-metric catches any constant-time contribution to execution time. Especially when the developer passes no metrics, like I did for the input and output stages of FFplay, this behind-the-scenes addition enables the predictor to extrapolate future execution times from previously measured execution times. Otherwise, the predictor would always return zero for jobs with no metrics.
- When an Atlas-enabled application is launched, the predictor always starts off from an empty state with no knowledge about job execution times. Before it has seen the first job executions, it cannot make any predictions and consequently returns zero. After the first jobs passed, the initial predictions may be less accurate because the internal coefficients need to settle. Resuming from a saved predictor state avoids these problems. Applications can save this state on every exit or run a training workload at install time.

From metrics selection to inferring deadlines and modifying FFplay, I hope I conveyed an understanding of integrating Atlas into a complex real-time application. Unlike the classical periodic task model, deadlines can be placed arbitrarily. Unlike priority-based schedulers, no knowledge of surrounding applications and their priorities is needed. Unlike task models based on execution times, Atlas is independent of the user's hardware. The estimator uses the metrics to derive approximate execution times. But to be useful for look-ahead scheduling, we want the estimator to operate predictively. It needs to provide execution times of submitted, but not yet executed jobs before they run. The next section evaluates this ability.

[25] `clock_getres, clock_gettime, clock_settime` – *Clock and Timer Functions.* The Open Group Base Specifications, Volume 7. IEEE, 2008

[26] On 64-bit Linux, `clock_gettime` does not transition into the kernel, but executes in user space thanks to the *kernel's vDSO facility*.

Auto-Training Look-Ahead

Thanks to queueing, FFplay exposes jobs to the ATLAS interface ahead-of-time. For every such submitted job, a predicted execution time e is calculated as the dot product of the job's workload metrics vector $\underline{m}$ and the current coefficient vector $\underline{x}$ from the auto-regressive predictor:

$$e = \underline{m} \cdot \underline{x}$$

For every completed job, the actual execution time is measured and used to train the predictor, updating the coefficient vector $\underline{x}$. The queue therefore introduces a gap between prediction and training illustrated in Figure 4.22.

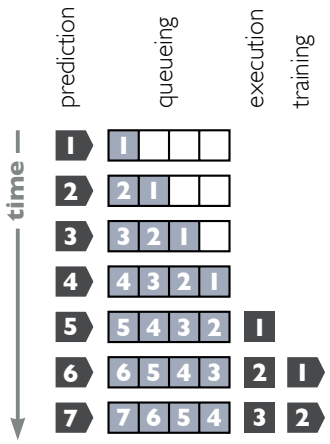


Figure 4.22: Queueing-Gap Between Prediction and Training

In this section, I evaluate the accuracy of the execution time prediction.[27] In addition to the test videos used so far,[28] I supplement select experiments with the short and feature films listed in Table 4.3.

I discuss alternatives for the decoder metrics and also measure the overhead introduced by the predictor. To improve stability I introduced the aging factor and automatic metrics sub-setting based on the column contribution threshold. I substantiate the choice of values for these parameters and demonstrate that the stable subset settles.

[27] The test machine is the same as in preceding experiments: a 2.4 GHz Intel Core i5-520M Arrandale with 4 GiB of 1066 MHz DDR3 SDRAM. To ensure stable and reproducible results, CPU frequency scaling including Intel Turbo Boost (cf. *Intel: Turbo Boost Technology*) is disabled.

[28] see Table 4.1 on page 51

Video (Source)	Content	Resolution	FPS	Length	Profile	Encoding
Rear Window (DVB recording)	segment of feature film "Rear Window" (1954)	1280×720	50	20 min	Main	by TV station
Sintel (*Xiph Archive*)	short film "Sintel" (2010), computer animated	4096×1744	24	15 min	High	x264, CRF, rate factor 23

Table 4.3: Properties of Movies for Experiments

NO METRICS WERE PASSED for the input and output stages of FFplay. For convenience, ATLAS adds a one-metric to enable prediction based on the average of previous execution times. Figure 4.23 shows the mean relative error and its lower and upper quartiles as error bars. Given the small overall execution times of these two stages,[29] those errors can be tolerated. The overall evaluation of scheduling behavior will verify that additional effort to find suitable metrics is not warranted.

[29] recall Figure 4.12 on page 53

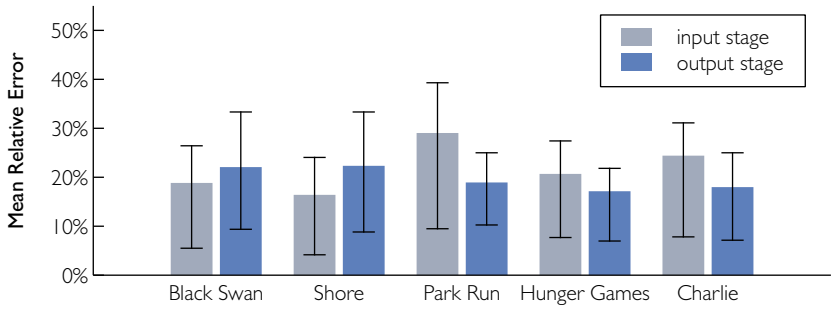


Figure 4.23: Prediction Accuracy for Input and Output Stages

THE DECODING STAGE is the most time consuming part of FFplay and thus calls for a precise prediction of its execution time. To demonstrate the importance of workload knowledge, I compare three alternatives for the decoding time metrics.

The first version uses no metrics and consequently does not exploit workload insights. Thanks to the automatic one-metric, the prediction is based on previous execution times. I call this alternative TIME ONLY and it serves as a lower bound. The other predictions I present should not be worse.

The option I call FULL METRICS uses the complete set of metrics identified previously. We expect this version to provide the most accurate predictions, because it forwards the largest set of valuable workload information to the estimator. Unfortunately, the majority of the metrics hide in the video bitstream behind an outer layer of entropy coding, so extracting them requires performing part of the decoding work.

Two solutions can remedy this problem. A staged decoder[30] first processes the entropy compression layer and then extracts the metrics for the second stage, which performs the remaining decoding steps. This design requires intricate changes to the video decoder, which I do not think developers are prepared to implement.

Instead, I chose an offline preprocessing step, which embeds the video metrics into the bitstream. The MPEG-4 Part 2 video standard[31] describes a complexity estimation header which optionally contains video metadata similar to my metrics. Unfortunately, this header has not been carried over to the H.264 standard. My preprocessing recreates such metadata for H.264 by adding user-defined datagrams. The resulting video is still a standards compliant stream that any unmodified H.264 decoder can play. Preprocessing one video only requires decoding it once and thus is fast. To keep the extra data small, I apply exp-golomb coding,[32] an established compression technology for bitstream header elements. The size overhead from the embedded metrics amounts to a geometric mean of 0.3% over all test clips.

The full metrics with stream preprocessing serve as an upper bound for implementor effort and resulting prediction accuracy. This option demonstrates, how good predictions can get when designers are willing to apply an end-to-end approach and are able to engineer the data format of their application to support a desired non-functional property. But a middle ground between the full metrics and the time-only approach is needed.

This intermediate option is limited to information that can be easily obtained from an unmodified bitstream:

- the number of pixels per frame,
- the number of bits of the compressed frame, and
- the frame type (I, P, or B).

The first two values are used as metrics directly. The frame type is passed to the estimator as three separate binary metrics, each of which can only take the values 0 or 1. The combination $(1, 0, 0)$ signals an I-frame, $(0, 1, 0)$ a P- and $(0, 0, 1)$ a B-frame. I call this alternative REDUCED METRICS.

[30] Peter Altenbernd, Lars-Olof Burchard, Friedhelm Stappert: *Worst-Case Execution Times Analysis of MPEG-2 Decoding.* Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS), pp. 73–80. IEEE, June 2000

[31] *ISO/IEC 14496-2:2004: Information Technology – Coding of Audio-Visual Objects – Part 2: Visual*. ISO, Edition 3, September 2009

[32] Solomon W. Golomb: *Run-Length Encodings.* IEEE Transactions on Information Theory, Volume 12 (3): pp. 399–401. IEEE, July 1966

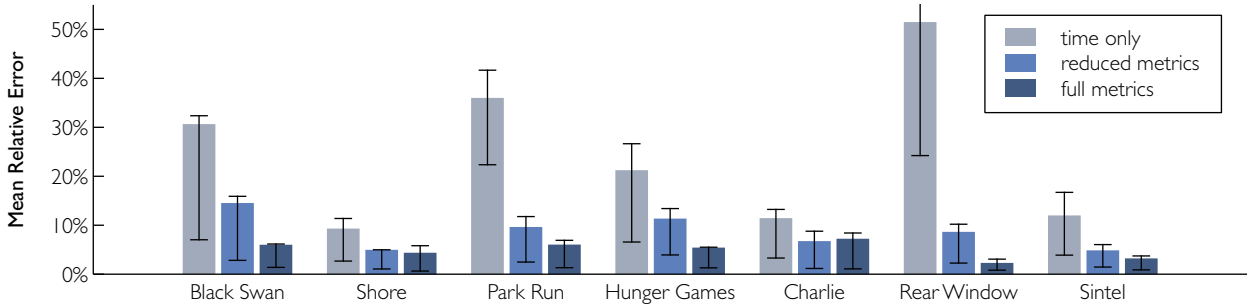


Figure 4.24: Prediction Accuracy for the Decoder Stage

Figure 4.24 shows that workload awareness improves prediction accuracy over a purely time-based predictor. Again, the error bars represent the lower and upper quartile of the per-frame relative prediction errors. The reduced metrics, which do not require preprocessing or a staged decoder provide a good middle ground, whereas the full set of metrics offers the highest precision, with typical relative errors of less than 10%. The convincing results indicate that the prediction provides accurate estimates for short clips and full-length features, regardless of the encoding profile and video resolution being used.

The Shore and Charlie clips exhibit the smallest improvement of metrics-based over time-only prediction. Both videos were encoded by third parties and compared to the x264 material, their bitstreams are rather uniform. Hence, a prediction based on previous execution times alone sufficiently approximates their behavior.

In contrast, the Rear Window video is badly predicted by the time-only option. It was captured from a DVB station broadcasting at 50 frames per second, but the presented movie features only 25 frames per second. Therefore, every other frame is identical to its predecessor, resulting in large decoding time variations between the even- and odd-numbered frames. The time-based predictor blindly averages these jumps, resulting in a high prediction error. The reduced and full metrics improve accuracy significantly.

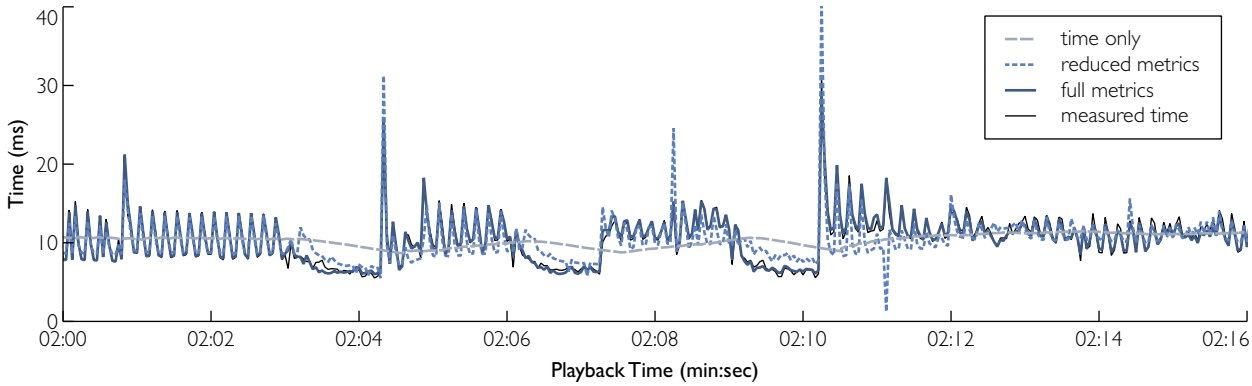


Figure 4.25: Zoomed View of Prediction Alternatives

Figure 4.25 zooms in on a section of the Hunger Games video to illustrate that the full-metrics prediction tightly follows the large variations of the decoding time, while time-only prediction yields a long-term moving average.

Across the test videos, the predictor introduces a runtime overhead of 0.1%, but with an interquartile range of 0.7%, meaning that the significance of the overhead is smaller than the variability of FFplay's execution time measurements.

To show the hardware independence of the resulting preprocessed videos, I reran the prediction experiment on an Intel Atom D2550 with 1.86 GHz. The results in Figure 4.26 are similar to those from the Core i5, with full metrics leading to relative errors of less than 10%. If the target hardware was known and fixed, embedding a decoding time trace into the video would solve the prediction problem. In contrast, workload metrics allow a hardware-independent prediction of execution times.

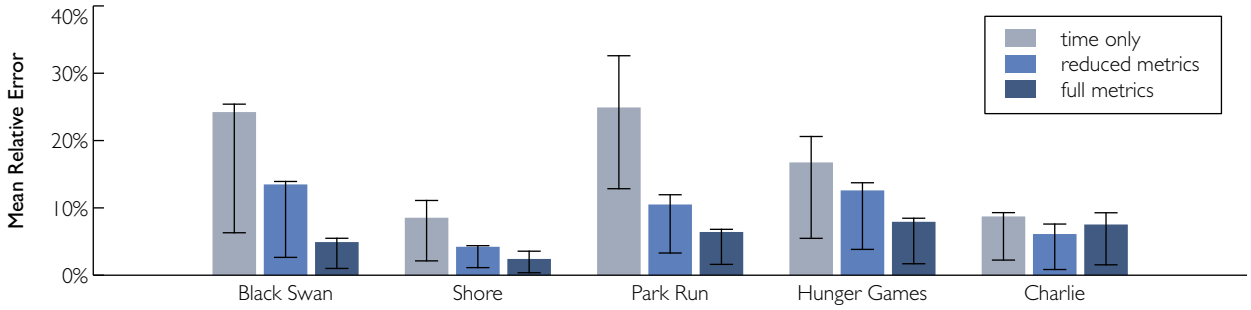


Figure 4.26: Prediction Accuracy on Intel Atom Hardware

Two tunable parameters are part of the predictor design: The aging factor determines, how quickly historical job data should be discarded in favor of newly acquired knowledge. The column contribution threshold trades prediction accuracy against numerical stability. The following experiments analyze the effect of these parameters.

Figure 4.27 demonstrates how the aging factor affects prediction accuracy for the Hunger Games video. The factor determines, how quickly the predictor discards old knowledge and adapts to new application behavior. A larger factor means quicker adaption, while a smaller factor preserves historic data longer.

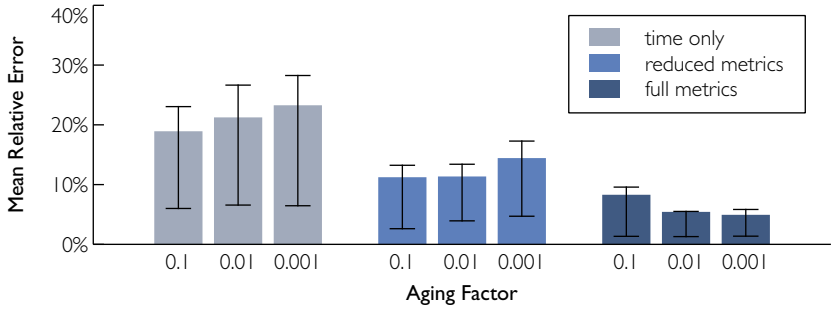


Figure 4.27: Prediction Accuracy Influenced by Different Aging Factors

With the less precise time-only and reduced-metrics options, faster aging improves the prediction. This result is expected, because these metrics rely on the moving average to capture application behavior. This moving average adapts quicker with faster aging. However, for the full-metrics option, quick aging diminishes prediction accuracy. These metrics already cover the complete application behavior and therefore

profit from more knowledge aggregating in the predictor. Because the benefit for the full-metrics prediction levels off at 0.01, I chose this value as the default. It also constitutes an acceptable compromise for the reduced metrics, which are important if videos are not preprocessed. A 0.01 aging factor reduces the weight of past knowledge to 10% after about 230 jobs.[33] Should the need arise, applications are free to override the default.

[33] $\ln 0.1 / \ln 0.99 = 229.105...$

The column contribution threshold trades numerical stability against accuracy. A small value uses more metrics, which should result in more accurate predictions in the long term, but can lead to overshoots due to instability in the short term. A larger value ignores more metrics by dropping their corresponding column from the linear system. Dropping more metrics increases numerical robustness, but reduces long-term precision.

Figure 4.28 illustrates an instability problem at the beginning of the Hunger Games video: Like every movie trailer, it starts with the typical green MPAA rating card. Without column dropping, the predictor overfits to those simple frames and produces intermittent spikes and a series of exaggerated prediction once the actual trailer begins. Note the logarithmic ordinate of the plot. These overshooting predictions are undesirable and the figure also shows, how a column contribution threshold of 1.1 prevents them.

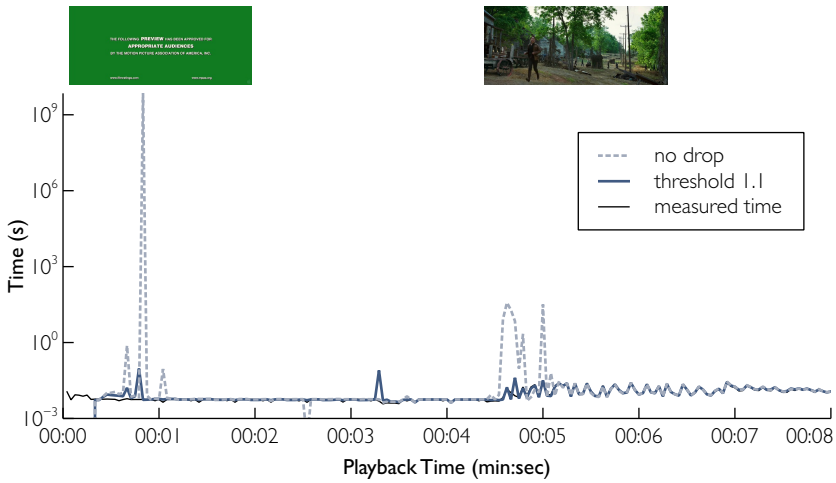


Figure 4.28: Instability Problem at the Start of the Hunger Games Trailer
Note the logarithmic ordinate.

