Parallelizing a Real-time Audio Application - A Case Study in Multithreaded Software Engineering

Marc Aurel Kiefer, Korbinian Molitorisz, Jochen Bieler, Walter F. Tichy
Institute for Program Structures and Data Organization (IPD)
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, 76131 Karlsruhe, Germany
Email: {kiefer,molitorisz,walter.tichy}@kit.edu, jochen.bieler@student.kit.edu

***Abstract*—Multicore hardware is ubiquitous, but billions of lines of code in performance-critical commodity software are still sequential. Although parallel libraries, design patterns, and best practice guidelines are available, thinking parallel is still a big challenge for many software engineers.**

In this paper we present a case study on parallelizing commodity software using a commercial real-time audio application with over 700,000 lines of code. In contrast to best practice guidelines, our goal is to investigate what parallelization strategy can effectively be used in data stream-intensive applications. Performing an in-depth analysis of the software architecture and its run-time performance, we locate parallelization potential and propose three different parallelization strategies. We evaluate them with respect to their parallel performance impact.

Regarding the application's intrinsic real-time requirement and a very short audio cycle turnaround time, a busy-waiting strategy offers the best audio performance of 327 µs per cycle on an eight-core machine. With an efficiency of 99% this is close to the optimal schedule.

***Keywords*-Software Engineering; Multicore; Parallelization; Study**

I. Introduction

The free lunch is over [1]. Performance-critical software has to be parallel to benefit from the computing power available in modern multicore processors. Especially real-time software has to be designed carefully to make efficient use of the available computing resources in order to satisfy its real-time requirements. However, billions of lines of performance-critical software are still sequential.

Parallel libraries such as OpenMP [2], TBB [3], or Cilk [4], and parallel design pattern guidelines [5] let software engineers develop parallel software. Yet the studies in [6] and [7] show that software engineers need assistance with such complex tasks. Thinking parallel continues to be challenging for many software engineers.

With this paper we want to approach this problem in three ways: 1) We select a commercial real-time audio application with over 700,000 lines of code (LOC) as a typical example of a performance-critical software. 2) We conduct an accurate study on its parallelizability and propose three different parallelization strategies. To achieve this, we analyze the software architecture, run-time performance, and data and control dependencies. 3) We evaluate these strategies with respect to their parallel performance impact and describe their parallel potential. With the three steps we aim at providing software engineers parallelization guidelines for parallelizing performance-critical and computing-intensive commodity software.

This study has been performed under non-disclosure agreement with the collaborating company. In order to be able to publish, we anonymized the name of the audio software.

This paper is organized as follows: In section II we introduce the software we call *DJ Star*. It is used by DJs to load, manipulate and mix audio streams (and output the result via loudspeakers). The first step in our study was to reverse engineer its software architecture and run-time-intensive data structures in order to assess the parallelization potential. The results of this analysis are shown in section III. Section IV deals with the main hotspot in this application. This hotspot is a complex data structure called task graph. For this data structure we propose three different parallelization strategies. They are shown in section V. In section VI we evaluate them with respect to the real-time condition. We conclude this paper with related work in section VII and the conclusion in section VIII.

II. *DJ Star* - A real-time audio application

The application we deal with is an example of a real-time audio software used by disc jockeys. Since this is a commercial product and we are exposing some of its internal workings, we will refer to it as *DJ Star*.

The main purpose of *DJ Star* is to load, manipulate, filter, and mix audio data. In contrast to the traditional way of using vinyl records on turntables, a Disc Jockey (DJ) nowadays uses a computer to mix multiple digital tracks stored on a computer to a continuous stream of music. External hardware devices that imitate turntables are still frequently used and serve to navigate and adjust the current playback position within single tracks. A typical DJ setup is shown in Figure 1.

DJ Star supports up to four audio decks for concurrent playback of up to four tracks. The program offers a wide

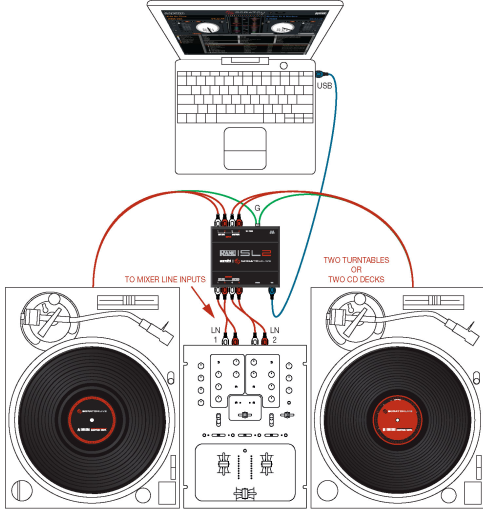

Figure 1. Example DJ Hardware Setup [8]
i

range of audio effects that can be applied to the audio stream of each track independently before the results are combined into a single audio stream. The resulting audio is streamed to the sound hardware and the music plays. The real-time constraint of *DJ Star* is: Applying audio effects must not take longer than it takes the sound hardware to play its audio buffer, which is 2.9 ms. Otherwise, the audio signal is distorted.

DJs typically bring their own hardware and computer, so they mostly use laptops for mobility reasons. Concerning the aspect of parallel computing under the constraint of real-time demand, their computing power is limited.

On screen, the disc jockey is supported with a visual representation of a workbench to help create a seamless audio experience. The workbench consists of external or virtual turntables, audio effect control, music library, and status of the current tracks (e.g. audio waveform, mixer, applied effects).

III. Analyzing *DJ Star* for Parallelization

In this section, we describe the static and dynamic analysis steps we performed to gain an insight into *DJ Star* as a typical example of a data stream intensive application. Our goal was to identify parallelization potential.

