2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium

Bus Access Design for Combined Worst and Average Case Execution Time Optimization of Predictable Real-Time Applications on Multiprocessor Systems-on-Chip

Jakob Rosén[1], Carl-Fredrik Neikter[1], Petru Eles[1], Zebo Peng[1], Paolo Burgio[2], Luca Benini[2]
[1]Department of Computer and Information Science, Linköping University, Sweden
[2]Department of Electronics, Computer Sciences and Systems, University of Bologna, Italy

***Abstract*—Optimization techniques for improving the average-case execution time of an application, for which predictability with respect to time is not required, have been investigated for a long time in many different contexts. However, this has traditionally been done without paying attention to the worst-case execution time. For predictable real-time applications, on the other hand, the focus has been solely on worst-case execution time optimization, ignoring how this affects the execution time in the average case. In this paper, we show that having a good average-case delay can be important also for real-time applications for which predictability is required. Furthermore, for real-time applications running on multiprocessor systems-on-chip, we present a technique for optimizing the average case and the worst case simultaneously, allowing for a good average-case execution time while still keeping the worst case as small as possible.**

I. Introduction and Related Work

For real-time systems, correctness of a program not only depends on the produced computational results, but also on its ability to deliver these on time, according to specified time constraints. Therefore, for a real-time application, predictability with respect to time is of uttermost importance. The obvious example is safety-critical hard real-time systems, such as medical and avionic applications, for which failure to meet a specified deadline not only renders the computations useless, but also can have catastrophic consequences. However, predictability is getting more and more desirable for other classes of embedded applications, for instance within the domains of multimedia and telecommunication, for which QoS guarantees are desired [1]. As these kinds of applications grow more complex, they also require more computational power in terms of hardware resources. In order to satisfy these demands, multi-core systems on a single chip are used to an increasing extent [2].

To achieve predictability with respect to time, various techniques are applied, assuming that the worst-case execution time (WCET) of every task is known. A lot of research has been carried out within the area of worst-case execution time analysis [3]. However, according to the proposed techniques, each task is analyzed in isolation, as if it was running on a monoprocessor system. Consequently, it is assumed that memory accesses over the bus take a constant amount of time to process, since no bus conflicts can occur. For multiprocessor systems with a shared communication infrastructure, however, transfer times depend on the bus load and are therefore no longer constant, causing the traditional methods to produce incorrect results [4], [5]. The main obstacle when performing timing analysis on multiprocessor systems is that the scheduling of tasks assumes that their worst-case execution times are known. However, to calculate these worst-case execution times, knowledge about the task schedule is required. The traditional method of separating WCET analysis and task scheduling no longer works, and new approaches are required. We have previously proposed a novel technique to achieve predictability on multiprocessor systems by doing WCET analysis and scheduling simultaneously [6].

The worst-case program path is, for most applications, taken very seldom. This generally leads to much longer execution times than what can be expected on average, resulting in a gap between the worst-case global delay (WCGD) and the average-case global delay (ACGD). Therefore, when designing periodic systems, there will be a significant interval of time after the program has finished until the next period starts. During this time interval, the processors are free to be used for anything, as long as they are ready and free at the start of the new period. Thus, instead of just letting them be idle, doing nothing, we can utilize this slack for performing computations not requiring strict predictability, or we can just shut off the processors to save energy. Consequently, it can be of great interest that the average-case global delay is as short as possible, even for hard real-time systems.

Consider the hard real-time application in Figure 1a, composed of the three tasks τ_1, τ_2 and τ_3 running on two processors. The application is periodically executed with a period equal to the worst-case global delay. After finishing the last computation, the processors are powered off, to save energy, until the start of the next period. Consequently, in the average case, the processors are shut off between the time instants ACGD and WCGD, and, since the goal is to reduce the power consumption as much as possible, we would like to maximize this time interval. However, we also need to have a short period and, therefore, the worst-case global delay must remain small. Hence, optimizing for the average case without caring for the worst case is not suitable for these kind of systems. On the other hand, if we optimize for the ACGD while also making sure that the WCGD is kept at a near-minimum, it is possible to benefit from a substantial reduction of power consumption while