A threshold of 1.1 drops a metric, if it contributes less than 10% improvement to the residual error. Figure 4.29 confirms that higher thresholds lead to less accurate predictions in the long run. The figure shows the median instead of the mean relative error, because the median is robust against the overshooting outlier predictions. Error bars again indicate the upper and lower quartile. A threshold of 1.1 performs close to the no-drop case, but without its instability. A threshold of 1.2 would reduce long-term prediction accuracy, substantiating the choice of 1.1. Again, applications are free to override this default.

The column dropping algorithm comes at the cost of a worst-case time complexity in $\mathcal{O}(l^3)$ for l metrics. However, the predictor reorders

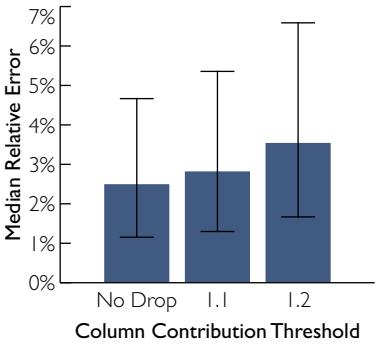


Figure 4.29: Prediction Accuracy for Different Column Contribution Thresholds

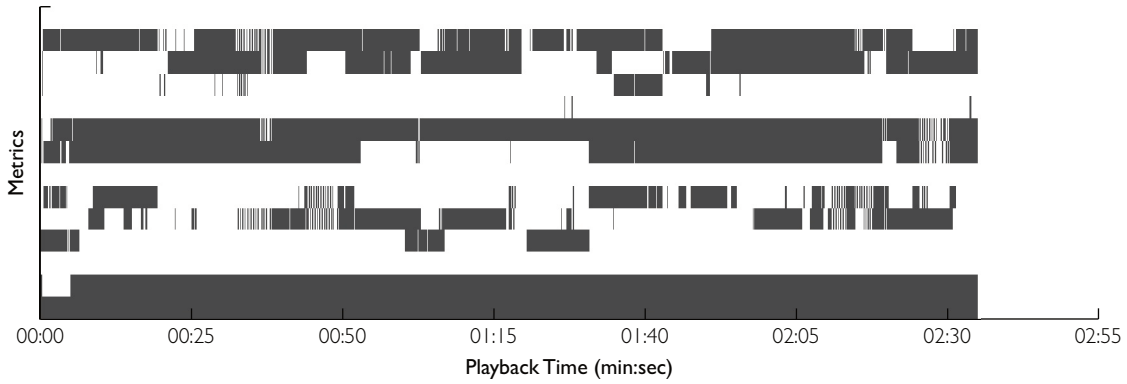


Figure 4.30: Stability of Column Subset

columns so that a computational cost is only incurred when the state of a column changes from keep to drop or vice versa. If the column state does not wildly fluctuate, the average case runtime of the predictor improves. Figure 4.30 demonstrates the stability of the subset of active columns. Shaded areas denote that the predictor uses the column corresponding to this metric, white areas signal dropped columns. We can see a band structure, where a column stays in use for multiple consecutive invocations of the predictor.

The interface between the Atlas-enhanced GCD runtime and the predictor is thin. Apart from housekeeping, it contains a function to train the predictor with a new metrics vector and the corresponding measured execution time. The second important function is to generate a prediction from a metrics vector alone. This modularity allows applications to use a custom predictor, but I think the one presented and evaluated here is a good starting point.

The linear auto-regressive predictor offers applications a suitable default predictor. It is easy to use for application developers, because it derives execution time predictions from workload metrics that are part of the application domain. I demonstrated the accuracy of the resulting predictions: Typical relative errors of less than 10% can be achieved if deep workload insight is available. Predictions with reduced workload knowledge are still competitive compared to purely time-based predictions. I have shown the hardware independence of the predictor and the sensitivity of its parameters. Metrics dropping improves prediction stability by automatically selecting relevant metrics, relieving the developer of manual pre-filtering.

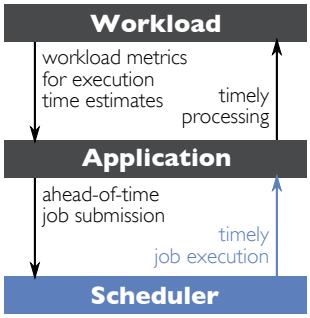
Atlas hands execution times from the predictor down to the scheduler. We have to keep in mind that over- and underestimation is possible, which the kernel scheduler must consider. I also ask the reader to remember the three evaluated prediction options: time only, reduced metrics and full metrics, because they will reappear when I evaluate the overall scheduling behavior.

System Scheduler

5

Multiple applications may independently submit jobs, which compete for execution time. Because we cannot assume applications to trust each other, a privileged management component needs to oversee and enforce the allocation of CPU time. Microkernel research explores user-level scheduling primitives[1] and hierarchical scheduling,[2] but on commodity operating systems, a single system-wide scheduler lives in the kernel.[3]

In this chapter, I motivate the design of the Atlas scheduler and explain, how it derives from the application task model. I describe properties of the scheduler contract and prove a key guarantee the scheduler provides. Atlas does not perform a formal admission, because it never rejects jobs. It also does not consider fairness between applications. The scheduler services applications to meet their timing requirements, even if that involves unevenly distributed service.



[1] Adam Lackorzyński, Alexander Warg, Michael Peter: *Virtual Processors as Kernel Interface.* 12th Real-Time Linux Workshop (RTLWS). OSADL, October 2010

[2] Bryan Ford, Mike Hibler, et al.: *Microkernels Meet Recursive Virtual Machines.* Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–151. USENIX, October 1996

[3] Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel*, Chapter 10: Process Scheduling. O'Reilly, October 2000

[4] Stefan Wächtler: *Look-Ahead Scheduling.* Master's Thesis, Technische Universität Dresden, December 2012. Master's Thesis, in German

[5] John M. Calandrino, Hennadiy Leontyev, et al.: *LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers.* Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 111–126. IEEE, December 2006

The Atlas scheduler comes as a kernel patch to Linux version 3.5.7. Stefan Wächtler implemented the scheduler during his master's thesis work, which I supervised together with Björn Döbel.[4] I explain the general design here, a thorough description, implementation details, and overhead analysis can be found in Stefan Wächtler's thesis.

The scheduler integrates in an otherwise unmodified Ubuntu Linux kernel. We could have used a scheduler testbed such as LITMUS^{RT},[5] but Atlas' aperiodic task model, its custom system call interface and the tight integration with the Linux default scheduler diminishes the potential simplifications, so we decided to go for vanilla Linux.

I describe the scheduler in three parts: I first provide context and present initial design constraints that derive from the task model. Next, I explain the scheduling algorithm assuming all reported execution times were precise. I later relax this assumption and show how the scheduler accounts for over- and underestimated execution time predictions.

Scheduling Job Sets

The Atlas task model allows applications to submit arbitrary jobs $\mathcal{J} = (e, d)$ to the scheduler. Each job $\mathcal{J}$ consists of an absolute deadline d and an execution time estimate e.[6] For now, we assume precise execution times.

[6] Revisit Figure 3.8 on page 40 for an overview of the Atlas architecture.

The order and parameters of jobs submitted to the scheduler depend on what work the applications dispatch when and to what queue. Therefore, deadlines generally do not follow any pattern such as periods and no minimum distance between two deadlines can be assumed. By calling *atlas_next,* a thread transitions execution from one job to the next and therefore from one deadline to the next. As a consequence, the urgency of the thread relative to other threads changes. Scheduling algorithms for fixed task priorities like the Rate-Monotonic Scheduling algorithm are thus unfit for the given task model.

However, the deadline for a job does not change after submission, so the class of dynamic task and fixed job priority algorithms appears attractive. For the uniprocessor case considered here, this class contains algorithms such as Earliest Deadline First,[7] which is optimal with regard to schedulability. An optimal algorithm only fails to find a schedule that fulfills all timing constraints if there is no such schedule within the bounds of the class. However, this optimality only holds for independent and preemptible tasks that do not block on shared resources or critical sections.

[7] Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

In real applications, jobs may arbitrarily block for different reasons. They may wait for peripherals or for computation results from other jobs, or they may self-suspend by just sleeping for a while. FFplay contains all these situations. The input stage waits for the video file to arrive from disk, the output stage sleeps until the next frame is due for display. The producer-consumer queues introduce dependencies: When the output stage wants to display a frame, the decoder stage must have finished preparing the frame, otherwise the output stage blocks on an empty picture queue. Unfortunately, such blocking conditions are only visible to the scheduler circumstantially and potentially too late.

An example illustrates how the Earliest Deadline First (EDF) algorithm with unmodified deadlines[8] fails to find a feasible schedule when dependencies are involved. Figure 5.1 depicts a set of three jobs (e_i, d_i), each with an execution time of $e_i = 1$ and absolute deadlines $d_A = 4, d_B = 2$ and $d_C = 3$. Job B depends on job A, meaning B can only execute after A completes. Imagine B calling the synchronization function `pthread_cond_wait()`[9] on a condition variable that A signals at the end of its execution. This dependency is spelled out in the jobs' code, but is unknown to the EDF scheduler.

[8] as opposed to EDF with effective deadlines

[9] `pthread_cond_timedwait, pthread_cond_wait` – *Wait on a Condition.* The Open Group Base Specifications, Volume 7. IEEE, 2008

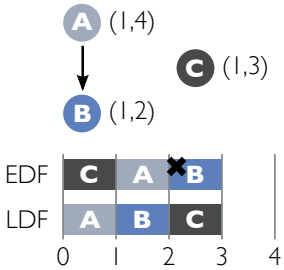


Figure 5.1: EDF and LDF Scheduling with Dependent Jobs

According to the EDF algorithm, at time instant zero, job B is selected to run, because its deadline is earliest among the present jobs. B however blocks immediately on the condition variable. The scheduler decides again and finds C to have the earliest deadline among the remaining jobs. C runs to completion at which point job A is scheduled. It also runs to completion and releases B, which can finally run, but completes at time instant three and therefore one time unit behind its deadline. Figure 5.1 shows the schedule graphically.

An optimal algorithm for scheduling jobs with dependencies is the Latest Deadline First (LDF) algorithm.[10] It iteratively picks the job with the latest deadline among all leaf jobs in the dependency graph,

[10] Eugene L. Lawler: *Optimal Sequencing of a Single Machine Subject to Precedence Constraints.* Management Science, Volume 19 (5): pp. 544–546. INFORMS, January 1973

remembers this job in a list and removes the corresponding node from the graph. It continues until the last job has been picked and then executes the list of remembered jobs in reverse order. The resulting schedule in Figure 5.1 completes all jobs before their deadline.

The LDF algorithm is optimal with dependent jobs, but requires a-priori knowledge of all dependencies.[11] The ATLAS scheduler does not have this knowledge and can only detect at runtime when a job blocks. Even worse, ATLAS can only see that job *B* blocks, but not that job *A* is responsible for resolving the blocking condition. Therefore, achieving optimality under arbitrary job dependencies remains an open problem in the ATLAS context. Our scheduler contract will have to settle for lesser guarantees.

[11] EDF is also optimal with dependent jobs, if it operates on effective deadlines. Calculating effective deadlines equally requires prior knowledge of all dependencies.

Decisions Based on Slack

The ATLAS kernel scheduler combines design cues from three existing algorithms: EDF, LDF, and LRT — the Latest Release Time scheduler,[12] often summarized as "EDF backwards." I first explain what ATLAS does and then analyze the guarantees the scheduler assures.

[12] Jane W. S. Liu: *Real-Time Systems.* Prentice Hall, Edition 1, April 2000

The kernel scheduler keeps a single global list of submitted jobs, which it orders by deadline, similar to EDF. Every newly submitted job is inserted into this list, which I call the JOB LIST. Every job is also associated with exactly one real-time task, which corresponds to one specific serial GCD queue in an application.[13]

[13] Recall that I do not consider parallel queues here.

A JOB IN THE LIST can be in one of four states: RUNNING, BLOCKED, READY, or QUEUED. A job is running, when it currently executes on the CPU. Only a single job can be running. A job waiting for a resource other than the CPU is blocked. A job is ready if it is not running or blocked and is the frontmost job in its associated real-time task. At most one job per task can be in the ready state. In the originating GCD queue, blocks are also sorted by deadline, so the block corresponding to a ready job executes next in that queue. This way, the scheduler's job list and the application's queues stay in sync. Any job not running, ready, or blocked is queued. In each real-time task, all jobs but the frontmost job are queued.

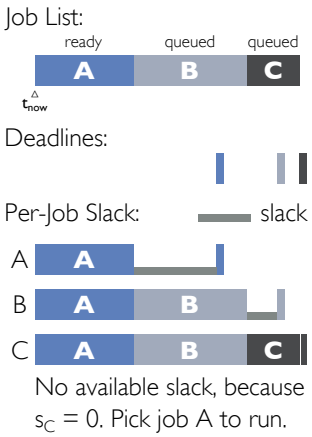


Figure 5.2: Slack Calculation

When asked to make a scheduling decision, the scheduler uses the job list to calculate the AVAILABLE SLACK[14] at the current time instant t_{now}: Every job $\mathcal{J}_i$ comes with a current per-job slack $s_i(t_{now})$, which is the difference between the time remaining until $\mathcal{J}_i$'s absolute deadline d_i and the sum of all execution times of jobs scheduled to run between t_{now} and d_i:

$$s_i(t_{now}) = (d_i - t_{now}) - \sum_{\{\mathcal{J}_k \mid \text{job scheduled between } t_{now} \text{ and } d_i\}} e_k$$

Figure 5.2 visualizes this calculation. The available slack $s(t_{now})$ is the minimum of all $s_i(t_{now})$.[15] The intuition behind available slack is that we can safely delay execution of all upcoming jobs for another $s(t_{now})$

[14] Literature sometimes uses the term static slack, but usually with schedules based on worst-case execution times. ATLAS estimates job execution times based on runtime knowledge, so the word "static" is inappropriate.

[15] The actual implementation is optimized and does not continuously recalculate all per-job slacks to find the minimum.

time units without violating any timing constraints.[16] The calculation of available slack ensures, that there is still enough time to finish all jobs before their deadline. For reasons I motivate after this formal description, ATLAS only picks a job when the available slack is zero.

If $s(t_{now})$ is zero, then some $s_i(t_{now})$ was zero, meaning we have a job in the system which misses its deadline if we do not choose to run a job now. According to EDF, we select the frontmost ready job from the job list, set its state to running and return it as the job to execute.[17] As in Figure 5.2 on the previous page, this frontmost ready job is not necessarily the job with zero per-job slack.

If $s(t_{now})$ is greater than zero, there is no urgent need to run a job now. The scheduler selects no job to run, but programs a timer to decide again after $s(t_{now})$ time units, at which point the available slack will be zero. This scheduling strategy is similar to LRT, which also runs work only when the available slack is zero.

Apart from the principal scheduling decision, the other operations of the ATLAS scheduler are technical and I briefly list them here:

- Before every scheduling decision, the currently running job transitions to the ready state. At this point, the execution time it already consumed is subtracted from its job description to ensure correct slack time calculations.

- Calling *atlas_submit* to report a new job forces a scheduling decision.

- When the running job calls *atlas_next,* it is removed from the scheduler and a scheduling decision is made. Removing the job causes the next job in the corresponding real-time task to transition from the queued state to ready.

- When the running job blocks, its state changes from running to blocked. Unblocking reverts the state to ready. Both events cause a scheduling decision.

- If the execution time of a job was underestimated, a timer fires once the reported time is depleted. A similar timer fires when a job has passed its deadline. In both cases, the job is removed from the job list, but remains registered with its real-time task. The job is no longer involved in scheduling decisions, but the next job in the real-time task does not become ready because its predecessor has not yet completed. I discuss this situation later.

WORK-CONSERVING SCHEDULERS always execute when jobs are ready. From the description of the ATLAS scheduling algorithm, we can observe that it is not work conserving: It may not return a job to run although there are ready jobs. In fact, ATLAS only runs jobs when the available slack is zero, at which point any further delay would lead to a certain deadline miss. The research community has explored non-work-conserving schedulers to improve CPU response times[18] or throughput.[19] Here, the rationale behind this design is threefold:

[16] Too Seng Tia: *Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1995

[17] Due to latencies in the Linux kernel, s may drop below zero, in which case we also run the frontmost ready job. To avoid these situations and reduce the number of context switches, values of s less than 1 ms are considered to be zero.

[18] Emilia Rosti, Evgenia Smirni, et al.: *Analysis of Non-Work-Conserving Processor Partitioning Policies.* Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), pp. 165–181. Springer, April 1995

[19] Alexandra Fedorova, Margo Seltzer, Michael D. Smith: *A Non-Work-Conserving Operating System Scheduler for SMT Processors.* Proceedings of the 2nd Workshop on the Interaction Between Operating Systems and Computer Architecture (WIOSCA), pp. 10–17, June 2006

- The ATLAS task model does not express an arrival time to the scheduler. Any job with a constraint of not running before a certain time must self-suspend. The FFplay output stage does so because it wants to display video frames neither too early nor too late. ATLAS runs jobs as late as possible which helps to minimize such self-suspensions. The hidden assumption is that most deadline-limited work has its execution sweet spot toward the end of its scheduling window. Other models like the gravitational task model[20] can express execution sweet spots explicitly, ATLAS treats it heuristically.

- When slack calculation reveals that no real-time job needs to run, we can use the CPU to improve response times for non-real-time work.[21] I want to run ATLAS-enabled applications next to unmodified Linux applications, so the scheduler has to expect significant non-real-time load. Part of it may be applications with user interfaces, which run as regular time-shared threads, but expect timely service. Even applications converted to ATLAS like FFplay start with a main thread, which acts as the application backbone and orchestrates all real-time work, but is initially not real-time scheduled itself. To solve this chicken-and-egg problem, time not claimed by the ATLAS scheduler is handed to Linux' default CFS scheduler. The LRT-like strategy guarantees that time is donated to CFS as early as possible, until an urgent real-time job must run. Within its time allocation, CFS can apply all its throughput and responsiveness optimizations to improve service to non-real-time applications. Reaping the benefits of existing non-real-time schedulers is a known idea in real-time research.[22]

- The non-work-conserving scheduling also benefits jobs, which have overrun their announced execution time. They can catch up quickly by using the available slack early. I return to this point later.

In summary, the ATLAS scheduler behaves like a hybrid between EDF and LRT: It executes jobs in deadline order, but delays execution until the available slack is zero.