A. Static Analysis

As there was no code documentation available for *DJ Star*, we started our analysis by manually inspecting the source code in order to get an idea of the application's structure and important code regions. As *DJ Star* consists of over 700,000 LOC, we searched the code in a breadth-first manner for the most relevant pieces.

As a result, we manually identified the software architecture using reverse engineering techniques. *DJ Star* uses

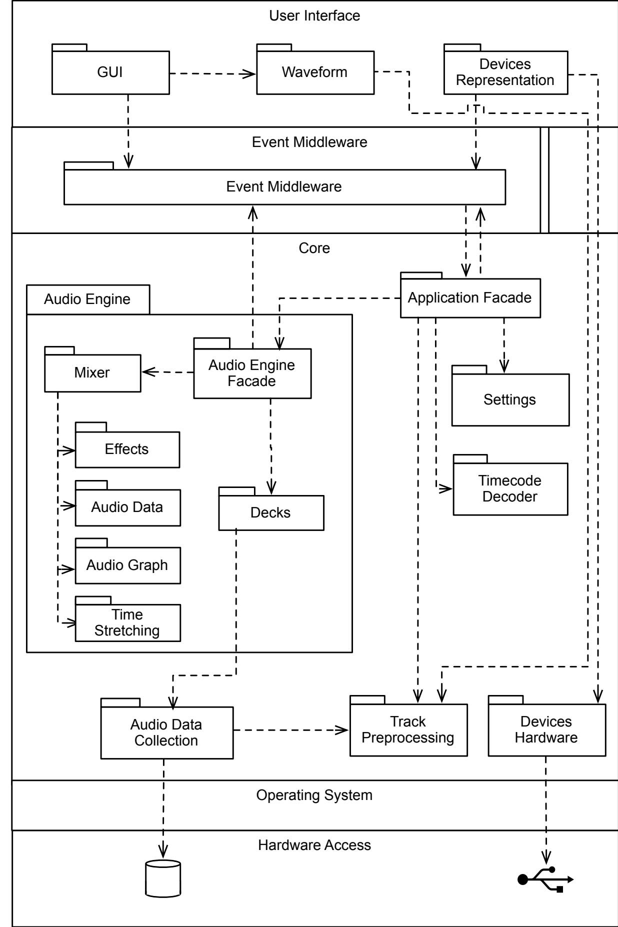

Figure 2. Software Architecture of *DJ Star*

a 4-layer architecture as described in [9] that consists of *User Interface*, *Event Middleware*, *Core*, and *Hardware Access*. The software architecture and components are shown in Figure 2. The *User Interface* layer is responsible for drawing the graphical user interface (GUI) that visualizes the audio waveform, the playback representation, and the control devices. It communicates with the *Core* subsystems indirectly via the *Event Middleware*. Most of the program logic resides in the *Core*. It also contains the *Audio Engine* subsystem which handles the audio stream processing. The *Hardware Access* layer passes through the operating system and connects directly to the hard disk for efficiently loading music files. A second task of this layer is to connect to external control devices via USB.

After reconstructing the application's architecture, we selected the *Audio Engine* as target component for our analysis, because we expected the highest performance demands. In fact, the *Audio Engine* plays the most important role in our analysis, so we concentrated on this subsystem. One of the basic concepts in real-time audio software is that the operation cycle is dictated by the sound card: Audio streams

are output at 44.1kHz, so the software needs to deliver audio samples at that rate. If this timing condition cannot be met and handing over the audio packet occurs too late, the sound hardware is forced to either replay the last audio packet or to output silence. Human hearing is very sensitive to these errors, so even the tiniest duplicated or silenced audio packet is annoying. Therefore, the delivery of audio packets at a constant rate has to be ensured at all cost.

In *DJ Star*, the audio buffer size is configurable. As disk jockeys often change effects or mixer parameters during their live performances, low latency is a key factor. This results in rather small buffer sizes. At the same time timing constraint are tightened, because a smaller buffer is processed faster and has to be filled with new audio data at a higher frequency. *DJ Star* uses a standard buffer size *BS* of 128 samples. With a sampling rate *SR* of 44.1kHz the sound card requests packets at a rate of 344.53Hz. This means that the audio processing cycle (APC) that computes the resulting audio packet must not take more than 2.9ms.

$$T(APC) < \frac{BS}{SR}$$

B. Dynamic Analysis

With the software architecture and the breadth-first source code analyses we gained a good understanding of the static structure of the software and the real-time constraint enforced by the sound card. In order to further investigate the parallelization potential, we analyzed the run-time behavior for locations that consume most of the CPU cycles. We used the profiler tool in Microsoft Visual Studio and performed a hotspot analysis.

As it turns out, 88% of the total run-time is consumed by the APC. Most of the remaining 12% fall in the scope of the GUI. A further diversification of the APC run-time reveals that 33% is used for audio stream preprocessing (time stretching, phase alignment, buffer overhead), 38% for the execution of the audio effect graph (decks, mixer, effects, ...), and 16% for the timecode decoder which interprets external control signals.

Although audio stream preprocessing and graph execution together form 71% of the total run-time, we decided not to cover audio preprocessing, because it does not carry coarse-grained parallelization potential. Furthermore, preprocessing makes use of certain common algorithms, for which parallel equivalents are already available. We could also have parallelized on the algorithm level because a lot of the run-time is spent time-stretching or computing audio effects. Audio effects heavily rely on core algorithms such as Fourier transformation, for which good parallel solutions already exist. This makes audio stream preprocessing less attractive from a research standpoint. Instead, we focus on parallelizing the main audio graph, which is structured as a graph of different tasks.