1080-1812/11 $26.00 © 2011 IEEE
DOI 10.1109/RTAS.2011.35

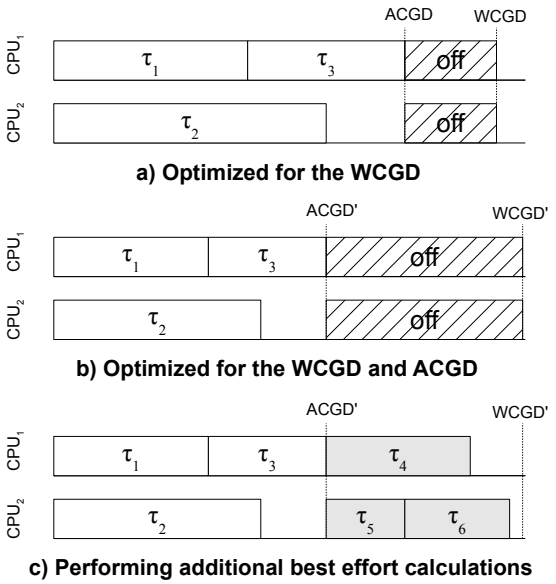

Figure 1. Motivational Example For a Hard Real-Time System

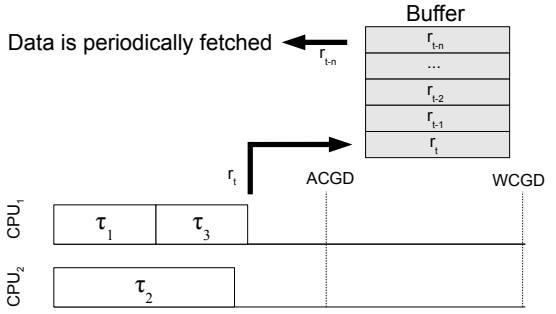

Figure 2. Motivational Example For a Buffer-Based System

extending the application period only marginally. This is exactly the case in Figure 1b, where it can be seen that the energy consumed for running the application is reduced, but since the WCGD is increased only by a small amount, the application period can still be kept low.

The interval between the end time of the application and the WCGD can, obviously, also be used for other purposes than switching off the processors. In Figure 1c, we have used the remaining time, after the end of the application, for running best effort calculations, represented by the tasks τ_4, τ_5 and τ_6. These tasks can, if needed, be preempted at the end of the application period.

Another example can be found in Figure 2. The application, consisting of the three tasks τ_1, τ_2 and τ_3 running on two processors, is writing a produced result to a FIFO buffer at the end of its execution. An external consumer is periodically reading data from the other end of the buffer. A small ACGD allows for high data rates, but in order to guarantee a minimum rate and to help the system designer dimension the buffer, the WCGD must be known and preferably be small as well.

Optimization techniques for improving the average-case

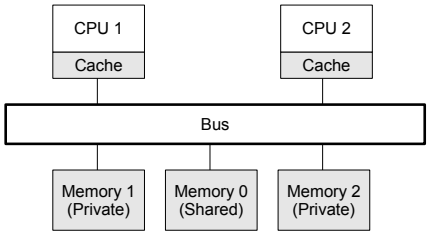

Figure 3. Hardware model

execution time of an application, for which predictability with respect to time is not required, have been investigated in nearly every scientific discipline involving a computer. However, this has traditionally been done without paying attention to the worst-case execution time. For predictable applications, on the other hand, the focus has been solely on worst-case execution time optimization, which still is a hot research topic [7][6]. The main contribution of this paper is the combination of these two concepts and, to the best of our knowledge, this is the first time it has ever been done within the context of achieving predictability.

II. System and Application Model

A. Hardware Architecture

As hardware platform, we have considered a multiprocessor system-on-chip with a shared communication infrastructure, as shown in Figure 3, typical for the new generation of multiprocessor system-on-chip designs [8]. Each processor has its own cache for storing data and instructions, and is connected to a private memory via the bus. For interprocessor communication, a shared memory is used. All memory accesses to the private memories are cached, as opposed to accesses to the shared memory which, in order to avoid cache coherence problems, are not cached. All memory devices are accessed using the same, shared bus. However, in the case of private memory accesses, the bus is used only when an access results in a cache miss.