WHAT GUARANTEES can we expect from this scheduler? For independent, preemptible and not self-suspending jobs on a single processor, EDF is an optimal algorithm. I prove by schedule transformation that the ATLAS scheduler retains this property:

1. Take an arbitrary but fixed set of independent, preemptible and not self-suspending jobs. EDF can successfully schedule this job set, because the criteria for EDF optimality are met.
2. Iterate over all jobs in the EDF schedule in descending deadline order. Move each visited job maximally to the right, until it either hits its deadline or collides with the next job in ascending deadline order. Observe that each moved job still completes before its deadline and that jobs are not reordered, because no job moves beyond its successor. Figure 5.3 illustrates the schedule transformation.

[20] Raphael Guerra, Gerhard Fohler: *A Gravitational Task Model for Target Sensitive Real-Time Applications.* Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS), pp. 309–317. IEEE, July 2008

[21] John P. Lehoczky, Sandra Ramos-Thuel: *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems.* Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 110–123. IEEE, December 1992

[22] Lars Reuther, Martin Pohlack: *Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS).* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 374–385. IEEE, December 2003

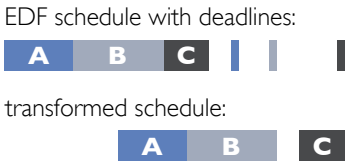


Figure 5.3: Schedule Transformation

3. In the thus constructed schedule, available slack is zero whenever a job is scheduled to begin execution, i.e., at its release time. We can prove this by induction over all jobs in descending deadline order: Per-job slack and thus available slack is trivially zero at the release of the last job $\mathcal{J}_{last}$, because its execution starts at $d_{last} - e_{last}$. For any other job, the same argument applies if it ends at its deadline. For any job $\mathcal{J}_i$ not ending at its deadline, we know by the induction premise that available slack is zero at the release of $\mathcal{J}_{i+1}$. Thus, there is some job $\mathcal{J}_\star$ whose per-job slack is zero at the release time r_{i+1} of job $\mathcal{J}_{i+1}$: $s_\star(r_{i+1}) = 0$. Moving the time instant under consideration from r_{i+1} to r_i adds $r_{i+1} - r_i$ time units to the slack of $\mathcal{J}_\star$. But simultaneously, the execution of $\mathcal{J}_i$ is now part of the interval $[r_i, d_\star]$, so we subtract its execution time: $s_\star(r_i) = s_\star(r_{i+1}) + (r_{i+1} - r_i) - e_i$. By construction there is no gap between the execution of $\mathcal{J}_i$ and its successor $\mathcal{J}_{i+1}$. Thus, $r_{i+1} - r_i = e_i$, resulting in $s_\star(r_i) = s_\star(r_{i+1}) = 0$. Therefore, at the release of job $\mathcal{J}_i$, the per-job slack of $\mathcal{J}_\star$ is still zero and thus the available slack is zero as well, which completes the induction.
4. The constructed schedule runs all jobs in deadline order and only when available slack is zero. The presented ATLAS algorithm also runs all jobs in deadline order and only when available slack is zero. Without loss of generality we can assume the same tie-breaking rules in EDF and ATLAS. Therefore, deadline order is unambiguous and the constructed schedule and the one ATLAS finds are identical.
5. The constructed schedule and thus the ATLAS schedule are feasible, because no job completes after its deadline. The presented transformation therefore generates a feasible ATLAS schedule for every feasible EDF schedule. Because EDF is optimal, ATLAS is optimal.

What remains for discussion are the requirements of EDF optimality: preemption, no self-suspension and independence.

Preemption is unproblematic, because all ATLAS jobs are fully preemptible, a feature naturally provided by the Linux scheduling infrastructure.

Self-Suspension is alleviated by the late execution of jobs as discussed previously. Jobs are still free to shoot themselves in the foot by sleeping past their deadline, but I consider this a programmer error. The guarantees ATLAS aims to provide collapse in the presence of jobs self-suspending arbitrarily.

Dependencies thwart the optimality of EDF, as we have seen by example. LDF however is optimal for dependent jobs. From the way LDF operates, it is easy to show that both algorithms find the same schedule, as long as deadlines never decrease along edges of the dependency graph:

LDF iteratively picks the latest-deadline job among the leaf jobs in the dependency graph. If deadlines do not decrease along graph edges, the job selected by LDF also has the latest deadline overall,

disregarding dependencies, because all non-leaf jobs have earlier deadlines. With compatible tie-breaking to resolve ambiguities, LDF results in the same deadline-ordered task list as EDF.

Because LDF is optimal for all dependent job sets, it is also optimal for job sets with non-decreasing deadlines along dependency edges. EDF results in the same schedule for those job sets and therefore must be optimal as well. Applying the preceding proof shows that ATLAS is also optimal for such job sets.

The limitation that dependent jobs must have a deadline not earlier than their prerequisite job is not difficult in practice. The deadline flow I devised for FFplay follows the flow of data through the player queues, so the output job for a given frame has a later deadline than the matching decode job.

Relaxing the Assumptions

Leaving the perfect world of nicely arranged deadlines and precise execution times, I now discuss what the scheduler can do for applications that do not behave perfectly. Care must be taken not to break the properties I have shown above. I first talk about over- and underestimated execution times, followed by blocking. ATLAS programs High Precision Event Timers[23] to police deadline or reserved execution time overruns.[24]

Overestimated Execution Times are the easiest to handle. Whenever a job calls *atlas_next* early, before the reserved execution time has been consumed, the remaining time becomes available as additional dynamic slack. ATLAS does not donate slack across real-time jobs,[25] all slack goes to CFS instead. Robustness against runaway jobs is one reason for this design decision, which I explain further during the evaluation.

Underestimation is by far the harder problem. Our LRT-inspired schedule comes with the downside that any reservation overrun leads to a certain deadline miss. ATLAS mitigates this problem by always making the frontmost ready job eligible for scheduling by the default CFS scheduler. Whenever slack time in ATLAS causes CFS to run, it not only time-shares the CPU among all non-real-time threads, but also finds one extra thread from the ATLAS world. The time a job receives in CFS is not accounted against the reported execution time estimate, thereby allowing for a free head start, albeit with no progress guarantee. If the job even completes in CFS, the next job from ATLAS' job list takes its place. If sufficient compute time is available in CFS, this head start mechanism thus transforms the non-work-conserving LRT-arrangement of ATLAS jobs into a classical work-conserving EDF release.

A job may still overrun its reservation, even with the help of the head start. In this case, ATLAS has to protect the remaining jobs to uphold the scheduler guarantees. The infringing job is marked

[23] *IA-PC HPET (High Precision Event Timers) Specification.* Intel Corporation, Edition 1.0a, October 2004

[24] Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda: *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* Proceedings of the IEEE International Conference on Multimedia Computing and Systems (MCS), pp. 90–99. IEEE, May 1994

[25] Caixue Lin, Scott A. Brandt: *Improving Soft Real-Time Performance Through Better Slack Reclaiming.* Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS), pp. 410–421. IEEE, December 2005

as late and is removed from the ATLAS job list. CFS exclusively schedules all late jobs in the slack left over by ATLAS. Because slack is available early, late jobs get a chance to catch up quickly.

Blocking can push job completion behind the deadline, for example when a job experienced an unexpected delay while waiting for a peripheral device or a needed result from another job. In order to protect innocent jobs, ATLAS has to stop considering the delayed job after its deadline has passed. However, as long as this job has not depleted its reserved execution time, we do not want to drop it to CFS. Instead, we allow jobs behind their deadline to consume their remaining reservation with a higher priority than all CFS threads. If the job overruns its reservation, it is still demoted to CFS. This policy does not diminish service to CFS, because the total CPU time claimed by ATLAS and therefore the available slack do not change. Blocking just distributes slack time differently: When a job has blocked, CFS already received extra time earlier.

The implementation uses a scheduling band between ATLAS and CFS to catch such delayed jobs. All jobs within this band have deadlines lying in the past, but the band is still scheduled according to EDF to allow jobs with the highest tardiness to complete first.[26] Note that the ATLAS guarantees are unaffected, because this band only runs when ATLAS does not select a job to run.

[26] Alexander D. Stoyenko, Leonidas Georgiadis: *On Optimal Lateness and Tardiness Scheduling in Real-Time Systems.* Computing, Volume 47 (3-4): pp. 215–234. Springer, 1992

THE IMPLEMENTATION OF ATLAS within the Linux scheduling infrastructure introduces two new layers between the existing CFS and POSIX real-time bands. Figure 5.4 shows the resulting scheduler stack, including the two special layers stop and idle. Linux calls the schedulers in a strict hierarchy, with any lower layer only receiving time not claimed by the upper layers. Consequently, the stop and POSIX real-time layers could take time away from ATLAS. However, the stop layer is only used by the Linux kernel internally for rare operations during CPU shutdown. Access to the POSIX real-time layer can be controlled by way of the `RLIMIT_RTPRIO` resource limit.[27] The placement of the ATLAS layer retains the semantics of the POSIX real-time layer. Simultaneous use of both layers would require more sophisticated analysis methods[28] to determine the guarantees provided by ATLAS.

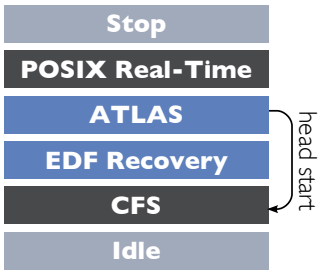


Figure 5.4: Linux Scheduler Layers

Available slack in the ATLAS layer automatically passes down to the EDF recovery layer, which catches jobs that overran their deadline due to blocking. When this layer has no jobs to run, the default CFS scheduler executes, but it cooperates with the ATLAS layer to implement the head start mechanism.

[27] `sched_setscheduler`, `sched_getscheduler` – *Set and Get Scheduling Policy/Parameters.* Linux Man Page, Volume 2, 2012

[28] Insik Shin, Insup Lee: *Periodic Resource Model for Compositional Real-Time Guarantees.* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 2–13. IEEE, December 2003

THE SCHEDULER CONTRACT provided by ATLAS is comprehensible and useful. To receive strong guarantees, submitted execution times must be precise, jobs must not self-suspend, and deadlines must never decrease along dependency edges. Developers can reason about these requirements by analyzing just their own application code, without making assumptions about system context or other applications. In

return, ATLAS guarantees optimal behavior regarding schedulability: When there is a feasible schedule, all jobs will meet their deadlines.

For imperfect real-time applications, where the developer did not spend the analysis effort or where technical reasons prevent satisfying all requirements, ATLAS employs band-aid mechanisms to uphold good service: Jobs get a free head start in EDF order to remedy wrong execution times. Jobs delayed by blocking or self-suspension can consume their reservation with higher priority than any CFS job. Lastly, ATLAS donates its slack time to CFS early, allowing late jobs to catch up quickly and non-real-time threads to make progress. In all these cases, ATLAS ensures that misbehaving jobs never interfere with the timeliness of behaving jobs.

ATLAS offers a vertically integrated solution, fully implemented from an asynchronous-lambda-based application runtime and a linear auto-regressive predictor down to the kernel scheduler. On all layers, developers need to reflect only on application-local behavior. Based on a clear contract, the scheduler provides useful timeliness guarantees. The custom GCD runtime mediates between application and scheduler, enabling developers to drive the entire machinery by way of a single function call: `dispatch_async_atlas`.

FFplay meets all conditions of the scheduler contract but one: We have seen that predictions are precise and I argued that the deadline placement intuitively follows dependencies. However, the output stage may self-suspend. Frames should not be displayed too early, so this violation is due to technical requirements of the application.

Additionally, all ATLAS guarantees collapse when the system is permanently or temporarily overloaded. The scheduler interface does not allow to reject jobs, because I think users and developers would be startled. But thanks to jobs being submitted ahead of time, ATLAS is able to detect overload and notify applications. In the next chapter, I evaluate this feature and the overall scheduling behavior.

Timely Service

6

The Atlas interface is friendly to developers, but does it provide useful real-time behavior to applications? I answer this question with experiments[1] using three different workloads: gesture tracking, user interface responsiveness, and the FFplay video player. I show how Atlas satisfies the timing constraints of these applications when the system is not overloaded. Finally, I validate my claim that for an overloaded system, Atlas can anticipate deadline misses before they occur, thereby helping applications to react to overload early. The chapter closes with a discussion of previous research results and how they relate to Atlas.

In select experiments, I compare the scheduling behavior of Atlas with competing schedulers by running background load next to the real-time application under test. In the following, whenever the term background load is used without further explanation, it refers to ten concurrently running CPU-hogging processes that each execute a tight endless loop.[2] Background load is scheduled by the Linux default CFS scheduler.

In all experiments, the Atlas execution time predictor runs with the default aging factor of 0.01 and column contribution threshold of 1.1 as previously established.

To account for scheduling overhead, time measurement jitter, and unexpected latencies in the runtime library[3] or the Linux kernel, all job execution times are enlarged before being submitted to the scheduler. I oversubscribe jobs by 2.5% of their execution time as motivated by micro-benchmarks in Stefan Wächtler's master's thesis.[4] Otherwise, a job which precisely needs its specified execution time may overrun its deadline because of these overheads and inaccuracies. Oversubscribing all jobs by 2.5% of course leads to an equivalent loss of 2.5% schedulable utilization. Short jobs with execution times below 1 ms receive a minimum oversubscription of 25 μs. This extra time becomes a limitation only with hundreds of short jobs per second, a scenario which Atlas is currently not optimized for.

Bodytrack

The first evaluation workload is part of the Parsec 3.0 benchmark suite.[5] Bodytrack implements tracking of a human body's 3D pose from camera images. Figure 6.1 visualizes the tracking result. In con-

[1] The test machine is the same as in preceding experiments: a 2.4 GHz Intel Core i5-520M Arrandale with 4 GiB of 1066 MHz DDR3 SDRAM and CPU frequency scaling disabled.

[2] Their complete source code: `int main(void) { while (1); }` One of the hoggers may be instrumented to track the throughput of background work.

[3] My libdispatch reimplementation uses ordinary pthread locks at places that could benefit from lock-free synchronization with atomic instructions.

[4] Stefan Wächtler: *Look-Ahead Scheduling.* Master's Thesis, Technische Universität Dresden, December 2012. Master's Thesis, in German

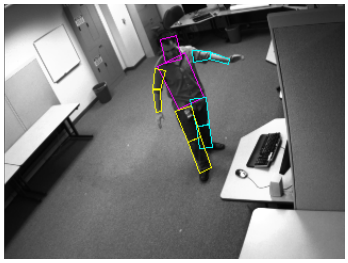


Figure 6.1: Result of the Bodytrack Benchmark

[5] Christian Bienia: *Benchmarking Modern Multiprocessors.* Ph.D. Thesis, Princeton University, January 2011

trast to commercial systems such as Microsoft Kinect,[6] Bodytrack performs unassisted tracking with no markers or depth images. I chose this benchmark because gesture tracking is part of an emerging field of user interface research.[7]

Bodytrack employs an annealed particle filter. With its default parameters of 4000 particles and five annealing layers, the workload is too CPU intensive for useful real-time operation on a single CPU. I therefore reduce the parameters to 750 particles and three annealing layers, resulting in an average execution time of 153 ms per camera image. This time does not fluctuate much across the test set of captured images, so a purely time-based prediction is sufficient. The error bars in Figure 6.2 indicate the upper and lower quartile of measured execution time and prediction.

[6] cf. *Microsoft: Kinect for Windows*

[7] Robert J. K. Jacob, Audrey Girouard, et al.: *Reality-Based Interaction: A Framework for Post-WIMP Interfaces.* Proceedings of the 2008 SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 201–210. ACM, April 2008

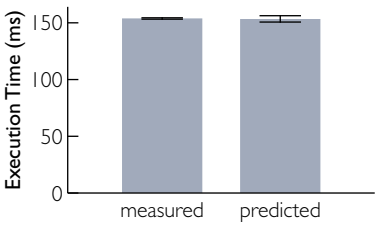


Figure 6.2: Bodytrack Execution Time

The original Bodytrack benchmark was not meant to run as a real-time application, but using gesture tracking as user input requires deadlines to ensure responsive behavior. I configured the processing of each camera image to finish within 250 ms and modified the Bodytrack code accordingly. No metrics are involved and the modifications touch just 16 lines of code.[8]

Bodytrack now behaves like a classical periodic real-time task: Jobs exhibit constant execution time and one job instance is released at the beginning of a 250 ms period and must complete within that interval. I use this opportunity to compare Atlas against a fixed priority scheduler, which is often used to schedule periodic tasks. Because the application knows it will continuously process frames, it can submit jobs in advance. My modified Bodytrack submits eight jobs ahead of time, providing the scheduler with a two second look-ahead horizon.

[8] Instead of the GCD interface, I used the lower-level estimator interface to adapt Bodytrack for Atlas.

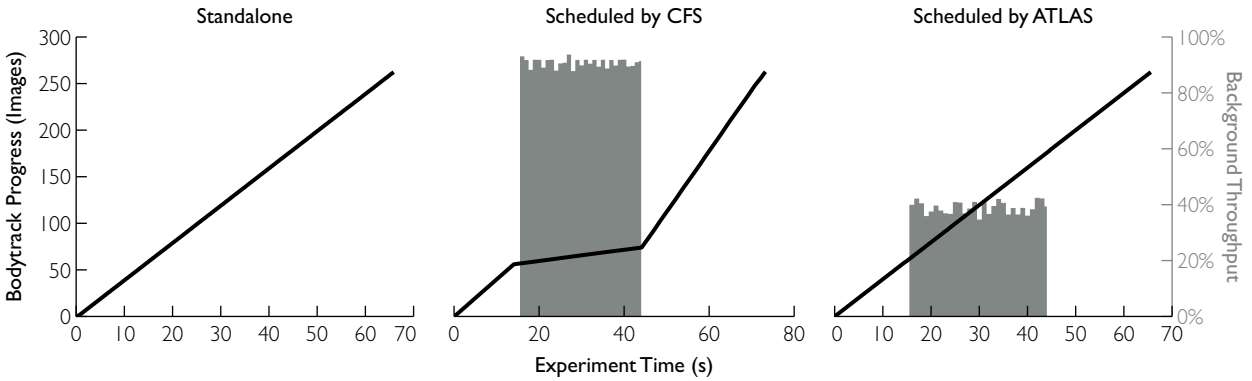


Figure 6.3: Bodytrack Scheduling Behavior

Figure 6.3 shows how Atlas protects the real-time work from competing background load. The first subfigure illustrates the behavior of a Bodytrack run with no concurrent load. The application progresses linearly through the captured image sequence. Every job meets its deadline. The second subfigure demonstrates scheduling behavior of the Linux default CFS scheduler, when Bodytrack competes with background load, which starts 15 seconds into the experiment and stops at time instant 45. Bodytrack's progress slows down because the background load receives too much CPU time. When the background load

recedes, Bodytrack tries to catch up. Scheduling under ATLAS avoids this problem. As the third subfigure confirms, ATLAS successfully schedules Bodytrack according to its specified timing requirements. Background load throttles as needed.