IV. Audio Graph in *DJ Star*: Task Graph

The audio processing cycle (APC) is implemented as a task graph. The nodes represent different audio computations and the edges describe the data flow or data dependencies. The data stream intensive characteristics of *DJ Star* can be seen in this graph: Data packets of audio information continuously originate from the four decks and are being passed along deck effects, filters, and equalizers to the master section. Here, they are mixed together and sent to the sound card. Figure 3 shows a simplified version of the task graph that we used during our experiments and evaluation. We assembled it to include the most relevant compute-intensive tasks and dependencies.

The actual graph contains 67 nodes of which some have no dependencies and do not modify the audio packets. Although these nodes only amount to a small portion of the APC total execution time, we also included them for a fair average.

For each task graph, it is crucial to find a good schedule that takes into account the run-time distribution of the nodes in the task graph and the existing dependencies. It is not possible to precompute an optimal schedule in advance, because the run-time distribution of the task graph and each individual node heavily depends on the number of decks used, effects, and mixer configurations together with their parameters. For nodes that alter the audio stream, the run-time additionally depends on the actual audio stream data.

In order to find an optimal schedule and to assess the parallel potential for the task graph, we performed a graph simulation using the simulation tool RESCON [10]. At first, we measured the average vertex computation time using 10k APC executions. For this, we defined the *earliest start scheduling strategy*. This strategy schedules each vertex as soon as all its dependencies are met, disregarding resource constraints (i.e. infinite processors). This approach is similar to a critical path analysis, but in addition it reveals the maximum concurrency in the graph.

Our results show that an optimal schedule computes the graph mentioned above in 295 µs and requires 33 processors. The simulation results are depicted in Figure 4, with the numbers representing node IDs and the four large blocks representing the effect nodes for each deck. As it turns out, these 33 concurrent nodes all have rather short computation times and no dependencies. This means that they can be computed in parallel right from the start. After ~25 µs the concurrency level drops down to four. This seems sensible, because we use four decks. Once the computation reaches the master section, even less parallelism is possible. Since only four cores are required most of the time, we simulated the graph with a resource constraint of four cores to find an optimal schedule. Our simulation results show that the task graph can be computed in 324 µs using only four cores. This is only 8% slower than the schedule without resource

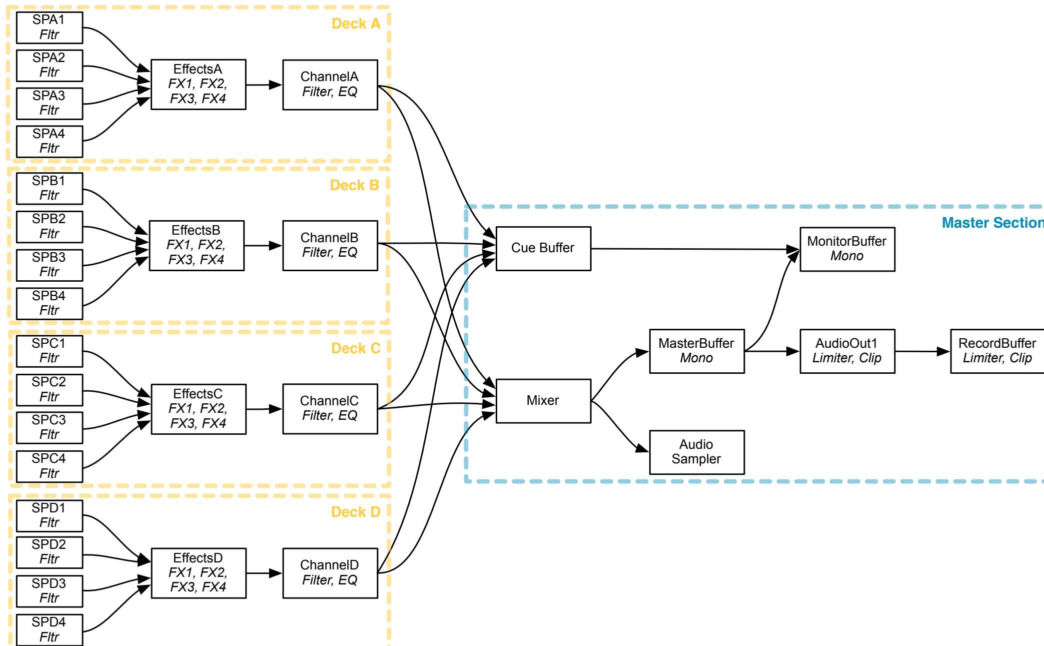

Figure 3. Audio graph configuration

constraints. At the same time it is more practical when we consider the multicore hardware in today's laptops.

As the simulation can not take into account aspects such as thread management, scheduling overhead, and dependency checking, the effective run-time numbers will be higher. Still, the simulation gives us a good estimate of what to aim for during our parallelization process.

In *DJ Star*, the task graph is implemented using a simple queue. Nodes are inserted according to their depth in the dependency graph. Figure 3 visualizes the insertion: Nodes are added column by column and from left to right. Nodes in the same column do not carry dependencies to other nodes in the same column, so single nodes can simply be removed from the queue in the same order (FIFO) during graph execution and processed sequentially.

V. Parallelization Strategies

In this section we introduce three different parallelization strategies for the performance-critical and computing-intensive audio application *DJ Star*. References [5], [11], [12] provide best practice guidelines for parallel programming that focus on general-purpose use cases, also for audio stream processing. Examples are software pipelining, data parallel processing, or task-based parallel processing.

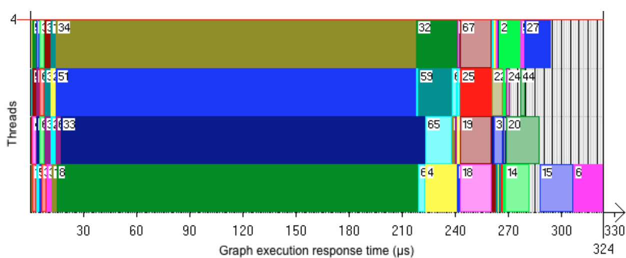

Figure 4. Simulated optimal scheduling on four cores

Like other applications from this category, *DJ Star* imposes specific constraints that distinguish it from the general approach. For *DJ Star*, the task graph cannot be executed with a data parallel strategy on different audio packets, because the packets are not available in advance. As we explained in section III, one key performance requirements is low latency on the buffers with respect to changes in effects or mixer parameters, so only one audio packet at a time is available. It has to be computed as fast as possible to prevent an audible glitch from the sound card. The same argument holds for transforming the task graph into a pipeline: The stages can be assigned, but the data stream to pass through the stages and fill the pipeline is not available. Besides data-parallelism or pipelining, a third approach is to parallelize on the algorithm level. The effect nodes in the task graph are the most expensive nodes in terms of run-time consumption, but their code is strictly sequential. Although algorithm parallelization would probably work well, here we decided against it for the reason that many of the effect algorithms are proprietary and equivalent parallel implementations already exist. We chose to parallelize the intrinsic structure and operations of the task graph by executing multiple nodes in parallel and defining a schedule that is adequate for the given constraints. We implemented the three different parallelizations *busy-waiting*, *thread-sleeping*, and

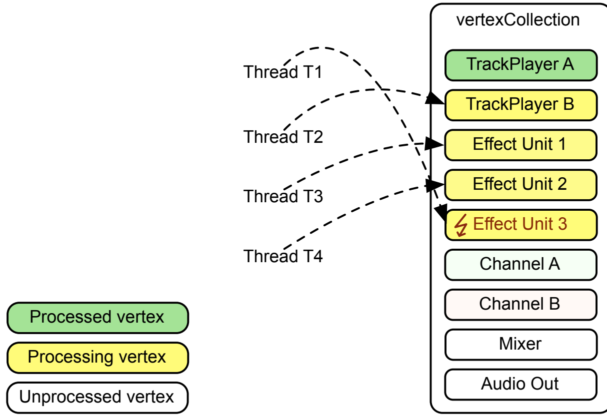

Figure 5. Busy-waiting scheduling example

work-stealing.

A. Busy-waiting

As we showed in our analysis in section III, the graph nodes are already in a sorted queue with respect to their dependencies in the sequential code. Because of this, they can be easily assigned to threads in a round-robin manner. In the parallel case, nodes might be assigned to threads before their dependent nodes are computed. To avoid this kind of dependency violation, each node performs a dependency check. When a node gets scheduled, it first checks its dependencies and performs busy-waiting until they are met.

Figure 5 shows an example for a busy-waiting loop: *Effect unit 1* and *Effect unit 2* depend on *TrackPlayer A*, and *Effect unit 3* depends on *TrackPlayer B*. As soon as thread $T1$ has finished computing *TrackPlayer A*, it is assigned to *Effect unit 3*. Since the dependencies are not yet fulfilled, $T1$ goes into busy-waiting until $T2$ finishes *TrackPlayer B*.

Busy-waiting generally consumes CPU time by actively waiting and not computing, but putting threads to sleep and waking them up is also an expensive operation. With the run-time of a single APC being in the order of microseconds and very short idle times for threads waiting for their dependencies, busy-waiting is a reasonable strategy in this scenario. In order to validate this claim, we implemented a sleep strategy that does not wait actively, but relies on external wake-up calls.

B. Thread-sleeping

Instead of actively waiting for dependency fulfillment, we implemented a second strategy which puts threads to sleep and hence prevents a waste of CPU cycles. The queue is organized as in the busy-waiting approach, but the threads are explicitly put to sleep until their dependencies are met. We implemented the wake-up calls in the computing nodes. Nodes that are finished computing send a signal to their successor node which in turn wakes up its assigned thread.

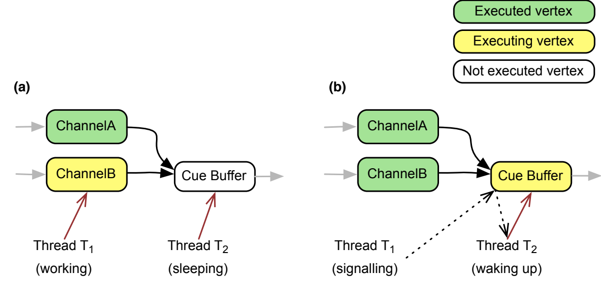

Figure 6. Thread-sleeping scheduling example

The wake up procedure only occurs when all predecessor nodes are finished and have sent the notification signal.

The mechanism is depicted in Figure 6. In (a) the thread T_2 just finished the execution of *ChannelA* and moves on to *Cue Buffer*. It check its dependencies and finds the dependency to *ChannelB* which is still executing. It therefore cannot begin the execution. Hence, T_2 registers itself as executor for *Cue Buffer* and goes to sleep. In (b) T_1 has just finished the execution of *ChannelB*, and signals *Cue Buffer* its termination. As all dependencies have now been executed, *Cue Buffer* wakes up its executor thread T_2 which immediately starts the execution.

While this approach prevents active waiting and saves CPU cycles, the implementation of the sleeping strategy is not optimal and contains further potential for improvement regarding a better management of the node queue. Currently, nodes are assigned to dedicated threads in a round-robin fashion and when a node cannot be executed, its executor thread is put to sleep. While this node might be blocked by open dependencies, there could be other ones available for computation. Instead of putting the executor thread to sleep because its node is currently blocked, it could look for other available nodes and compute them. As available nodes do not have to wait for their assigned executor thread but be executed by one thread that has just finished its work, this strategy potentially has the earliest start times for node computations. At the same time, this aspect raises the queue management overhead. As we mentioned, it generally does not take very long until open dependencies are resolved. Because of this we intentionally keep the queue structure simple. In order to assess this, we implemented a third strategy that trades early node start times for more queue management overhead and implemented a work-stealing strategy.