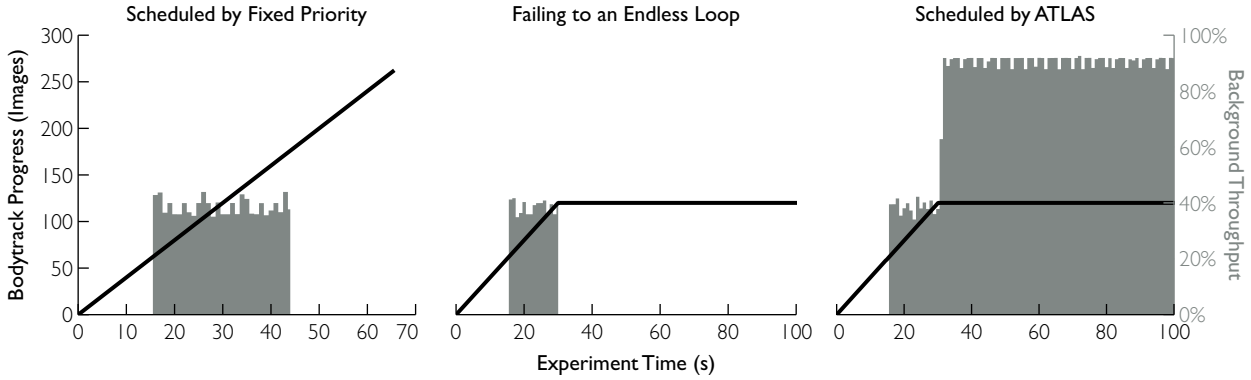


Figure 6.4: Bodytrack Failing to an Endless Loop

PERIODIC TASKS CAN BE SCHEDULED with a traditional fixed priority scheduler such as the POSIX real-time scheduler in Linux. The left part of Figure 6.4 demonstrates that prioritizing Bodytrack indeed protects it from background load similarly to ATLAS. However, when average applications without special vetting or certification are allowed to use real-time scheduling, we have to expect programmer errors. Imagine a bug that causes Bodytrack to fail to an endless loop. With fixed priority scheduling, any runaway real-time application can lock up the entire system.[9] The second subfigure of Figure 6.4 illustrates, how all background activity comes to a halt. The system becomes unusable at this point.

The third subfigure reveals how ATLAS handles these situations gracefully: The real-time application still ceases to make progress, which is expected because ATLAS cannot fix the application bug. But background load immediately receives its fair share of the CPU once Bodytrack overruns the time reserved for its job. The system is fully operable, allowing the user or a system service to kill the faulty application. This benefit confirms our design decision not to implement donation of dynamic slack across jobs, but to always pass all slack to CFS. Any failing job may otherwise receive more time by way of slack donation, prolonging the time it is able to prevent service to non-real-time applications. Instead, ATLAS demotes jobs overrunning their reservation directly to CFS. Only when they catch up they are promoted to ATLAS again.

This design is similar to the time-constrained threads facility[10] in the XNU kernel of Apple's OS X. XNU offers ordinary applications an interface to specify a period and an execution time requirement. The kernel however reserves the right to demote threads to best-effort scheduling to defend against denial of service.

In summary, ATLAS provides the same scheduling behavior to periodic tasks as a fixed-priority scheduler, but without inheriting its vulnerability against failing jobs. ATLAS handles such jobs similarly to

[9] Jason Nieh, James G. Hanko, et al.: *SVR4UNIX Scheduler Unacceptable for Multimedia Applications.* Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), pp. 41–53. Springer, November 1993

[10] *Mach Scheduling and Thread Interfaces.* Mac Developer Library. Apple Inc., February 2012. From `developer.apple.com` as of June 2013

XNU's time-constrained threads, but without requiring developers to report execution times. Due to the constant execution time of Bodytrack's jobs, enabling ATLAS scheduling was only a matter of reporting the deadline, leaving execution time recognition entirely to the runtime.

User Interface Responsiveness

Graphical user interface code has timing requirements much different from a classical periodic task. Spontaneous user interaction releases jobs without prior warning and not bound to a period. With the next set of experiments, I evaluate how ATLAS handles such jobs. I developed a test application based on GTK+,[11] the standard application framework for the GNOME[12] desktop environment. Figure 6.5 shows the minimal user interface. A click on the only button triggers work that uses the `strcasestr()` function to simulate searching a word in a document. For each click, the scanned memory is chosen randomly between 64 and 128 MiB. The chosen size is passed as the only workload metric and the work is submitted to execute asynchronously with a deadline of 100 ms, a typical time bound where users perceive a system response as immediate.[13] ATLAS scheduling is integrated into the button click handler with just seven lines of code as shown in Figure 6.6.

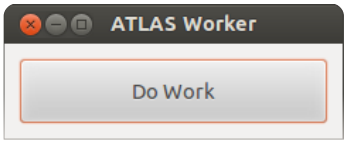


Figure 6.5: User Interface Test Application

[11] *GTK+* is a multi-platform toolkit for creating graphical user interfaces.

[12] *GNOME* is the standard desktop for the Ubuntu Linux distribution.

[13] Stuart K. Card, George G. Robertson, Jock D. Mackinlay: *The Information Visualizer, an Information Workspace.* Proceedings of the 1991 SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 181–186. ACM, April 1991

```
double metrics[] = { size };
atlas_job_t job = {
  .deadline = atlas_now() + 0.1,
  .metrics_count = 1,
  .metrics = metrics
};
dispatch_async_atlas(queue, job, ^{ scan_document(size); });
```

Figure 6.6: Pseudocode for ATLAS Scheduling in Worker

Figure 6.7 displays the accuracy of the predicted execution times. The worker application can be used interactively, but in the experiments, the button is "clicked" programmatically. Two clicks are separated by a randomly chosen interval between 0.5 and 1.5 seconds. The experiments run for five minutes each.

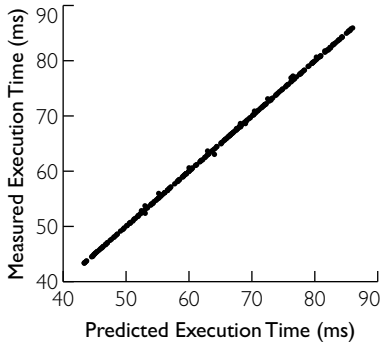


Figure 6.7: Worker Execution Time Prediction

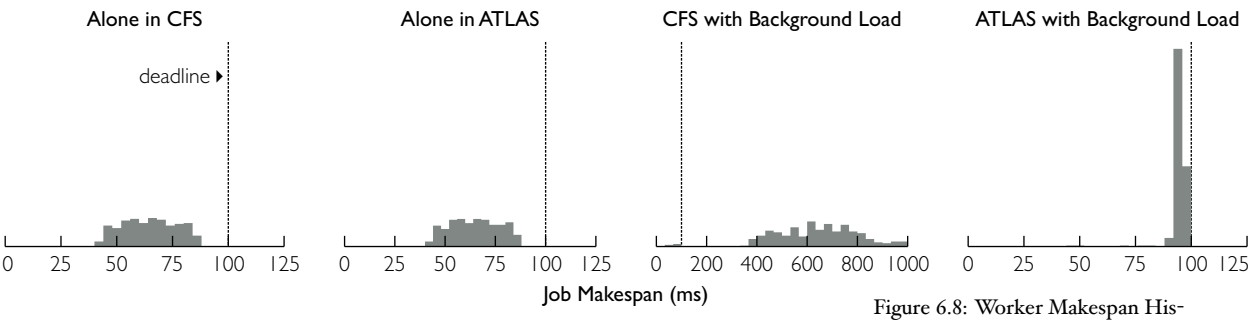


Figure 6.8: Worker Makespan Histograms

Figure 6.8 demonstrates the benefit of using ATLAS by visualizing the job makespan — the time spent between release and completion. Makespan lengths below 100 ms meet the specified deadline. The first

subfigure shows the worker scheduled by CFS and executing with no concurrent load. All jobs finish before their deadline. We observe the same completion behavior in the second subfigure, where the application is scheduled by ATLAS. Without background load, ATLAS-scheduled jobs finish as early as CFS-scheduled work thanks to the head start mechanism.

The third and fourth subfigures add background load. When scheduled by CFS, the competing load causes excessive deadline misses, with jobs taking up to one second to complete. The last subfigure illustrates the ATLAS behavior of scheduling real-time work as late as possible, but reliably completing all jobs before their deadline.

This noticeable advantage of ATLAS over CFS for spontaneous jobs was not expected, because CFS employs a concept called sleeper fairness[14] to prioritize threads waking from sleep over continuously running background activity. The worker application sleeps for long intervals in between its short bursts of activity, so it should receive preferential treatment by CFS. However, we have clearly seen that this heuristic is insufficient and that the ATLAS scheduler is better suited to ensure application responsiveness.

[14] Ingo Molnar: *CFS and SD Internals, Design.* Linux Weekly News. Eklektix, July 2007

Smooth Video Playback

I now turn to FFplay to evaluate the scheduling behavior of a complex application that exhibits highly dynamic resource requirements. At the beginning of the chapter *Real Simple Real-Time,* I motivated the need for real-time scheduling with an experiment running the Hunger Games video with competing background load.[15] Figure 6.9 repeats this experiment, comparing the original CFS behavior with ATLAS. Watching a video, the user cares about smooth playback, so the interval between the display instants of two consecutive frames should be a constant 41.7 ms, as given by the video speed of 24 frames per second. ATLAS achieves this target almost perfectly.

[15] see page 36

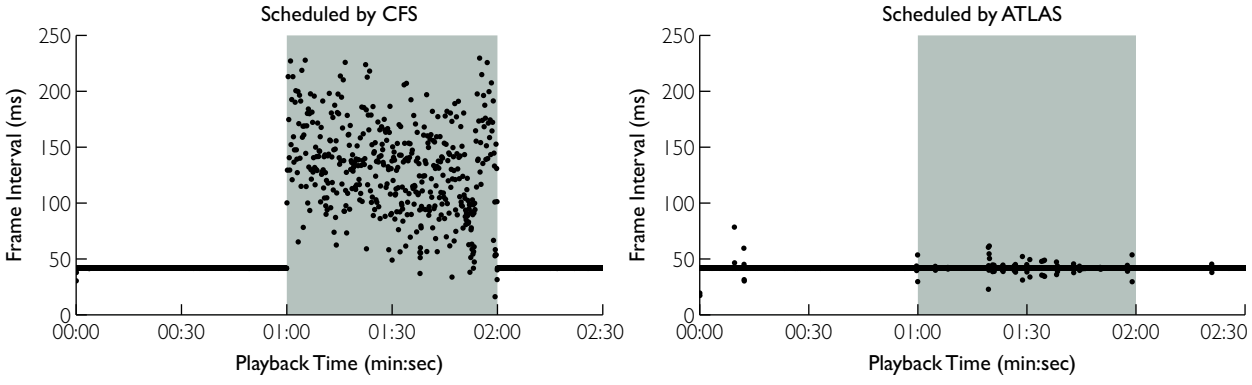


Figure 6.9: Video Smoothness with Competing Background Load

Visual inspection of playback confirms the smoothness. I asked a group of twelve human viewers to blindly judge playback smoothness of the ATLAS version with background load against a CFS-scheduled FFplay without background load. Nine of the twelve viewers could not dis-

cern a difference between the two versions, two viewers preferred the standalone CFS version, one viewer preferred the ATLAS version.

Inaccuracies in the prediction of execution times are the primary source of the remaining jitter. The next experiment thus compares the three workload metric alternatives to predict video decoding times:[16] a time-only prediction relying on previous execution times alone, the reduced metrics option which can handle unmodified video streams, and the full metrics which uses deep workload insight.

[16] see page 60 for detailed explanation

ALL FOLLOWING FIGURES compare the mean frame display jitter, which is the average time difference between the intended and actual display instants of each frame. Error bars indicate the lower and upper quartiles of the jitter. Lower values therefore represent smoother playback. All videos are played with competing background load over the entire duration of the clip except for the first second to avoid unpredictable delays in FFplay's non-real-time stream format recognition and setup code.

Figure 6.10 demonstrates the results for scheduling under CFS, which shows unacceptable playback behavior for high resolution videos.

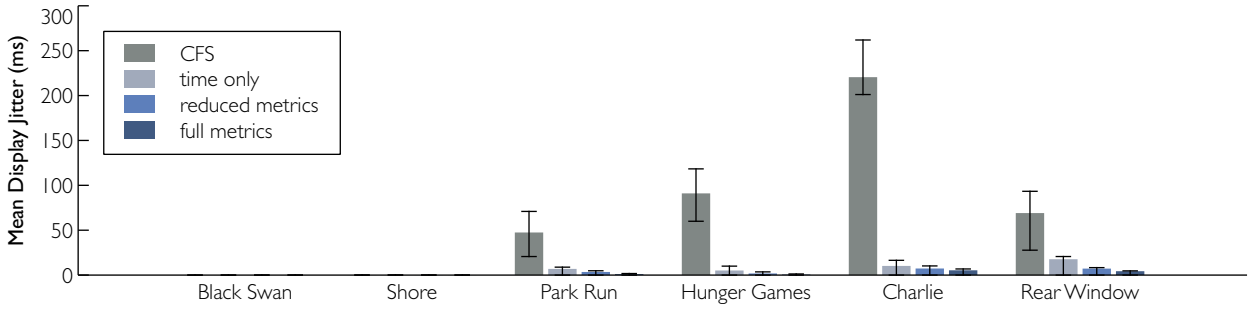


Figure 6.10: Video Smoothness with Different Schedulers

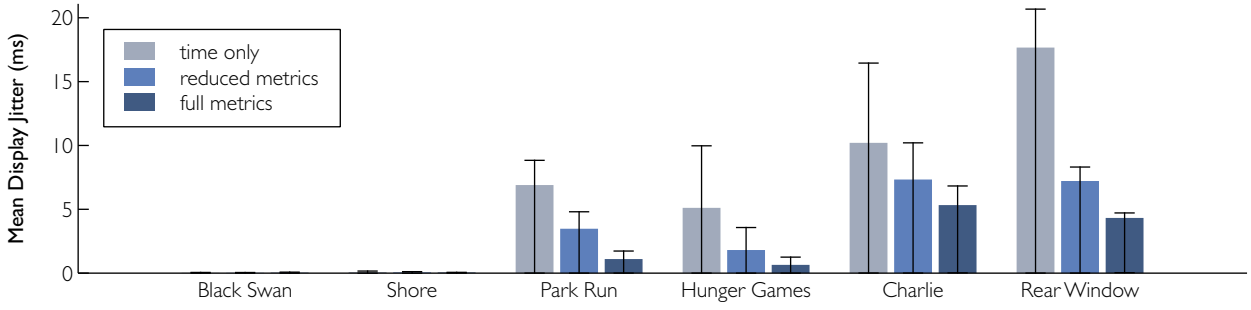


Figure 6.11: Video Smoothness with Different Metrics Options

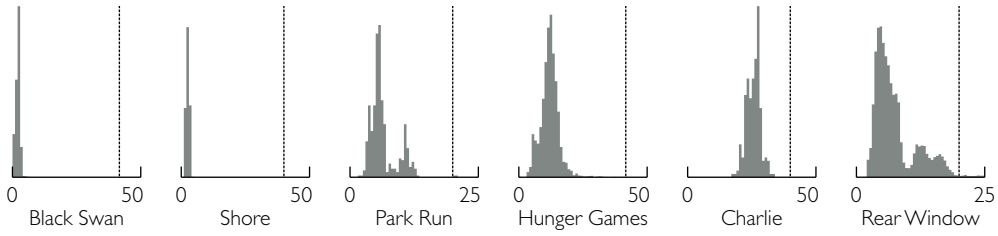


Figure 6.12: Decoding Time Histograms

All ATLAS options perform significantly better. Interestingly, CFS is able to successfully schedule the two low resolution videos in the mix, because the fair share of $1/11 \approx 9\%$ CPU time, which FFplay receives against the ten CPU hoggers is sufficient to decode these videos. Figure 6.11 shows the same values as the preceding figure, but removes the CFS bars to zoom in on the ATLAS behavior. All metrics options exhibit typical jitter of less than 20 ms, but we also see an advantage of the reduced and full workload metrics over a prediction based only on execution times. The reduced metrics, which do not require video preprocessing cut the average jitter in half compared to time-only prediction for all but the Charlie video. The full metrics further improve the scheduling behavior, down to 5 ms jitter. They represent an end-to-end design where even workload modifications are justifiable to improve real-time behavior.

Figure 6.13 details the jitter behavior of the metrics alternatives by showing a histogram of the frame interval for the Hunger Games video. The peak has been cropped to focus on the base part of the histogram. We see that the more precise metrics cause frames to group tighter around the intended display time. Figure 6.14 plots the same data along the time axis.