C. Work-stealing

The work stealing strategy addresses the problem that threads do not look for executable nodes once they finished their work, but simply go to sleep. With the sleeping strategy, an executable node informs its executor thread. Taken together, this causes an unnecessary overhead of sleep and wakeup. We implemented additional logic to let threads look

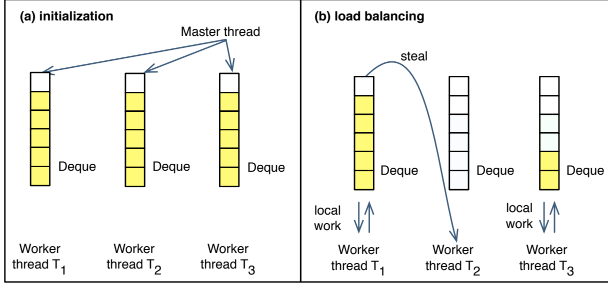

Figure 7. Work-stealing scheduling example

for other executable threads in three steps: 1) Each thread gets its own working queue. 2) This queue only contains nodes which are executable, i.e. all dependencies are met. 3) Threads can steal nodes from other threads once their own queue is empty.

Each time an executor thread finishes the computation of its node, it checks all successors of this node for unresolved dependencies. If all dependencies are met for one successor node, it adds this nodes to its own processing queue and then proceeds as expected.

Since the queues can now be accessed by multiple threads simultaneously, we have to ensure mutual exclusion to prevent data races. We implemented the queues as *double ended queues* (deque) which can be accessed from both sides. We implemented the convention that stealing threads access the queue from the top and local executor threads access their queue from the bottom (see Fig. 7 (b)). This convention enables a theft and a local access to happen at the same time as long as $length(deque) \geq 2$ without the need to use explicit locking. A second advantage is a higher cache efficiency for two reasons: 1) A thread iterates the successor nodes recursively and hence prioritizes nodes that have been added to the queue more recently (LIFO principle). This maximizes the chance that the related data are still available in the processor cache. 2) If a thread steals a node, it always gets the node with the longest waiting time in the queue. When we project this back onto the underlying graph model, this means that the stolen node is more likely to produce a high number of new tasks after its completion. These new tasks can then be processed locally by the stealing thread.

As the threads now have a local queue, they have to be filled initially. When a new APC starts, the main thread fills up the processing queues of all executor threads. It distributes all nodes without dependencies (source nodes) to the threads. This situation is depicted in Figure 7 (a). We categorize the source nodes as *Deck A/B/C/D or Master* in order to be able to assign nodes from the same section to the same thread. This supports data locality as nodes from the same section work on the same audio data.

VI. Evaluation

In this paper we designed and implemented three different scheduling strategies for the audio application *DJ Star*. We evaluated them by executing 10K audio processing cycles (APC) for each parallelization strategy and measured the execution times. Our test system was an 8-core AMD FX 8120h with 3.1 Ghz each and 8 GB RAM running Windows 7 (64-bit).

The typical average execution times and speedup values are interesting to get a general idea of how the parallelizations perform, but they are not very meaningful when it comes to satisfying the real-time condition. Therefore, we also measure worst case execution times, missed deadlines, and the distributions of the execution time between task graph iterations. After that we compare our measurements with the optimal schedule that we found in section IV using simulations. We conclude with a look at typical schedule realizations.

As our simulations showed, the maximum number of nodes that can be processed in parallel is four, due to data dependencies within the task graph, except for a small portion of the graph. In later stages of the graph the parallelism gradually decreases to one. Our measurements indicated that increasing the thread count above four does not accelerate the computations any further, and the increased thread overhead even lowers the speedup slightly. Therefore, we limited the thread count in the experiments to four. This is also an appropriate number for modern laptops used by DJs.

To fully understand the measurements with respect to the real-time constraint we need to differentiate further between APC and task graph execution: While every APC includes a task graph execution *Graph* it also has the additional components timecode processing *TP*, graph preprocessing *GP*, and various calculations[1] *VC*. So the total APC runtime calculates as:

$$T(APC) = T(TP) + T(GP) + T(Graph) + T(VC)$$

The average execution times of *TP*, *GP*, and *VC* add up to 0.8 ms. Delivering 128 audio packets per seconds to the sound card means one packet every 2.9 ms. To satisfy the real-time constraint, this leaves us with the following requirement for our task graph:

$$T(Graph) \leq 2.1ms$$

The following measurements therefore relate to task graph execution times, not APC times.

Table I shows the average execution times and Figure 8 the corresponding speedup. Comparing the busy-waiting (*BUSY*), thread-sleeping (*SLEEP*), and work-stealing (*WS*) scheduling strategies shows that they all perform very similar in terms of average execution times. The speedup in comparison to the sequential execution increases to 2.4 on

[1] Accounting calculations, for example updating the master tempo

Table I
TASK GRAPH AVERAGE RESPONSE TIMES (MS)

Threads	1	2	3	4
BUSY	1.0785	0.6371	0.5683	0.4516
SLEEP	1.1130	0.6447	0.6444	0.4657
WS	1.1111	0.6394	0.5844	0.4690

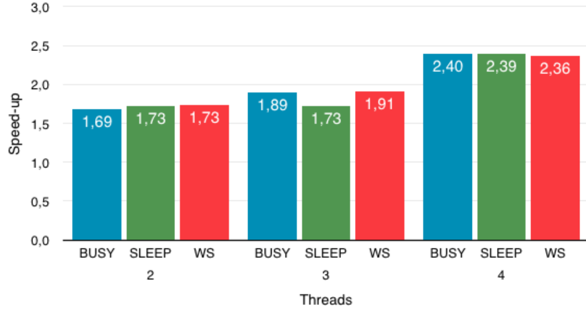

Figure 8. Speedup comparison of the scheduling strategies

four cores. Linear speedup is not to be expected due to the inherent data dependencies between nodes.