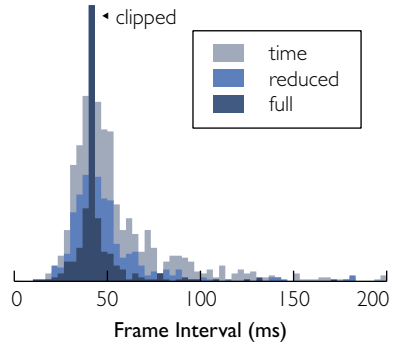


Figure 6.13: Histogram of Frame Intervals with Different Metrics Options

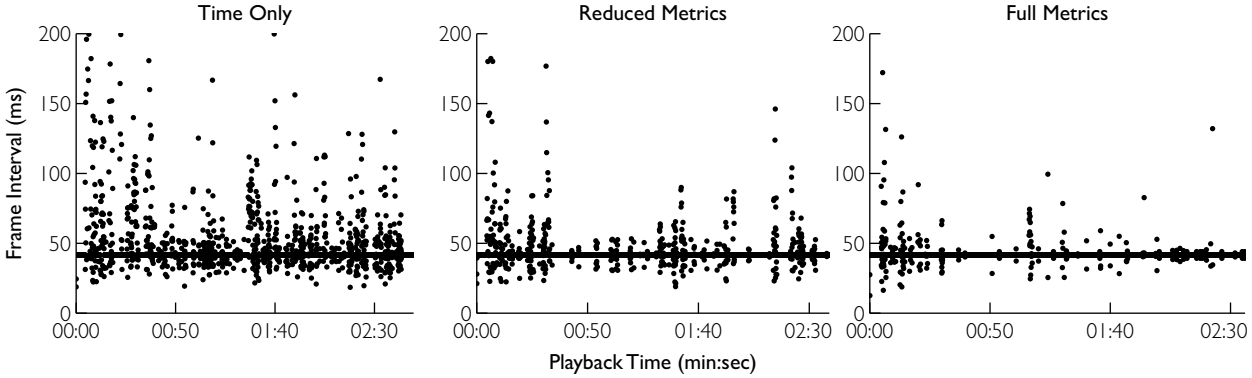


Figure 6.14: Time Profile with Different Metrics Options

Two videos require closer inspection: The Charlie clip does not benefit as much from the more detailed metrics. Remember that its prediction accuracy was also less affected by the choice of metrics compared to the other videos,[17] because the broadcast encoder creates a rather uniform video. Therefore, we cannot expect better scheduling behavior with the more detailed metrics.

The Charlie and the Rear Window videos also stand out because of their more pronounced overall jitter. Figure 6.12 shows histograms of the per-frame decoding times in milliseconds, the dotted line marks the frame interval and thus the relative deadline for frame decoding. We see that the Charlie and Rear Window videos exhibit the highest load, thus limiting the chances for blocked or delayed jobs to catch up. ATLAS is still able to schedule these videos with typical display jitter of less than 10 ms for the reduced and full metrics.

APART FROM THE METRICS ALTERNATIVES, three other factors influence the jitter behavior: the amount of background load, the job oversubscription and the placement of intermediate deadlines within

[17] see Figure 4.24 on page 62

FFplay. I briefly explore these influences using the Hunger Games video as an example.

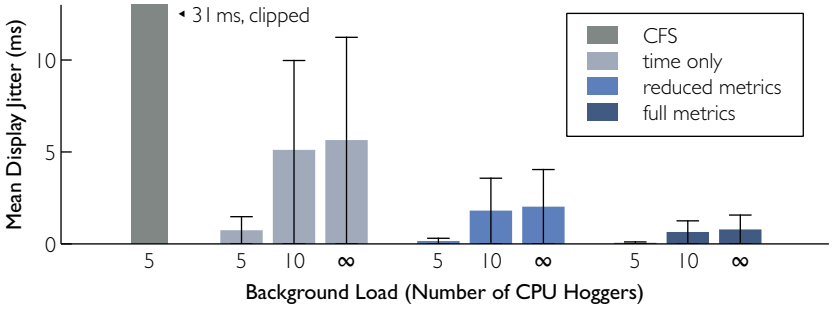


Figure 6.15: Video Smoothness Depending on Background Load

Figure 6.15 depicts the frame jitter with different levels of background load. Five CPU hoggers allow for dramatically better scheduling behavior than ten, even the less precise time-only and reduced metrics achieve unrecognizable display jitter of less than 1 ms. CFS jitter with five hoggers is still disruptive, the result is off the scale at 31 ms.

Compared to ten CPU hoggers, five of them leave more breathing room in the CFS portions that run when ATLAS recognizes available slack. Delayed jobs recover faster and the head start becomes more beneficial, leading to better scheduling behavior. This result validates the design decision of combining the non-work-conserving ATLAS real-time layer with a CFS head start and catch-up mechanism.

The implementation of the ATLAS kernel scheduler allows to disable CFS head start, to simulate a situation where excessive background load completely clogs CFS. The results do not significantly diverge from the ten hoggers case, indicating that less effective head start will not further diminish playback smoothness.

The remaining FFplay experiments revert to the default ten CPU hoggers with head start enabled.

JOB OVERSUBSCRIPTION should also affect scheduling behavior. The default setup enlarges jobs by 2.5% or 25 μs, whichever is more. I compare against no oversubscription, using the predictor output unmodified, and against dynamically oversubscribing based on the accuracy of the prediction. The latter strategy continuously keeps track of the mean squared error (MSE) of the prediction and adds its square root to all job execution times. Larger prediction errors lead to higher oversubscription.

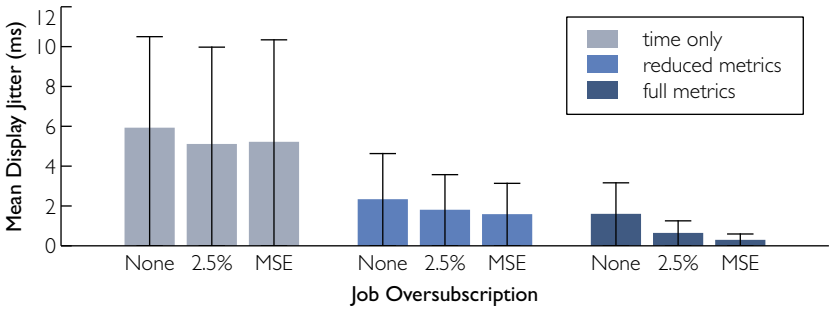


Figure 6.16: Video Smoothness Depending on Job Oversubscription

Figure 6.16 shows that the oversubscription method has little influence on scheduling behavior. Although leading to minor scheduling improvements, the dynamic strategy based on the mean squared error results in an unpredictable loss of schedulable utilization. I therefore decide against it and retain my initial choice of 2.5%.

INTERMEDIATE DEADLINES IN FFPLAY were placed early to keep internal queues filled.[18] Only the final display deadline is dictated by the frame rate of the video stream. However, the input and decoder stages also need real-time scheduling to serve the output stage in time. ATLAS does not automatically infer intermediate deadlines, so the developer needs to orchestrate the pipeline manually. All experiments so far ran with early intermediate deadlines to keep FFplay's internal queues filled.

[18] refer to page 56 for details

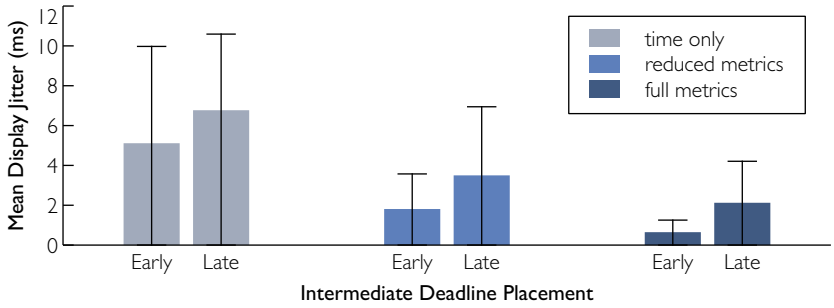


Figure 6.17: Video Smoothness Depending on Intermediate Deadlines

Figure 6.17 compares early deadlines that keep queues filled with late deadlines that allow queues to deplete. The buffering provided by the queues improves scheduling behavior, but tight scheduling with late deadlines is competitive. A video conferencing application would likely prefer the late deadlines because they reduce communication latency by discouraging buffering in the player.

Co-Scheduling Multiple Applications

The ATLAS scheduler should be able to mediate between multiple real-time applications. Constructing an application mix, we have to be careful to avoid overload situations. Therefore, I combine the user interface test application with FFplay showing the smaller videos Black Swan, Shore, and Park Run. A single CPU hogging process completes the mix. This combination pits different types of applications against each other: FFplay is complex and dynamic, the user interface worker is bursty and flushes caches by scanning large amounts of memory, and the hogger burns all remaining CPU time.

Two of these three applications come with timing constraints. The user interface worker requires between 43 and 86 ms of execution time[19] within a 100 ms scheduling window. Its jobs therefore cause bursts of CPU load from 43 to 86%.

[19] recall Figure 6.7 on page 80

In the experiment, I run the user interface application, FFplay, and the CPU hogger in different scheduling configurations. As before, I present the FFplay results by mean display jitter for each of the three

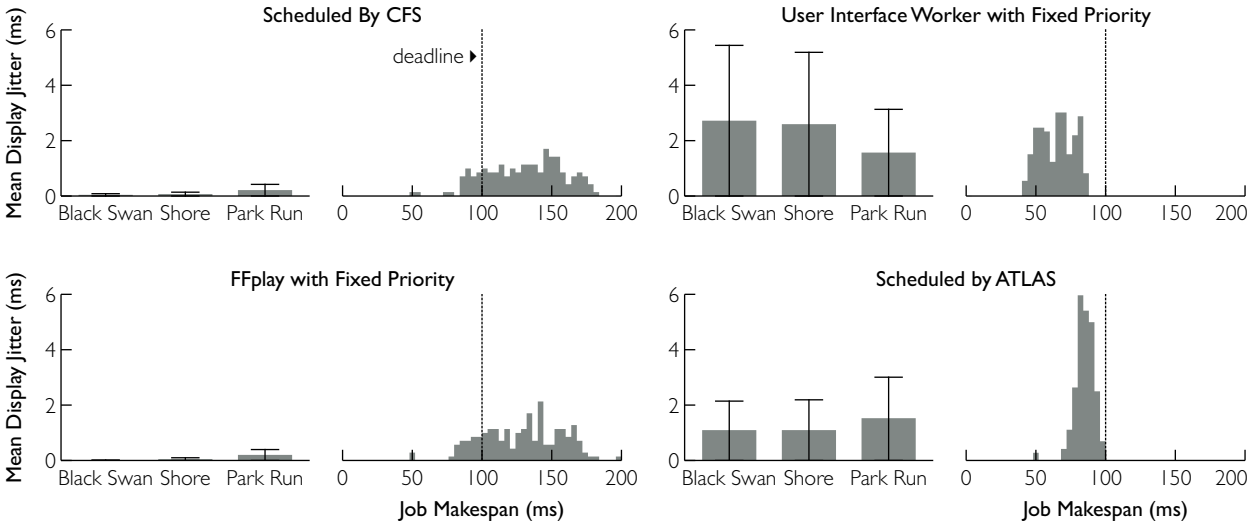


Figure 6.18: User Interface Worker Competing with FFplay

videos played. The timeliness of the user interface jobs is evaluated by showing histograms of job makespan. Figure 6.18 depicts the results. The first configuration schedules all applications by CFS. The first FFplay results show smooth playback, but the fair distribution of CPU time between all three applications leaves the user interface worker with two little time to meet its timing requirements. The first histogram shows the majority of jobs finish behind their deadline.

Trying to fix this, we run the worker in the fixed priority POSIX real-time scheduler. It now always completes in time, but video playback quality suffers. In addition, at development time, the author of the user interface application cannot know that the user will run an application mix where his program requires higher priority. Offering priorities as an interface can lead to PRIORITY INFLATION,[20] where developers pick increasingly higher priority levels to ensure their application's quality.

Consequently, the developer of FFplay may also consider his application more important, resulting in the third configuration: FFplay receives the high-priority treatment and runs smoothly, but the user interface deadlines are again violated.

Scheduling both applications by ATLAS resolves the competition. Playback is reasonably smooth and all worker jobs finish before their deadline. The remaining video jitter is has two reasons: First, FFplay jobs may underestimate their execution time. Applications that correctly report their execution time to the kernel scheduler are guaranteed to never miss their deadline. Because the user interface application predicts its jobs precisely, it is treated preferentially at the cost of delaying FFplay jobs. Second, the user interface worker requests jobs with up to 85% CPU load at very short notice. Because user interaction is spontaneous, these jobs are not submitted ahead of time and can conflict with already scheduled FFplay jobs. Therefore, the scheduler experiences brief intervals of overload, especially with the high-definition Park Run video.[21]

[20] Kenneth J. Duda, David R. Cheriton: *Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler*. Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), pp. 261–276. ACM, December 1999

[21] Figure 6.12 on page 82 illustrates the CPU load of the test clips.

Forecasting Deadline Misses

By submitting jobs before they are due for execution, applications provide the scheduler with look-ahead knowledge. This foresight enables ATLAS to anticipate deadline misses before they occur. I examine this capability using the Sintel video, which has so far been absent from the video smoothness evaluation, because it heavily overloads the system.[22]

[22] see Figure 1.6 on page 17 for a decoding time histogram of the Sintel clip

ATLAS should be able to detect this overload ahead of time. More so, it should additionally tell applications, how far behind the deadline jobs are expected to complete. This information is available in the scheduler's job list, because it aggregates all submitted jobs with their predicted execution times. Cumulating those times and comparing with the deadlines results in expected job tardiness. Unfortunately, this forecast will aggregate a number of errors: all inaccuracies in the predicted execution times and any blocking delay, clock jitter or scheduling overhead will accumulate in the completion prognosis.

Figure 6.19 illustrates that the forecast is still remarkably accurate. These scatter plots were generated by extracting expected job completion times for every job in the scheduler's job list whenever *atlas_submit* or *atlas_next* is called. Any forecasted completion is remembered and compared with the true completion time when its job finishes. The forecasted completion then results in a dot in the figure whose y-coordinate represents the error between forecasted and measured completion time. The x-coordinate marks, how early the forecast was generated relative to the job's actual completion instant. A good completion forecast happens early and with low error. Figure 6.19 shows that many forecasts are available up to half a second before job completion.

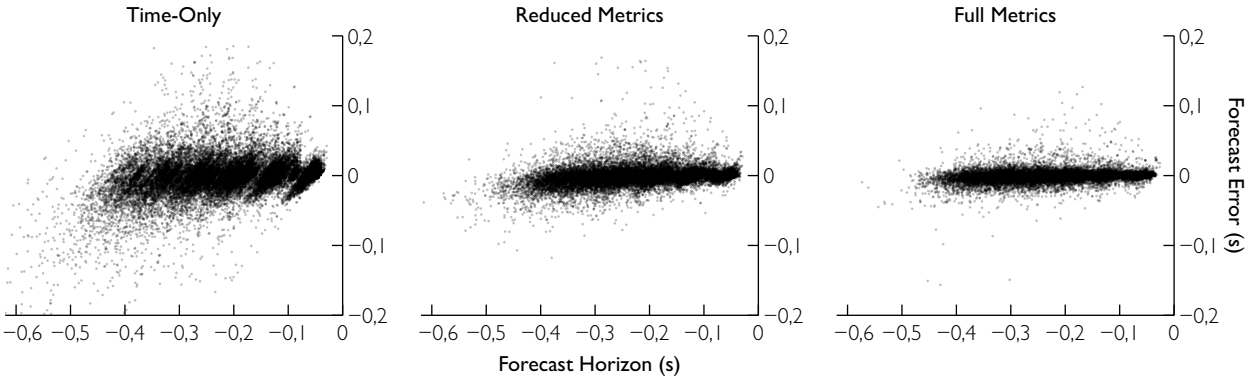


Figure 6.19: Completion Forecast for the Sintel Video

The forecast horizon is limited by the length of FFplay's video queue. ATLAS also cannot accurately forecast completion times of jobs that self-suspend, because the suspension unexpectedly delays the completion beyond the aggregate execution times from the job list. Therefore, my forecasting evaluation disregards the output stage and focusses on decode jobs instead. Because decode jobs are by far the major time consumers in FFplay, this is where the application would most likely employ overload management. Thus, accurately forecasting decode job misses is useful.

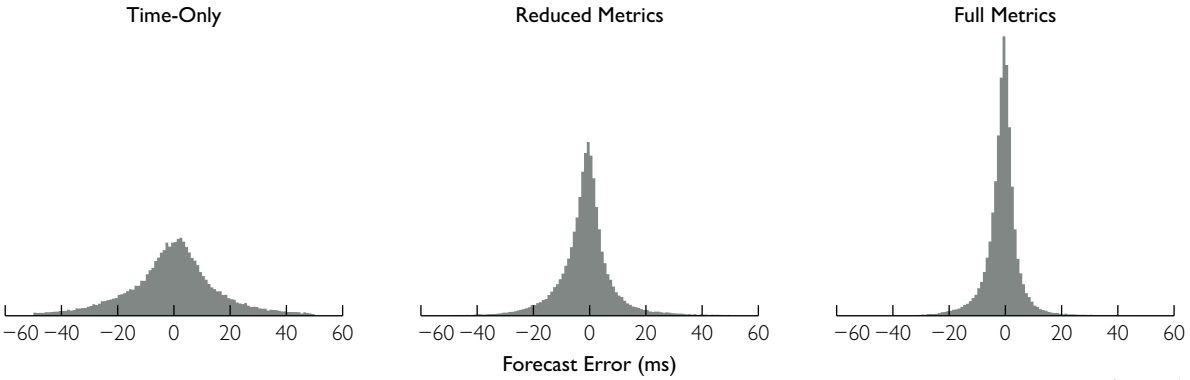


Figure 6.20: Histograms of Completion Forecast Error

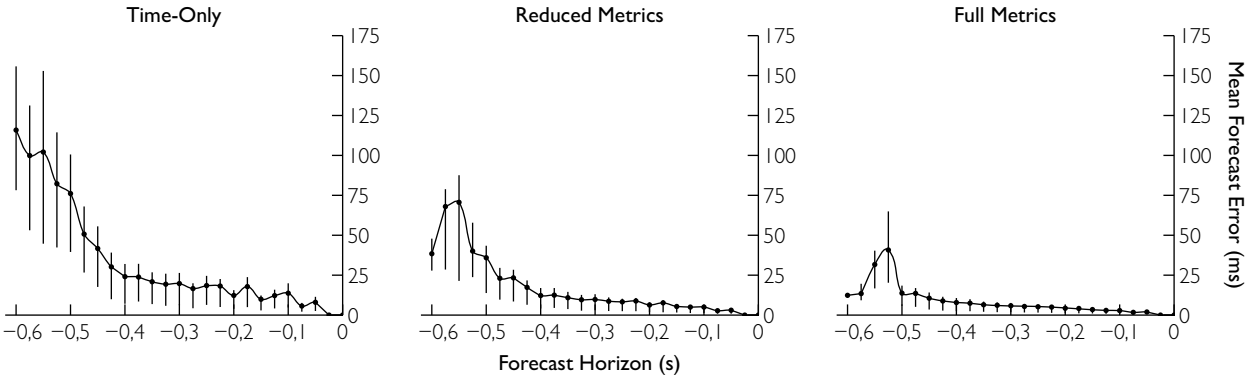


Figure 6.21: Completion Forecast Error Relative to Forecast Horizon

Figures 6.20 and 6.21 reinterpret the same data from Figure 6.19 to demonstrate the behavior along each of the axes. Figure 6.20 presents histograms of the forecast error, disregarding the time the forecast was available. Because of the point density, the error distribution is not well visible in the scatter plot. In the histograms we see that forecasts are well within a ± 20 ms error range for the reduced and full metrics. Time-only prediction is less accurate.

Figure 6.21 illustrates, how forecast accuracy changes depending on the time to job completion. Dots depict the mean forecast error, the lower and upper quartiles are given by error lines. As expected, forecast accuracy for a job improves, the closer we get to this job's completion instant. To generate the forecast for a job, predicted execution times of all preceding jobs are cumulated. The fewer such jobs we have to consider, the lower the error we aggregate from inaccurate execution time predictions. The figure shows that the reduced and full metrics allow forecasts up to half a second before job completion with typical errors under 35 ms, which is less than the video's frame interval of 41.7 ms.

Using ATLAS, applications contribute their local knowledge to the system-wide scheduler. I have shown how the scheduler uses this information to provide timely service. ATLAS successfully schedules recurring and spontaneous work and is therefore applicable to a diverse set of applications. I evaluated a gesture tracker and a responsive user interface. Unlike a priority-based scheduler, ATLAS is not vulnerable to failing applications and not subject to priority inflation. Applications with precise execution time predictions will always meet their deadline, applications with complex behavior benefit from band-aid mechanisms like head start, blocking delay recovery, and quick catch-up for late jobs.

I demonstrated how ATLAS *schedules video playback with display jitter below 5 ms when competing against aggressive background load.* ATLAS *transparently mediates between the timing requirements of multiple real-time applications and can accurately forecast deadline misses up to half a second before they occur.*

I now compare these benefits with existing research work to substantiate that ATLAS improves the state of the art in scheduling.

Related Work

In Chapter 3 on page 33, I introduced the dichotomy of scheduler interfaces: Systems that provide strong guarantees are typically hard to use for developers because they force applications into rigid task models or ask for information that is hard to obtain. The simplest scheduler interfaces support all kinds of applications without restrictions and operate with little or no parameters, but provide weak guarantees in return.

ATLAS wants to strike a balance between both extremes by combining some of their key benefits: The interface is designed from the application's perspective and only asks for local knowledge, but jobs still receive useful timeliness guarantees. ATLAS does not perform a formal admission to prevent overload, but can detect and report deadline misses before they occur.

In the following, I highlight clusters of solutions within the spectrum of scheduler research and I discuss representative work from each cluster.

THE PERIODIC TASK MODEL is the foundation for scheduling algorithms based on fixed or dynamic task priorities like the classical Rate-Monotonic Scheduling or Earliest Deadline First[23] algorithms. They target strong guarantees and thus require worst-case execution times and independent tasks to safely exclude deadline misses. Similar to ATLAS, applications specify their timing constraints and a central component calculates a schedule and enforces it. The actual job dispatching by the kernel scheduler is driven by priorities or deadlines and intentionally simple to allow formal analysis. Fixed-priority scheduling is available in Linux as part of the POSIX real-time scheduling classes. A kernel patch[24] adds deadline-based scheduling.

SOFTENING THE STRICT PERIODIC MODEL simplifies its use, but may come at the price of sacrificing some guarantees. Event stream systems[25] allow for periodic jobs with jitter or bursty behavior, but still focus on strong analysis rather than easy programming.

Worst-case execution times lead to pessimistic planning of jobs. Using execution time distributions instead of a single worst-case value enables more realistic modeling of task execution, but weakens hard deadlines to probabilistic guarantees.[26] Instead of meeting every deadline, the system only promises a percentage. ATLAS takes this idea further: Instead of using an aggregated execution time profile for all jobs in a task, it schedules every job according to its individually estimated execution time.

The Resource Kernel[27] architecture starts with a periodic task mod-

[23] Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

[24] Dario Faggioli, Fabio Checconi, et al.: *An EDF Scheduling Class for the Linux Kernel.* 13th Real-Time Linux Workshop (RTLWS). OSADL, October 2011

[25] Karsten Albers, Frank Slomka: *An Event Stream Driven Approximation for the Analysis of Real-Time Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS), pp. 187–195. IEEE, June 2004

[26] Claude-J. Hamann, Jork Löser, et al.: *Quality-Assuring Scheduling – Using Stochastic Behavior to Improve Resource Utilization.* Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pp. 119–128. IEEE, December 2001

[27] Raj Rajkumar, Kanaka Juvva, et al.: *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems.* Proceedings of the 1998 Multimedia Computing and Networking Conference (MMCN), pp. 150–164. SPIE, January 1998

el, but enhances it to improve practicality. Like ATLAS, this end-to-end design starts with an application interface and extends to kernel mechanisms. Priority inheritance handles inter-task dependencies, reservations mitigate overruns of the reported execution time. But jobs are still bound to the limits of the underlying periodic task model.

Reservation Approaches

To be usable in interactive systems and for complex multimedia applications, schedulers must integrate hard and soft real-time as well as best effort work. RBED[28] is a scheduling system that enables such integration while still preserving strong guarantees for hard deadlines. It is based on an underlying EDF scheduler, which it combines with a resource allocator to protect against overload. To support applications that do not precisely fit the periodic task model, RBED calculates safe relaxation bounds to adapt task parameters to the actual needs of the application. RBED offers developers the system calls `set_rbed_scheduler()` and `rbed_deadline_met()` to describe timing needs. They are conceptually similar to *atlas_submit* and *atlas_next,* but specify resource requirements using worst-case execution times.

[28] Scott A. Brandt, Scott Banachowski, et al.: *Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes.* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 396–407. IEEE, December 2003

THE CONSTANT BANDWIDTH SERVER (CBS)[29] is another influential approach to relax the limitations of the periodic task model. It wraps tasks with varying execution times in a server to make scheduling behavior more robust if task parameters are not exact. The underlying kernel scheduler is EDF with policing of reserved execution times.

The focus is still on temporally isolating those imperfect soft real-time loads from concurrent hard real-time load. Therefore, deadline assignment is managed by the server, not the application. ATLAS is not designed for hard real-time work and therefore can allow applications to freely manage their own deadlines. CBS provides applications with the illusion of a dedicated slower processor by supplying a virtual fluid flow of CPU time instead of catering each job individually. ATLAS services jobs as first-class citizens.

[29] Luca Abeni, Giorgio Buttazzo: *Integrating Multimedia Applications in Hard Real-Time Systems.* Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS), pp. 4–13. IEEE, December 1998

The basic CBS mechanism has been extended and enhanced in many different directions. IRIS[30] and BASH[31] distribute spare capacity to handle reservation overruns. These approaches share the same goal as the ATLAS interaction with CFS, but they function differently. The modified CBS schedulers direct slack time to overrunning jobs but keep them scheduled by EDF. ATLAS demotes misbehaving jobs straight to CFS, but also donates all slack time to CFS, which can assign this time at its own discretion. By reusing CFS this way, ATLAS keeps the system operable when real-time jobs fail into endless loops.

[30] Luca Marzario, Giuseppe Lipari, et al.: *IRIS: A New Reclaiming Algorithm for Server-Based Real-Time Systems.* Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 211–218. IEEE, May 2004

[31] Marco Caccamo, Giorgio C. Buttazzo, Deepu C. Thomas: *Efficient Reclaiming in Reservation-Based Real-Time Systems with Variable Execution Times.* IEEE Transactions on Computers, Volume 54 (2): pp. 198–213. IEEE, February 2005

TO HANDLE DYNAMIC APPLICATIONS with varying execution time requirements, Abeni et al. introduced adaptive reservations to CBS.[32] Together with a matching slack reclaiming mechanism,[33] this work resulted in the quality-of-service platform AQuoSA,[34] which was evaluated using video playback as an example for a dynamic real-

[32] Luca Abeni, Tommaso Cucinotta, et al.: *QoS Management Through Adaptive Reservations.* Real-Time Systems, Volume 29 (2): pp. 131–155. Springer, March 2005

[33] Luigi Palopoli, Luca Abeni, et al.: *Weighted Feedback Reclaiming for Multimedia Applications.* Proceedings of the 2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia), pp. 121–126. IEEE, October 2008

[34] Luigi Palopoli, Tommaso Cucinotta, et al.: *AQuoSA – Adaptive Quality of Service Architecture.* Software: Practice and Experience, Volume 39 (1): pp. 1–31. Wiley, January 2009

time load. Similar to ATLAS, AQuoSA dismisses worst-case planning, because it is wasteful. Consequently, overload cannot be prevented by admission, but needs to be handled at runtime.

AQuoSA employs a dedicated controller per real-time task, which adjusts task parameters at runtime. A system-wide scheduler mediates between all tasks in the system. This two-layer architecture is similar to the division of responsibilities between the ATLAS estimator and the kernel scheduler. However, the AQuoSA controller offers no interface for workload knowledge and only predicts from past execution times, much like the time-only metrics alternative I evaluated. Using workload knowledge, ATLAS can preconceive resource requirements of future jobs. A controller based on execution time alone can only react post-mortem, after it has seen the requirements change. However, ATLAS is not the first work recognizing the advantages of workload insight for scheduling video.[35]

Further CBS extensions combine it with a multicore scheduler based on EDF with windowed migration.[36] Multicore support is an area of future work for ATLAS.

ALL MENTIONED CBS VARIANTS require the developer to modify their application and report a period and CPU time reservation to the scheduler. Cucinotta et al. present a solution for unmodified programs.[37] They target applications with internal timing constraints that are not expressed through a defined interface. The developed Legacy Feedback Scheduler (LFS++) samples the application's system call behavior to automatically infer the period and execution time.

ATLAS exploits workload knowledge to improve scheduling, so applications have to cooperate and use the provided interface. However, this interface is inspired by the current programming trend of asynchronous lambdas. For applications built on this paradigm, the changes to enable ATLAS scheduling are small.

[35] Michael Ditze, Peter Altenbernd, Chris Loeser: *Improving Resource Utilization for MPEG Decoding in Embedded End-Devices.* Proceedings of the 27th Australasian Conference on Computer Science (ACSC), pp. 133–142. ACS, 2004

[36] Shinpei Kato, Ragunathan Rajkumar, Yutaka Ishikawa: *AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms.* Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), pp. 47–56. IEEE, July 2010

[37] Tommaso Cucinotta, Fabio Checconi, et al.: *Self-Tuning Schedulers for Legacy Real-Time Applications.* Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys), pp. 55–68. ACM, April 2010

Fair Processor Sharing

On the other end of the spectrum, fair-share schedulers offer simple interfaces at the price of weaker guarantees. The easiest interface is no interface and indeed, fair-share systems are designed to provide good average application behavior when developers do nothing. Historically, these schedulers are rooted in the days of time-sharing systems, where fairness between multiple simultaneous users was desired.

The Linux default CFS scheduler[38] follows this tradition and is a fair-share scheduler based on virtual time. I have demonstrated that CFS is inadequate for applications with timing constraints. Many research approaches start with virtual-time-based fair-share schedulers and extend them to support other guarantees than just fairness.[39] While initially starting at the other end of the spectrum, the resulting solutions overlap with the reservation-based approaches. Careful policing of sharing weights ensures real-time guarantees for virtual-time-based algorithms.[40]

[38] Ingo Molnár: *This Is the CFS Scheduler*, May 2007

[39] Andy Bavier, Larry Peterson: *The Power of Virtual Time for Multimedia Scheduling.* Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), June 2000

[40] Ion Stoica, Hussein Abdel-Wahab, et al.: *A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems.* Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS), pp. 288–299. IEEE, December 1996

Goyal et al. recognized the need for different scheduling regimes next to fair sharing in order to support multimedia applications. Start-Time Fair Queuing[41] proposes an architecture for hierarchical partitioning of CPU bandwidth. Different leaf-schedulers service applications with different timeliness requirements. However, the partitioning of CPU capacity is based on sharing weights, not on a global management of deadlines as conducted by the ATLAS scheduler.

[41] Pawan Goyal, Xingang Guo, Harrick M. Vin: *A Hierarchical CPU Scheduler for Multimedia Operating Systems.* Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 107–121. USENIX, October 1996

TO EXPRESS TASK URGENCY, the Borrowed-Virtual-Time (BVT) scheduler[42] introduces the warp value as a task parameter. While the sharing weight parameter determines the long-term CPU allocation, warp controls the short-term dispatch order of tasks. Unfortunately, the warp value bears some disadvantageous similarities to priorities: Because it is not a parameter inherent to an individual application, deciding on the right value requires knowledge of the warp values of surrounding applications. Similar to priority inflation, global policing would be needed to prevent warp value inflation.

[42] Kenneth J. Duda, David R. Cheriton: *Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler.* Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), pp. 261–276. ACM, December 1999

DEADLINES DESCRIBE TIMING REQUIREMENTS more naturally and without the need for context knowledge. The BEST scheduler[43] auto-detects periods by observing when processes enter the ready queue. It assigns deadlines accordingly and schedules jobs by EDF.

The BERT scheduler[44] extends a virtual-time algorithm with deadline handling to guarantee that a task receives a certain share of CPU cycles and that it receives them before a given deadline. Tasks reserve execution time according to an estimator observing its behavior. This estimator operates at a granularity in the order of seconds and adjusts CPU reservation in the long term. ATLAS aims for a precise allocation of every individual job. When a job overruns its reservation, BERT will steal cycles from lower-priority tasks. ATLAS demotes overrunning jobs to CFS to protect the guarantees of well-behaving jobs.

[43] Scott A. Banachowski, Scott Brandt: *The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes.* Proceedings of the 2002 Multimedia Computing and Networking Conference (MMCN), pp. 46–60. SPIE, January 2002

[44] Andy Bavier, Larry Peterson, David Mosberger: *BERT: A Scheduler for Best Effort and Realtime Tasks.* Technical Report TR-587-98, Princeton University Computer Science Department, August 1998

Real-Rate Scheduling[45] similarly proposes a task model based on shares and periods, both managed by a feedback controller at runtime. Applications can expose progress hints to the controller, for example buffer fill levels in a producer-consumer scenario. ATLAS shares the underlying idea of exploiting workload information to help scheduling.

[45] David C. Steere, Ashvin Goel, et al.: *A Feedback-Driven Proportion Allocator for Real-Rate Scheduling.* Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 145–158. USENIX, February 1999

FAIR-SHARE SCHEDULING primarily regulates the overall interaction of different loads. Servicing the timing requirements of individual applications comes second. Leslie et al. point out that the performance available to an application depends on the load of other applications. Like priorities, shares require global knowledge to determine the service a task receives.[46] The above research approaches try to retrofit timing requirements into the fair-share concept. ATLAS puts time constraints of applications first and considers fairness a secondary goal.

[46] Ian M. Leslie, Derek McAuley, et al.: *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal on Selected Areas in Communications, Volume 14 (7): pp. 1280–1297. IEEE, September 1996

Lightweight Specification

ATLAS postulates that applications should explicitly specify their real-time requirements, but with local parameters that are easy to obtain

for developers. Other scheduling architectures share similar goals and therefore come close to ATLAS in the spectrum of solutions. Compared to reservation approaches, such systems do not ask for execution times, because of their hardware-dependence. Other than fair-share approaches, these systems do not rely on sharing weights, which require contextual knowledge of the surrounding load.

SMART[47] is a scheduler for multimedia and other real-time scenarios. Applications specify their timing needs with deadlines. These are respected by the system unless it is fully loaded. User-configurable CPU shares become effective, when the system is loaded. The scheduler behaves like a hybrid between a deadline-based and a fair-share system. CPU-hogging background activity is therefore able to throttle real-time work. The rationale behind the fairness part is to keep non-real-time interactive applications responsive when the system is loaded. ATLAS strictly prefers real-time jobs over best-effort work. To maintain application responsiveness, user interface code should express its time constraints and ATLAS will satisfy them as I have shown.

[47] Jason Nieh, Monica S. Lam: *A SMART Scheduler for Multimedia Applications.* ACM Transactions on Computer Systems, Volume 21 (2): pp. 117–163. ACM, May 2003

Redline[48] is a comprehensive approach to scheduling that needs no modification of applications, but relies on simple specifications provided in external text files. The Redline example specification for mplayer — another Linux video player — statically reserves 5 ms of CPU time every 30 ms. Even though Redline will dynamically adapt the actual CPU budget at runtime, this fixed specification does not adequately address the high variability of decoder load. ATLAS automatically infers workload-aware task parameters that not only capture, but anticipate execution time variations. Apart from CPU, Redline also manages disk and memory, while ATLAS deals with CPU scheduling only.

[48] Ting Yang, Tongping Liu, et al.: *Redline: First Class Support for Interactivity in Commodity Operating Systems.* Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 73–86. USENIX, December 2008

Another scheduling infrastructure that targets high-throughput applications with timing constraints is Cooperative Polling.[49] Krasic et al. share my view that applications should be modified to expose their timing needs to the scheduler, but that the model needs to be simple for developers. Scheduling is split between an application component and the kernel. The interface primitives are similar to ATLAS. Cooperative polling however lacks a facility to provide execution times to the scheduler. CPU usage is decided by the scheduler according to a fairness policy. Because the ATLAS scheduler knows execution time estimates, it can detect future deadline misses before they occur.

[49] Charles Krasic, Mayukh Saubhasik, et al.: *Fair and Timely Scheduling via Cooperative Polling.* Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys), pp. 103–116. ACM, April 2009

Individual aspects of ATLAS have been investigated in previous research endeavors, but ATLAS is first to explore them in combination: The periodic task model requires applications to operate with a minimum job separation and constant execution times. ATLAS enables flexible submission of individual jobs with freely positioned deadlines. Reservation approaches employ feedback control to handle dynamic applications, but task parameters are only changed reactively. ATLAS uses workload insight to anticipate resource requirements and to service each job according to its actual need. Lightweight specification architectures combine useful timing guarantees with developer ease-of-use. The ATLAS programming model is equally simple, but provides the scheduler with enough information to forecast deadline misses.

The next chapter concludes the thesis and provides an outlook for future research directions.

The Road Ahead

7

In this dissertation, I demonstrated how ATLAS improves application service with regard to two important non-functional properties:

- timeliness and
- overload detection.

In this closing chapter, I outline the potential of the ATLAS architecture to support additional non-functional properties and I hope to inspire future research avenues. Specifically, I want to glimpse into:

- quality-based overload mitigation,
- efficient multicore placement, and
- energy-aware use of peripherals.