About five out of 10K APC executions exceed the deadline of 2.9 ms, although the average task graph execution time of ~0.45 ms on four cores is far below the threshold. In this case the audio gets distorted and unfortunately there is nothing we can do about it. This emphasizes why we also need to look at the worst case situations and run-time distributions. The goal is to execute as many audio packets as possible and considerably before the deadline, so headroom is created for such situations. The only way to guarantee no missed deadlines would be to use a real-time operating system, but the software is required to run on standard operating systems.

The execution time of a task graph iteration heavily depends on the audio data, so the average execution times are not significant to determine what the best scheduling strategy is. The histogram of task graph execution times for 10K iterations is depicted in Figure 9. It also shows the three different scheduling strategies. All three diagrams show two peaks, which mirror the run-time distribution of the nodes with varying input data. The BUSY strategy has a strong peak of early finishing nodes, while the SLEEP strategy clearly uses some time to wake up the threads (no graph executions below 0.4ms). It can be observed that the work-stealing approach has a rather even distribution schema, but it also has some unwanted high execution times near the 0.8 ms mark.

A cumulative histogram of the same data is presented in Figure 10. It also reveals the strong early start of the busy-waiting strategy. SLEEP starts very late but accomplishes to finish 80% of the iterations under 0.5 ms. The work-stealing approach averages the start times and has quite some

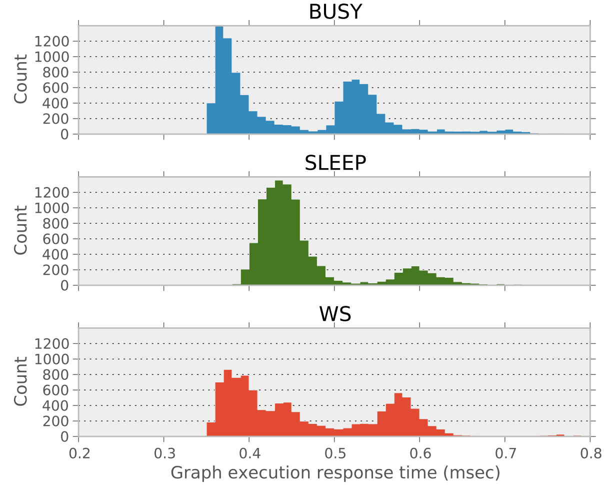

Figure 9. Execution time distributions of the scheduling strategies

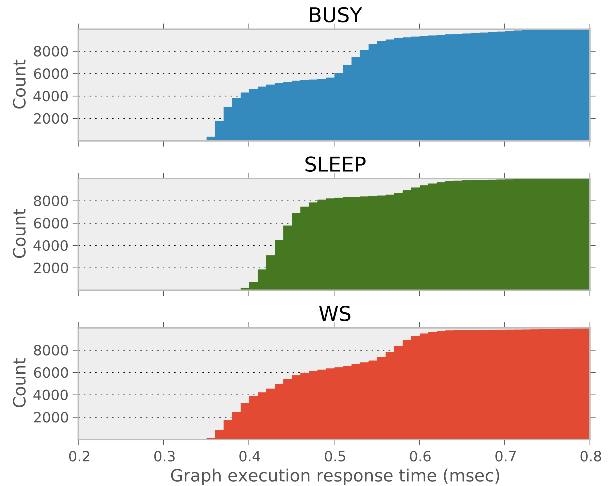

Figure 10. Cumulative histogram of execution times of the scheduling strategies

instances finish very late.

To understand how each scheduling strategy actually schedules nodes in the task graph, we visualized some typical schedule realizations in Figure 11. It shows how the strategies assign nodes to threads and in what order. The numbers on the bars indicate node IDs and they have been omitted on nodes with very short execution times to increase readability. The gray boxes in the BUSY schedule represent active waiting and white areas indicate sleeping threads. The small space between node executions is the time it takes for a thread to determine what the next executable node is and to check its dependencies.

Although the different schedules look very similar at first glance, the differences in their scheduling characteristics can be seen clearly: While the busy-waiting schedule has