Like timeliness, these non-functional properties share the trait that proper management requires the integration of application-specific and system-wide information.

Adapt to Handle Overload

Because ATLAS lacks a formal task admission, overload can occur at any time and needs to be handled dynamically. I have shown that ATLAS can accurately forecast overload situations, but detection is only half the solution. Under overload, the scheduler can no longer service all jobs according to their timing constraints and therefore has to administer cutbacks. Jobs will receive less CPU time than expected to meet their deadline.

These reservation reductions need global oversight to maintain a system-wide fairness policy. A straightforward solution would uniformly shorten all jobs contributing to the overload. However, some applications may be able to adapt their service according to a local notion of quality. Thus, integration of knowledge can help to improve overload management. The ATLAS architecture can support such cooperation.

SIMILAR TO THE QUALITY PARAMETER in Quality Rate-Monotonic Scheduling,[1] I envision that applications specify how much resource shortage their adaptation capabilities allow them to tolerate. I developed an adaptive video player that can sustain acceptable playback with half the CPU time usually needed,[2] so it would report an ADAPTATION TOLERANCE of 0.5.

[1] Claude-J. Hamann, Michael Roitzsch, et al.: *Probabilistic Admission Control to Govern Real-Time Systems Under Overload*. Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS), pp. 211–222. IEEE, July 2007

[2] Michael Roitzsch, Martin Pohlack: *Video Quality and System Resources: Scheduling Two Opponents.* Journal of Visual Communication and Image Representation, Volume 19 (8): pp. 473–488. Elsevier, December 2008

When overloaded, the kernel scheduler would use these values to proportionally downscale job execution times. An application that can adapt more will see its time reservation reduced more. Independently reporting deadlines and adaptation tolerance decouples the notions of job urgency and job utility. The scheduler should inform the affected applications about the intended cutbacks, so they can shed load to resolve the overload before it occurs.

SCALING DOWN JOBS based on adaptation tolerance would result in a quality-fair overload management policy as opposed to the resource-fair assignment usually performed by fair-share schedulers. But consider also that applications could maliciously cause overload. Any program can allocate large amounts of CPU time by submitting many heavy-weight ATLAS jobs. As long as all well-behaving applications still meet their timing requirements, the ATLAS philosophy does not consider this situation a problem. But when the system is overloaded, we may want to punish the malicious application, even if it claims to be unable to adapt. Hybrid policies between pure quality fairness and pure resource fairness need to be evaluated.

Multicore Scheduling

Multicore processors add a second dimension to the scheduling problem: placement of work in space extends the ordering of work in time. Placement becomes even more relevant when cores differentiate,[3] either because some of them feature specialized instruction sets or because homogeneous cores show different communication latencies to main memory.[4] The placement of work onto cores also influences execution times due to contention on shared caches and memory.[5]

THE ATLAS KERNEL SCHEDULER can be extended to maintain separate job lists per core. Compared to the non-real-time schedulers in use on commodity systems today, ATLAS is in the favorable position to have detailed knowledge about the current and upcoming load situation. It can consolidate work on fewer cores and shut down unused ones[6] without jeopardizing timing constraints. This technique may be useful to save energy[7] in a time where voltage scaling becomes less effective.[8] Because parallel speedup is typically sub-linear,[9] reducing core allocation can also reduce the overhead that comes with parallel execution.

The integration of ATLAS with the Grand Central Dispatch runtime library should be extended to parallel queues. However, submitting each block in a parallel queue as an individual job to the scheduler can cause high overhead: Applications with fine-grained parallelism may dispatch thousands of blocks that each execute only for a short time. The `dispatch_apply` function enqueues all iterations of a parallel loop at once.

Extending the task model toward intra-job parallelism[10] offers a way out. A job in the scheduler would no longer represent a single block in

[3] Jian Chen, Lizy K. John: *Efficient Program Scheduling for Heterogeneous Multi-Core Processors.* Proceedings of the 46th Annual Design Automation Conference (DAC), pp. 927–930. ACM, July 2009

[4] Sergey Blagodurov, Sergey Zhuravlev, et al.: *A Case for NUMA-Aware Contention Management on Multicore Systems.* Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC), pp. 1–16. USENIX, June 2011

[5] Sergey Zhuravlev, Sergey Blagodurov, Alexandra Fedorova: *Addressing Shared Resource Contention in Multicore Processors via Scheduling.* Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 129–142. ACM, March 2010

[6] Marcus Völp, Johannes Steinmetz, Marcus Hähnel: *Consolidate-to-Idle: The Second Dimension Is Almost for Free.* Work-in-Progress Session of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, April 2013

[7] Etienne Le Sueur, Gernot Heiser: *Slow Down or Sleep, That Is the Question.* USENIX Annual Technical Conference Short Papers (USENIX ATC), pp. 217–222. USENIX, June 2011

[8] Etienne Le Sueur, Gernot Heiser: *Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns.* Proceedings of the 2010 Workshop on Power-Aware Computing and Systems (HotPower), pp. 1–8. USENIX, October 2010

[9] Gene M. Amdahl: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* Proceedings of the 30th Joint Computer Conference (AFIPS), pp. 483–485. ACM, April 1967. The paper's main result went down in history as "Amdahl's Law"

[10] Sébastien Collette, Liliana Cucu, Joël Goossens: *Integrating Job Parallelism in Real-Time Scheduling Theory.* Information Processing Letters, Volume 106 (5): pp. 180–187. Elsevier, May 2008

the application, but a group of blocks that can execute either serially on one core or spread out in parallel over many cores. The scheduler now considers jobs that can span across cores as illustrated in Figure 7.1.

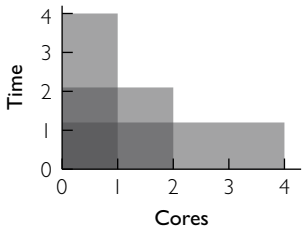


Figure 7.1: Parallel Execution Alternatives

LAST-LEVEL CACHE MISSES are a useful indicator for existing algorithms that place work on cores to improve cache sharing[11] or to avoid resource contention.[12] The ATLAS estimator, which currently predicts execution times from workload metrics can also be used to predict cache miss counts for each job.

Figure 7.2 shows the results of a preliminary experiment: Instead of measured execution times, the predictor was trained with cache miss numbers taken from hardware performance counters. It was then able to predict cache misses with an average error of only 8.9%. Because the same metrics were used, no changes to FFplay were necessary, only the internals of the estimator were modified.

This experiment shows how workload metrics can help inferring job metadata other than execution times. The process is transparent to the programmer and provides helpful information to the scheduler.

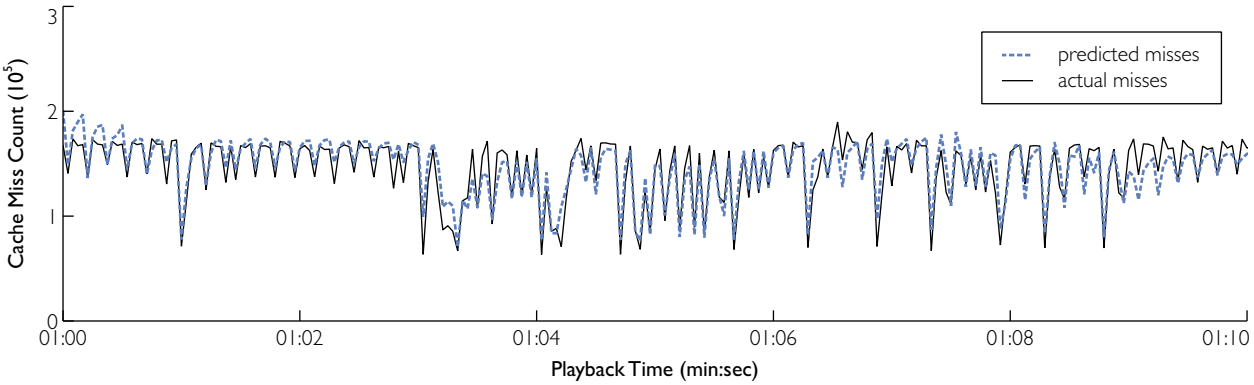


Figure 7.2: Predicting Last-level Cache Misses

Peripherals and Energy

When multiple real-time jobs wait for a peripheral, the more urgent job should receive service from the device first. Therefore, job deadlines should be propagated to device scheduling. Research has been conducted to organize GPU jobs with priority-ordered work queues.[13]

ATLAS applications currently submit jobs only for the CPU and use other devices collaterally. Extending the ATLAS concept to all resources would introduce a homogeneous system of managing work, no matter what device will execute it. Schedulers for the individual peripherals would then have knowledge of upcoming jobs and their deadlines, just like today's ATLAS provides look-ahead for CPU work. Such insight allows to power down a device, when outstanding jobs can safely be delayed, or to batch requests once the device is powered up.[14] Because power states dominate energy use,[15] such strategies are important to conserve battery life on mobile devices.[16]

[11] David Tam, Reza Azimi, Michael Stumm: *Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors.* Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys), pp. 47–58. ACM, March 2007

[12] Alexandra Fedorova, Margo Seltzer, Michael D. Smith: *A Non-Work-Conserving Operating System Scheduler for SMT Processors.* Proceedings of the 2nd Workshop on the Interaction Between Operating Systems and Computer Architecture (WIOSCA), pp. 10–17, June 2006

[13] Shinpei Kato, Karthik Lakshmanan, et al.: *TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments.* Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC), pp. 17–30. USENIX, June 2011

[14] H. Andrés Lagar-Cavilla, Kaustubh Joshi, et al.: *Traffic Backfilling: Subsidizing Lunch for Delay-Tolerant Applications in UMTS Networks.* Proceedings of the 3rd Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld). ACM, October 2011

[15] Abhinav Pathak, Y. Charlie Hu, et al.: *Fine-Grained Power Modeling for Smartphones Using System Call Tracing.* Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys), pp. 153–168. ACM, April 2011

[16] Arjun Roy, Stephen M. Rumble, et al.: *Energy Management in Mobile Devices with the Cinder Operating System.* Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys), pp. 139–152. ACM, April 2011

NON-FUNCTIONAL PROPERTIES like energy awareness and efficient use of multiple cores are increasingly important today. Parallelism has replaced frequency scaling as the means to increase performance, and in a world of battery-powered devices, saving energy becomes a critical factor for device usability. With the above starting ideas, I motivated how the ATLAS architecture can be applied as a foundation to support such non-functional properties. The ATLAS platform enables integration of knowledge and tight cooperation between applications and a system-wide scheduler.

I have presented ATLAS, the Auto-Training Look-Ahead Scheduler. It is designed to provide timeliness and overload detection, while being simple to use for developers. Its interface is aligned with the emerging programming paradigm of asynchronous lambdas. ATLAS only asks for parameters from the application domain. Deadlines express timing requirements, workload metrics describe jobs. An estimator applies machine learning to automatically and hardware-independently predict accurate job execution times from the metrics. The system scheduler orders jobs with proven optimality properties. ATLAS can service recurring and spontaneous work as well as complex applications. Video playback maintains display jitter below 5 ms when competing against aggressive background load. Thanks to its unique look-ahead and prediction abilities, ATLAS can forecast overload situations, allowing applications to adapt before deadline misses occur.

Bibliography

Luca Abeni, Giorgio Buttazzo: *Integrating Multimedia Applications in Hard Real-Time Systems.* Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS), pp. 4–13. IEEE, December 1998

Luca Abeni, Tommaso Cucinotta, et al.: *QoS Management Through Adaptive Reservations.* Real-Time Systems, Volume 29 (2): pp. 131–155. Springer, March 2005

Atul Adya, Jon Howell, et al.: *Cooperative Task Management Without Manual Stack Management.* Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC), pp. 289–302. USENIX, June 2002

Nasir Ahmed, T. Natarajan, K. R. Rao: *Discrete Cosine Transfom.* IEEE Transactions on Computers, Volume 23 (1): pp. 90–93. IEEE, January 1974

Karsten Albers, Frank Slomka: *An Event Stream Driven Approximation for the Analysis of Real-Time Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS), pp. 187–195. IEEE, June 2004

Peter Altenbernd, Lars-Olof Burchard, Friedhelm Stappert: *Worst-Case Execution Times Analysis of MPEG-2 Decoding.* Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS), pp. 73–80. IEEE, June 2000

Gene M. Amdahl: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.* Proceedings of the 30th Joint Computer Conference (AFIPS), pp. 483–485. ACM, April 1967. The paper's main result went down in history as "Amdahl's Law".

Oliver Arnold, Gerhard Fettweis: *Power Aware Heterogeneous MPSoC with Dynamic Task Scheduling and Increased Data Locality for Multiple Applications.* Proceedings of the 2010 International Conference on Embedded Computer Systems (SAMOS), pp. 110–117. IEEE, July 2010

Veronica Baiceanu, Crispin Cowan, et al.: *Multimedia Applications Require Adaptive CPU Scheduling.* Proceedings of the RTSS Workshop on Resource Allocation Problems in Multimedia Systems Scheduling. IEEE, December 1996

Scott A. Banachowski, Scott Brandt: *The BEST Scheduler for Integrated Processing of Best-Effort and Soft Real-Time Processes.* Proceedings of the 2002 Multimedia Computing and Networking Conference (MMCN), pp. 46–60. SPIE, January 2002

Andy Bavier, Larry Peterson: *The Power of Virtual Time for Multimedia Scheduling.* Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSS-DAV), June 2000

Andy Bavier, Larry Peterson, David Mosberger: *BERT: A Scheduler for Best Effort and Realtime Tasks.* Technical Report TR-587-98, Princeton University Computer Science Department, August 1998

Christian Bienia: *Benchmarking Modern Multiprocessors.* Ph.D. Thesis, Princeton University, January 2011

Sergey Blagodurov, Sergey Zhuravlev, et al.: *A Case for NUMA-Aware Contention Management on Multicore Systems.* Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC), pp. 1–16. USENIX, June 2011

Robert D. Blumofe, Christopher F. Joerg, et al.: *Cilk: An Efficient Multithreaded Runtime System.* Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pp. 207–216. ACM, July 1995

Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel*, Chapter 10: Process Scheduling. O'Reilly, October 2000

Scott A. Brandt, Scott Banachowski, et al.: *Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes.* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 396–407. IEEE, December 2003

Marco Caccamo, Giorgio C. Buttazzo, Deepu C. Thomas: *Efficient Reclaiming in Reservation-Based Real-Time Systems with Variable Execution Times.* IEEE Transactions on Computers, Volume 54 (2): pp. 198–213. IEEE, February 2005

John M. Calandrino, Hennadiy Leontyev, et al.: *LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers.* Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 111–126. IEEE, December 2006

Stuart K. Card, George G. Robertson, Jock D. Mackinlay: *The Information Visualizer, an Information Workspace.* Proceedings of the 1991 SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 181–186. ACM, April 1991

John M. Chambers: *Regression Updating.* Journal of the American Statistical Association, Volume 66 (336): pp. 744–748. American Statistical Association, December 1971

Philippe Charles, Christian Grothoff, et al.: *X10: An Object-Oriented Approach to Non-Uniform Cluster Computing.* Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 519–538. ACM, October 2005

Jian Chen, Lizy K. John: *Efficient Program Scheduling for Heterogeneous Multi-Core Processors.* Proceedings of the 46th Annual Design Automation Conference (DAC), pp. 927–930. ACM, July 2009

Michael R. B. Clarke: *Algorithm AS 163: A Givens Algorithm for Moving from One Linear Model to Another Without Going Back to the Data.* Journal of the Royal Statistical Society, Series C – Applied Statistics, Volume 30 (2): pp. 198–203. Wiley, 1981

Sébastien Collette, Liliana Cucu, Joël Goossens: *Integrating Job Parallelism in Real-Time Scheduling Theory.* Information Processing Letters, Volume 106 (5): pp. 180–187. Elsevier, May 2008

Greg Cowin: *Why Is the Next Major Paradigm Shift in Software Design About to Happen?* In Pursuit of Great Design. Private Blog, November 2011. From `www.pursuitofgreatdesign.com` as of February 2013

Tommaso Cucinotta, Fabio Checconi, et al.: *Self-Tuning Schedulers for Legacy Real-Time Applications.* Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys), pp. 55–68. ACM, April 2010

Amit Deshpande, Dirk Riehle: *The Total Growth of Open Source.* Proceedings of the 4th Conference on Open Source Systems (OSS), pp. 197–209. Springer, September 2008

Michael Ditze, Peter Altenbernd: *A Method for Real-Time Scheduling and Admission Control of MPEG-2 Streams.* Proceedings of the 7th Australasian Conference on Parallel and Real-Time Systems (PART). Springer, November 2000

Michael Ditze, Peter Altenbernd, Chris Loeser: *Improving Resource Utilization for MPEG Decoding in Embedded End-Devices.* Proceedings of the 27th Australasian Conference on Computer Science (ACSC), pp. 133–142. ACS, 2004

Kenneth J. Duda, David R. Cheriton: *Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler.* Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP), pp. 261–276. ACM, December 1999

Dario Faggioli, Fabio Checconi, et al.: *An EDF Scheduling Class for the Linux Kernel.* 13th Real-Time Linux Workshop (RTLWS). OSADL, October 2011

Sharif Farag, Ben Srour: *Improving Power Efficiency for Applications.* Building Windows 8. Microsoft, February 2012. From `blogs.msdn.com` as of May 2012

Alexandra Fedorova, Margo Seltzer, Michael D. Smith: *A Non-Work-Conserving Operating System Scheduler for SMT Processors.* Proceedings of the 2nd Workshop on the Interaction Between Operating Systems and Computer Architecture (WIOSCA), pp. 10–17, June 2006

Bryan Ford, Mike Hibler, et al.: *Microkernels Meet Recursive Virtual Machines.* Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–151. USENIX, October 1996

Ron Fosner: *Task-Based Programming – Scalable Multithreaded Programming with Tasks.* MSDN Magazine. Microsoft, November 2010. From `msdn.microsoft.com` as of May 2012

Simon F. Goldsmith, Alex S. Aiken, Daniel S. Wilkerson: *Measuring Empirical Computational Complexity.* Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 395–404. ACM, September 2007

Solomon W. Golomb: *Run-Length Encodings.* IEEE Transactions on Information Theory, Volume 12 (3): pp. 399–401. IEEE, July 1966

Pawan Goyal, Xingang Guo, Harrick M. Vin: *A Hierarchical CPU Scheduler for Multimedia Operating Systems.* Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 107–121. USENIX, October 1996