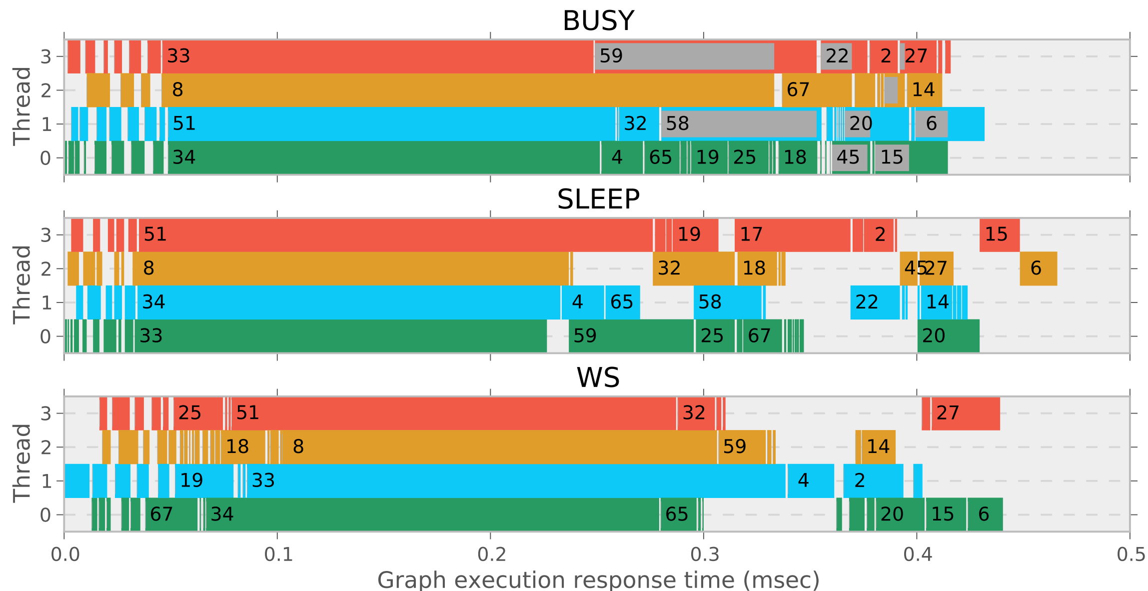

Figure 11. Typical schedule realizations with four threads

many active waiting boxes, the sleeping strategy has its threads sleeping while they wait for the resolution of their dependencies. Apart from that they look very similar because they use the same round-robin node prioritization scheme. The sleeping schedule has a longer total execution time because of the thread wake-up overhead.

In contrast to that, the WS strategy shows a totally different behavior: Instead of waiting for dependencies of the next node in the queue, it actively looks for other executable nodes and therefore executes many of the small nodes early. Sleeping in fact only occurs, when there are solely nodes available with unfinished dependencies. As can be seen, this is towards the end of the task graph execution.

The three schedules do not represent runs with the same audio packet but instead typical realizations of the schedules with execution times close to their respective average.

In section IV we computed an optimal schedule for four threads which predicted a task graph execution time of 327 µs. Our measurements came up with an average execution time of 452 µs, which is far from the optimal schedule. The inaccuracy of the simulation comes from the fact that it cannot take into account node assignment, thread management and dependency checking. Since these times are hard to measure, we chose the opposite approach and implemented our BUSY strategy in the RESCON simulation tool and compared the simulation result with our measurement. The results are shown in Figure 12. With 327 µs for the busy-waiting schedule, this is within 8% of the optimal schedule. With varying (and in advance unknown) execution times of individual nodes, computing an optimal schedule for every task graph iteration is not feasible. With minimal overhead, our BUSY scheduling with heuristic coarse screening of nodes by dependencies performs well.

Comparing the three scheduling strategies from average execution times alone is not sufficient, but with additional information from the histograms a winner can be selected. The busy-waiting approach not only has the lowest average execution times, it also makes sure that many nodes finish early, which is desirable. It comes at the cost of active waiting, but as the penalty is bearable in this case, it is clearly the best strategy. If wasting resources on waiting is not an option, work-stealing is a solid alternative.

VII. Related Work

In this paper, our research focuses on parallelizing a real-time audio application which uses a task graph as its main data structure. Related works address the parallel execution of *graph structures*, or the *parallelization of existing application* using tool support. In our approach, we used static analysis techniques in combination with a run-time profiler. Especially recent tool-assisted approaches also make use of a combination of static and dynamic analyses. For the generation of parallel source code they use parallel libraries such as OpenMP or the `java.util.concurrent` library in Java.

Parallel task graphs: There are several ways to parallelize a task graph, [13] uses a middleware approach while [14] is a real-time operating system that supports graph execution.

MCFlow [13] is a real-time, multicore-aware middleware for dependent task graphs. It uses a directed acyclic graph structure similar to the one used in *DJ Star*. The scheduling

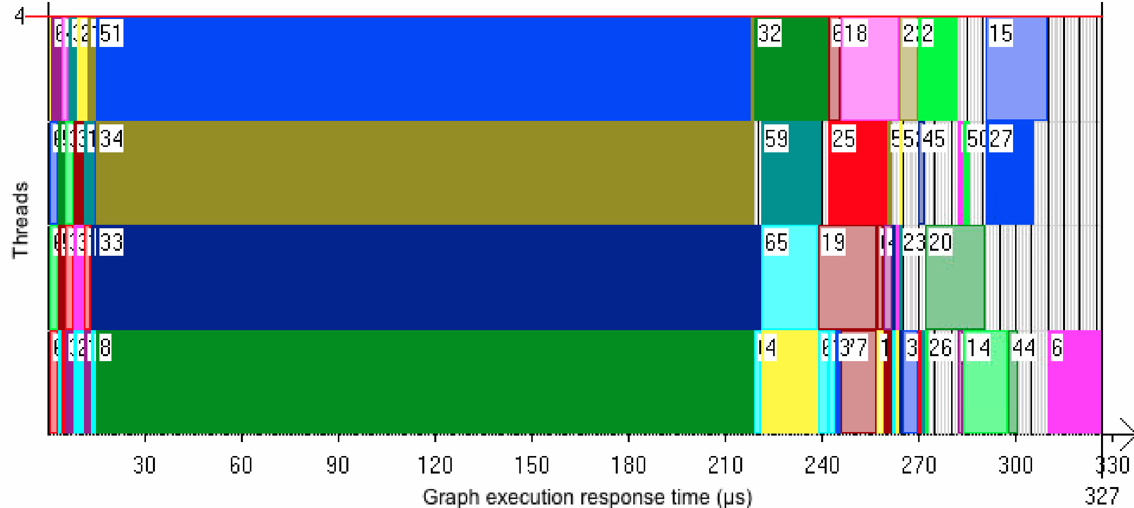

Figure 12. Simulation of the BUSY schedule

decision in MCFlow is taken offline while we use an online scheduling which enables us to dynamically load-balance the work items. In our case this makes much more sense, because the work is very imbalanced and a static procedure cannot take this into account.