Raphael Guerra, Gerhard Fohler: *A Gravitational Task Model for Target Sensitive Real-Time Applications.* Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS), pp. 309–317. IEEE, July 2008

Nicholas Halbwachs, Paul Caspi, et al.: *The Synchronous Data Flow Programming Language LUSTRE.* Proceedings of the IEEE, Volume 79 (9): pp. 1305–1320. IEEE, September 1991

Claude-J. Hamann, Jork Löser, et al.: *Quality-Assuring Scheduling – Using Stochastic Behavior to Improve Resource Utilization.* Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pp. 119–128. IEEE, December 2001

Claude-J. Hamann, Michael Roitzsch, et al.: *Probabilistic Admission Control to Govern Real-Time Systems Under Overload*. Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS), pp. 211–222. IEEE, July 2007

David Harel, Amir Pneuli: *On the Development of Reactive System.* Logics and Models of Concurrent Systems, pp. 477–498. Springer, 1985

Hermann Härtig, Steffen Zschaler, et al.: *Enforceable Component-Based Realtime Contracts.* Real-Time Systems, Volume 35 (1): pp. 1–31. Springer, January 2007

Damir Isović, Gerhard Fohler: *Quality Aware MPEG-2 Stream Adaptation in Resource Constrained Systems.* Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS), pp. 23–32. IEEE, June 2004

Kenji Itoh, D. Huang, Takao Enkawa: *Twofold Look-Ahead Search for Multi-Criterion Job Shop Scheduling.* International Journal of Production Research, Volume 31 (9): pp. 2215–2234. Taylor & Francis, September 1993

Robert J. K. Jacob, Audrey Girouard, et al.: *Reality-Based Interaction: A Framework for Post-WIMP Interfaces.* Proceedings of the 2008 SIGCHI Conference on Human Factors in Computing Systems (CHI), pp. 201–210. ACM, April 2008

David M. Jacobson, John Wilkes: *Disk Scheduling Algorithms Based on Rotational Position.* Technical Report HPL-CSP-91-7rev1, HP Laboratories, March 1991

M. Tim Jones: *Inside the Linux 2.6 Completely Fair Scheduler*. DeveloperWorks. IBM, December 2009

Andrew Josey (Editor): *Go Solo 2 – The Authorized Guide to Version 2 of the Single UNIX Specification*, Chapter 10: Thread-Safety and POSIX.1. Prentice Hall, May 1997

Jaakko Järvi, John Freeman, Lawrence Crowl: *Lambda Expressions and Closures: Wording for Monomorphic Lambdas.* Technical Report N2550-08-0060, C++ Standards Committee, February 2008

Shinpei Kato, Karthik Lakshmanan, et al.: *TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments.* Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC), pp. 17–30. USENIX, June 2011

Shinpei Kato, Ragunathan Rajkumar, Yutaka Ishikawa: *AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms.* Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS), pp. 47–56. IEEE, July 2010

Kenny Kerr: *Visual C++ 2010 and the Parallel Patterns Library*. MSDN Magazine. Microsoft, February 2009. From `msdn.microsoft.com` as of July 2011

Andrzej Kiełbasiński, Hubert Schwetlick: *Numerische lineare Algebra: eine computerorientierte Einführung.* Deutscher Verlag der Wissenschaften, 1988

Rob Knauerhase, Romain Cledat, Justin Teller: *For Extreme Parallelism, Your OS Is Sooooo Last-Millennium.* Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar). USENIX Association, June 2012

Hermann Kopetz: *Real Time Systems: Design Principles for Distributed Embedded Applications.* Springer, Edition 2, April 2011

Charles Krasic, Mayukh Saubhasik, et al.: *Fair and Timely Scheduling via Cooperative Polling.* Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys), pp. 103–116. ACM, April 2009

Maxwell Krohn, Eddie Kohler, M. Frans Kaashoek: *Events Can Make Sense.* Proceedings of the 2007 USENIX Annual Technical Conference (USENIX ATC). USENIX, June 2007

Milind Kulkarni, Keshav Pingali, et al.: *Optimistic Parallelism Requires Abstractions.* Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 211–222. ACM, June 2007

Adam Lackorzyński, Alexander Warg, Michael Peter: *Virtual Processors as Kernel Interface.* 12th Real-Time Linux Workshop (RTLWS). OSADL, October 2010

H. Andrés Lagar-Cavilla, Kaustubh Joshi, et al.: *Traffic Backfilling: Subsidizing Lunch for Delay-Tolerant Applications in UMTS Networks.* Proceedings of the 3rd Workshop on Networking, Systems, and Applications on Mobile Handhelds (MobiHeld). ACM, October 2011

Eugene L. Lawler: *Optimal Sequencing of a Single Machine Subject to Precedence Constraints.* Management Science, Volume 19 (5): pp. 544–546. INFORMS, January 1973

John P. Lehoczky, Sandra Ramos-Thuel: *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems.* Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 110–123. IEEE, December 1992

Ian M. Leslie, Derek McAuley, et al.: *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications.* IEEE Journal on Selected Areas in Communications, Volume 14 (7): pp. 1280–1297. IEEE, September 1996

Torsten Limberg, Markus Winter, et al.: *A Fully Programmable 40 GOPS SDR Single Chip Baseband for LTE/WiMAX Terminals.* Proceedings of the 34th European Solid-State Circuits Conference (ESSCIRC), pp. 466–469. IEEE, September 2008

Caixue Lin, Scott A. Brandt: *Improving Soft Real-Time Performance Through Better Slack Reclaiming.* Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS), pp. 410–421. IEEE, December 2005

Kwei-Jay Lin, Swaminathan Natarajan, Jane W. S. Liu: *Imprecise Results: Utilizing Partial Comptuations in Real-Time Systems.* Proceedings of the 8th IEEE Real-Time Systems Symposium (RTSS), pp. 210–217. IEEE, December 1987

Chang L. Liu, James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the ACM, Volume 20 (1): pp. 46–61. ACM, January 1973

Jane W. S. Liu: *Real-Time Systems.* Prentice Hall, Edition 1, April 2000

Robert Love: *Kernel Korner – I/O Schedulers.* Linux Journal, Volume 118. Belltown Media, February 2004

Shan Lu, Soyeon Park, et al.: *Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics.* Proceedings of the

13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 329–339. ACM, March 2008

Jan M. Maciejowski: *Predictive Control with Constraints.* Prentice Hall, June 2001

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho: *Evaluating Android OS for Embedded Real-Time Systems.* Proceedings of the 6th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), pp. 62–69. Instituto Politécnico do Porto, July 2010

John Markoff: *The iPad in Your Hand: As Fast as a Supercomputer of Yore.* Bits Blog. The New York Times, May 2011. From `bits.blogs.nytimes.com` as of December 2011

Detlev Marpe, Heiko Schwarz, et al.: *Context-Based Adaptive Binary Arithmetic Coding in JVT/H.26L.* Proceedings of the 2002 International Conference on Image Processing (ICIP), Volume 2, pp. 513–516. IEEE, September 2002

Luca Marzario, Giuseppe Lipari, et al.: *IRIS: A New Reclaiming Algorithm for Server-Based Real-Time Systems.* Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 211–218. IEEE, May 2004

Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda: *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* Proceedings of the IEEE International Conference on Multimedia Computing and Systems (MCS), pp. 90–99. IEEE, May 1994

Bartosz Milewski: *Async Tasks in C++11: Not Quite There Yet.* Bartosz Milewski's Programming Cafe. Private Blog, October 2011. From `bartoszmilewski.Com` as of July 2012

Ingo Molnar: *CFS and SD Internals, Design.* Linux Weekly News. Eklektix, July 2007

Ingo Molnár: *This Is the CFS Scheduler*, May 2007

Stephan Murer, Jerome A. Feldman, et al.: *pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation.* Technical Report TR-93-028, International Computer Science Institute, December 1993

Jason Nieh, James G. Hanko, et al.: *SVR4UNIX Scheduler Unacceptable for Multimedia Applications.* Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), pp. 41–53. Springer, November 1993

Jason Nieh, Monica S. Lam: *A SMART Scheduler for Multimedia Applications.* ACM Transactions on Computer Systems, Volume 21 (2): pp. 117–163. ACM, May 2003

Jason Olson: *Keeping Apps Fast and Fluid with Asynchrony in the Windows Runtime.* Windows 8 App Developer Blog. Microsoft, March 2012. From `blogs.msdn.com` as of May 2012

Luigi Palopoli, Luca Abeni, et al.: *Weighted Feedback Reclaiming for Multimedia Applications.* Proceedings of the 2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia), pp. 121–126. IEEE, October 2008

Luigi Palopoli, Tommaso Cucinotta, et al.: *AQuoSA – Adaptive Quality of Service Architecture.* Software: Practice and Experience, Volume 39 (1): pp. 1–31. Wiley, January 2009

Abhinav Pathak, Y. Charlie Hu, et al.: *Fine-Grained Power Modeling for Smartphones Using System Call Tracing.* Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys), pp. 153–168. ACM, April 2011

R. Hugo Patterson, Garth A. Gibson, et al.: *Informed Prefetching and Caching.* Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), pp. 79–95. ACM, December 1995

Simon Peter, Adrian Schüpbach, et al.: *Design Principles for End-to-End Multicore Schedulers.* Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar). USENIX Association, June 2010

Raj Rajkumar, Kanaka Juvva, et al.: *Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems.* Proceedings of the 1998 Multimedia Computing and Networking Conference (MMCN), pp. 150–164. SPIE, January 1998

Krishna K. Rangan, Gu-Yeon Wei, David Brooks: *Thread Motion: Fine-Grained Power Management for Multi-Core Systems.* Proceedings of the 36th International Symposium on Computer Architecture (ISCA), pp. 302–313. ACM, June 2009

John Regehr, John A. Stankovic: *HLS: A Framework for Composing Soft Real-Time Schedulers.* Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pp. 3–14. IEEE, December 2001

Lars Reuther, Martin Pohlack: *Rotational-Position-Aware Real-Time Disk Scheduling Using a Dynamic Active Subset (DAS).* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 374–385. IEEE, December 2003

John C. Reynolds: *The Discoveries of Continuations.* LISP and Symbolic Computation, Volume 6 (3): pp. 233–247. Kluwer, 1993

Michael Roitzsch: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video*, June 2005. Undergraduate Thesis

Michael Roitzsch: *Slice-Balancing H.264 Video Encoding for Improved Scalability of Multicore Decoding.* Proceedings of the 7th International

Conference on Embedded Software (EMSOFT), pp. 269–278. ACM, October 2007

Michael Roitzsch, Martin Pohlack: *Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video.* Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pp. 271–280. IEEE, December 2006

Michael Roitzsch, Martin Pohlack: *Video Quality and System Resources: Scheduling Two Opponents.* Journal of Visual Communication and Image Representation, Volume 19 (8): pp. 473–488. Elsevier, December 2008

Michael Roitzsch, Stefan Wächtler, Hermann Härtig: *ATLAS: Look-Ahead Scheduling Using Workload Metrics.* Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 1–10. IEEE, April 2013

Emilia Rosti, Evgenia Smirni, et al.: *Analysis of Non-Work-Conserving Processor Partitioning Policies.* Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), pp. 165–181. Springer, April 1995

Arjun Roy, Stephen M. Rumble, et al.: *Energy Management in Mobile Devices with the Cinder Operating System.* Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys), pp. 139–152. ACM, April 2011

Francisco Sant'Anna: *The Problem with Events.* The Synchronous Blog. Private Blog, August 2008. From `thesynchronousblog.wordpress.com` as of March 2012

John E. Sasinowski, Jay K. Strosnider: *ARTIFACT: A Platform for Evaluating Real-Time Window System Designs.* Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS), pp. 342–352. IEEE, 1995

Andrea Schaerf: *Combining Local Search and Look-Ahead for Scheduling and Constraint Satisfaction Problems.* Proceedings of the 15th International Joint Conference on Artifical Intelligence (IJCAI), Volume 2, pp. 1254–1259. Morgan Kaufmann, 1997

Insik Shin, Insup Lee: *Periodic Resource Model for Compositional Real-Time Guarantees.* Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pp. 2–13. IEEE, December 2003

David C. Steere, Ashvin Goel, et al.: *A Feedback-Driven Proportion Allocator for Real-Rate Scheduling.* Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 145–158. USENIX, February 1999

Ion Stoica, Hussein Abdel-Wahab, et al.: *A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems.* Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS), pp. 288–299. IEEE, December 1996

Alexander D. Stoyenko, Leonidas Georgiadis: *On Optimal Lateness and Tardiness Scheduling in Real-Time Systems.* Computing, Volume 47 (3-4): pp. 215–234. Springer, 1992

Josef Stör, Roland Bulirsch: *Introduction to Numerical Analysis.* Springer, Edition 3, 2002

Etienne Le Sueur, Gernot Heiser: *Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns.* Proceedings of the 2010 Workshop on Power-Aware Computing and Systems (HotPower), pp. 1–8. USENIX, October 2010

Etienne Le Sueur, Gernot Heiser: *Slow Down or Sleep, That Is the Question.* USENIX Annual Technical Conference Short Papers (USENIX ATC), pp. 217–222. USENIX, June 2011

Herb Sutter: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software.* Dr. Dobb's Journal, Volume 30 (3). UBM, March 2005

Herb Sutter: *Lambdas, Lambdas Everywhere.* Session at Microsoft's Professional Developers Conference (PDC), October 2010

Herb Sutter: *Heterogeneous Computing and C++ AMP.* Keynote at AMD Fusion Developer Summit, June 2011

Herb Sutter: *Welcome to the Parallel Jungle!* Dr. Dobb's Journal. UBM, January 2012

David Tam, Reza Azimi, Michael Stumm: *Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors.* Proceedings of the 2nd ACM European Conference on Computer Systems (EuroSys), pp. 47–58. ACM, March 2007

Too Seng Tia: *Utilizing Slack Time for Aperiodic and Sporadic Requests Scheduling in Real-Time Systems.* Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1995

Marcus Völp, Michael Roitzsch: *Elastic Manycores – How to Bring the OS Back Into the Scheduling Game?*, June 2013. In submission to the 1st Workshop On Runtime and Operating Systems for the Many-core Era.

Marcus Völp, Johannes Steinmetz, Marcus Hähnel: *Consolidate-to-Idle: The Second Dimension Is Almost for Free.* Work-in-Progress Session of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, April 2013

Matt Welsh, David Culler, Eric Brewer: *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.* Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP), pp. 230–243. ACM, October 2001

Thomas Wiegand, Gary J. Sullivan, et al.: *Overview of the H.264/AVC Video Coding Standard*. IEEE Transactions on Circuits and Systems for Video Technology, Volume 13 (7): pp. 560–576. IEEE, July 2003

Stefan Wächtler: *Look-Ahead Scheduling.* Master's Thesis, Technische Universität Dresden, December 2012. Master's Thesis, in German

Ting Yang, Tongping Liu, et al.: *Redline: First Class Support for Interactivity in Commodity Operating Systems.* Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 73–86. USENIX, December 2008

Philip S. Yu, Joel L. Wolf, Hadas Shachnai: *Design and Analysis of a Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications.* Multimedia Systems, Volume 3 (4): pp. 137–149. Springer, September 1995

Dmitrijs Zaparanuks, Matthias Hauswirth: *Algorithmic Profiling.* Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 67–76. ACM, June 2012

Sergey Zhuravlev, Sergey Blagodurov, Alexandra Fedorova: *Addressing Shared Resource Contention in Multicore Processors via Scheduling.* Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 129–142. ACM, March 2010

AV Foundation Programming Guide. Mac Developer Library. Apple Inc., October 2011. From `developer.apple.com` as of July 2012

Can I Call SDL Video Functions from Multiple Threads? SDL Documentation Wiki. SDL Project, April 2008. From `libsdl.org` as of September 2012

`clock_getres, clock_gettime, clock_settime` – *Clock and Timer Functions.* The Open Group Base Specifications, Volume 7. IEEE, 2008

Concurrency Programming Guide. Mac Developer Library, Chapter Eliminating Lock-Based Code. Apple Inc., July 2012. From `developer.apple.com` as of August 2012

Concurrency Programming Guide. Mac Developer Library, Chapter Performing Tasks on the Main Thread. Apple Inc., July 2012. From `developer.apple.com` as of August 2012

Grand Central Dispatch (GCD) Reference. Mac Developer Library. Apple Inc., November 2011. From `developer.apple.com` as of November 2012

Threading Basics. Qt Reference Documentation, 4.7. Digia, 2010. From `doc.qt.digia.com` as of May 2012

OS X Human Interface Guidelines. Mac Developer Library, Chapter User Control. Apple Inc., July 2011. From `developer.apple.com` as of January 2012

IA-PC HPET (High Precision Event Timers) Specification. Intel Corporation, Edition 1.0a, October 2004

Intel® Threading Building Blocks. Intel Corporation, October 2011

Introducing Blocks and Grand Central Dispatch. Mac Developer Library. Apple Inc., August 2010. From `developer.apple.com` as of January 2012

ISO/IEC 14496-2:2004: Information Technology – Coding of Audio-Visual Objects – Part 2: Visual. ISO, Edition 3, September 2009

ISO/IEC 14882:2011: Programming Languages – C++. ISO, Edition 3, September 2011

Mach Scheduling and Thread Interfaces. Mac Developer Library. Apple Inc., February 2012. From `developer.apple.com` as of June 2013

`nice` – *Change the Nice Value of a Process.* The Open Group Base Specifications, Volume 7. IEEE, 2008

NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA Corporation, Edition 1.1, 2009

OpenMP Application Program Interface. OpenMP Architecture Review Board, Edition 3.1, July 2011

`pthread_cond_timedwait, pthread_cond_wait` – *Wait on a Condition.* The Open Group Base Specifications, Volume 7. IEEE, 2008

`sched_setscheduler, sched_getscheduler` – *Set and Get Scheduling Policy/Parameters.* Linux Man Page, Volume 2, 2012

Threads and QObjects. Qt Reference Documentation, 4.7, Chapter Threads and QObjects. Digia, 2010. From `doc.qt.digia.com` as of October 2012

The Go Programming Language Specification. Google Inc., June 2012

Threading Programming Guide. Mac Developer Library, Chapter Thread Safety Summary. Apple Inc., April 2010. From `developer.apple.com` as of May 2012

Top 5 Operating Systems in 2011. StatCounter Global Stats. StatCounter, 2011. From `gs.statcounter.com` as of January 2012

List of Figures

List of Tables

Index