Tesselation OS [14] is an experimental operating system that provides quality of service guarantees. This is very valuable for a real-time audio application, since missing a deadline leads to distortion of the audio signal. Programs running on Tesselation OS are characterized by a directed acyclic graph with plug-ins as nodes. In contrast to our application, the user is able to rearrange and exchange the nodes during run-time as needed.

But the main difference is that *DJ Star* has to run on general purpose hardware on broad scale, which currently binds it to Mac OS, Windows or Linux and makes the use of Tesselation OS impractical. For the future it would be interesting to migrate *DJ Star* to an operating system that provides real-time capabilities and test the miss rate for deadlines.

Parallelization support: Similar to our approach, different recent works use static and dynamic analyses to identify parallel potential in sequential programs. Embla 2 [15] is a tool which identifies task parallelism through a dynamic analysis. It provides information to the user about possible fork and join locations in the code. Furthermore, it highlights critical paths to assist in transforming the program into a parallel version.

Both [16] and [17] try to find pipeline parallelism in sequential programs. While Rul et al. [16] consider control flow and data dependencies to identify pipeline structures, Tournavitis et al. [17] additionally incorporate run-time profiling information in an attempt to obtain balanced pipeline stages. In contrast to our work, these works operate on general-purpose applications without real-time constraints and identify fork/join or pipeline parallelism.

VIII. Conclusion

In this paper, we analyzed a real-time compute-intensive audio application with over 700,000 lines of code for parallel potential. We manually analyzed it, identified three different parallelization strategies and evaluated them on realistic input data (four decks with different audio tracks).

The application we call *DJ Star* is a commercial DJ application that is used to load, manipulate, filter, and mix audio data. *DJ Star* serves as a good example for many other stream-intensive applications such as signal, audio, video, or image processing software. It contains the specific timing constraint that the computation of a single audio packet must not exceed 2.9ms. This makes obvious parallelization approaches impractical, and general guidelines and best-practice parallel patterns are not applicable without further investigation. With the analysis in this paper, we provide guidelines for stream-intensive applications that also contain rather complex software architectures in combination with timing constraints.

At first we manually reverse engineered the application's software architecture and manually located parallelization potential in a complex task graph structure. The computation of this task graph consumed 38% of the total run-time. From our analysis we designed the three different parallelization strategies *busy-waiting*, *thread-sleeping*, and *work-stealing*. We evaluated them composing a benchmark of 67 different filters and audio effects that imitate a typical use case for a DJ performance. Furthermore, we simulated the software architecture using the simulation tool RESCON [10] to determine a theoretical optimal schedule and used this in our evaluation as reference value for each strategy.

The key finding in our evaluation is that the *busy-waiting* strategy provided the best performance with the lowest average graph execution times of 327 µs and an average speedup of 2.40 with four threads on an 8-core machine. At the same time it produced the fewest timeouts with 5 out of 10,000 executions. This is remarkable, because in best-practice guidelines this strategy is seen as unfavorable, as it actively uses CPU cycles waiting for a certain condition. In our case, the processing cycles of *DJ Star* are rather short and reoccur at a high frequency, so the time it takes to pause a thread and wake it up as in the *thread-sleeping* strategy costs too much time. The *work-stealing* approach prevents certain sleeps and wake ups so it performs better than the sleeping strategy but it produces more timeouts than busy-waiting.

Acknowledgment

We thank Siemens Corporate Technology for their financial support. We also acknowledge the support of the Initiative for Excellence at the Karlsruhe Institute of Technology.

References

[1] H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16 – 23, 2005.

[2] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[3] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of parallel and distributed computing*, vol. 37, no. 1, pp. 55–69, 1996.

[5] T. Mattson and M. Wrinn, "Parallel programming: Can we please get it right this time?" in *Proceedings of the 45th Annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 7–11. [Online]. Available: http://doi.acm.org/10.1145/1391469.1391474

[6] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, and M. V. Zelkowitz, "Parallel programmer productivity: A case study of novice parallel programmers," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 35–. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.53

[7] D. Dobb's, "The state of parallel programming - the parallel programming landscape," 2012.

[8] Rane Corporation, "Rane TTM56S Mixer Product Page." [Online]. Available: http://dj.rane.com/products/ttm56s-mixer/

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[10] F. Deblaere, E. Demeulemeester, and W. Herroelen, "RESCON: Educational project scheduling software," *Computer Applications in Engineering Education*, vol. 19, no. 2, pp. 327–336, Jun. 2011.

[11] U. Gleim and T. Schuele, *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C#*. Dpunkt, 2011.

[12] V. Pankratius, A.-R. Adl-Tabatabei, and W. F. Tichy, *Fundamentals of Multicore Software Development*. CRC-Press, 2012.

[13] H.-M. Huang, C. Gill, and C. Lu, "MCFlow: A Real-Time Multi-core Aware Middleware for Dependent Task Graphs," *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, no. 3, pp. 104–113, Aug. 2012.

[14] J. A. Colmenares, I. Saxton, E. Battenberg, R. Avizienis, N. Peters, K. Asanovi, J. D. Kubiatowicz, and D. Wessel, "Real-time musical applications on an experimental operating system for multi-core processors," in *Proceedings of the International Computer Music Conference 2011*, no. August, 2011, pp. 216–223.

[15] J. Mak, K.-F. Faxén, S. Janson, and A. Mycroft, "Estimating and exploiting potential parallelism by source-level dependence profiling," in *Euro-Par 2010-Parallel Processing*. Springer, 2010, pp. 26–37.

[16] S. Rul, H. Vandierendonck, and K. De Bosschere, "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Computing*, vol. 36, no. 9, pp. 531–551, 2010.

[17] G. Tournavitis and B. Franke, "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 377–388.