B. Application Model

The functionality of a software application is captured by a directed acyclic task graph, $G(\Pi, \Gamma)$. Its nodes represent computational tasks, and the edges represent data dependencies between them. A task cannot start executing before all its input data is available. Communication between tasks mapped on the same processor is performed by using the corresponding private memory, and is handled in the same way as memory requests during the execution of a task. Interprocessor communication, so called *explicit communication*, is done via the shared memory and is modeled as two communication tasks - one for transmitting and one for receiving - in the task graph. The transmitting communication task is assigned to the same processor as the task that is sending data to the shared memory and, similarly, the receiving communication task is assigned to

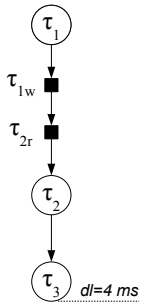

Figure 4. Task graph

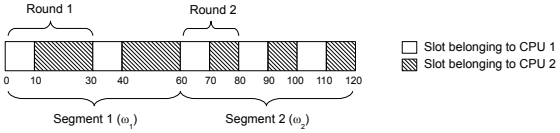

Figure 5. Example of a bus schedule

the processor fetching the same data. An example is shown in Figure 4 where τ_{1w} and τ_{2r} represent the transmitting and receiving task, respectively.

A computational task cannot communicate with other tasks during its execution, which means that it will not access the shared memory. However, the task is accessing data from the private memory and program instructions are continuously fetched. Consequently, the bus is accessed every time a cache miss occurs, resulting in what we define as *implicit communication*. As opposed to explicit communication, implicit communication has not been taken into account in previous approaches for system-level scheduling and optimization of real-time applications [9], [10].

The task graph has a deadline which represents the maximum allowed execution time of the entire application, known as the maximum global delay. Individual tasks can have deadlines as well. The example task graph in Figure 4 has a global delay of 4 milliseconds. The application is assumed to be running periodically, with a period greater than or equal to the application deadline.

C. Bus Model

A precondition for predictability is to use a predictable bus architecture. Therefore, we are using a TDMA-based bus arbitration policy, suitable for modern system-on-chip designs with QoS constraints [11], [12], [1], [13].

The behavior of the bus arbiter is defined by the *bus schedule*, consisting of sequences of slots. Each slot is owned by exactly one processor, and has an associated start and end time pair. Between these two time instants, only the processor owning the slot is allowed to use the bus. A bus schedule is divided into *segments*, and each segment consists of a *round*, that is, a sequence of slots, that is repeated periodically. See Figure 5 for an example.

The bus arbiter stores the bus schedule in a dedicated memory, and grants access to the processors accordingly. If CPU_i requests access to the bus in a time interval belonging to a slot owned by a different processor, the transfer will be

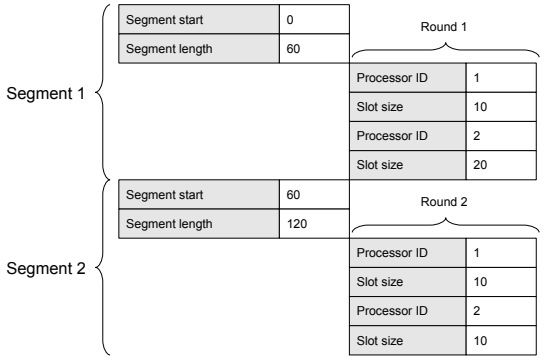

Figure 6. Bus schedule table representation

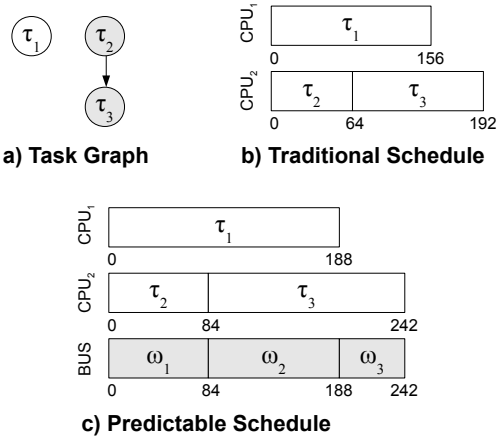

Figure 7. Overall Approach Example

delayed until the start of the next slot with owner CPU_i. A bus schedule is defined for one period of the application, and is then repeated periodically. A table representation of the bus schedule in Figure 5 is depicted in Figure 6.

To limit the required amount of memory on the bus controller, a TDMA round can be subject to various constraints. A common restriction is to let every processor own, at most, a specified number of slots per round. Also, one can let the sizes be the same for all slots of a certain round, or let the slot order be fixed [6]. The algorithms presented in this paper work regardless of what restrictions are imposed with respect to the TDMA round.

III. Preliminaries

For a task running on a multiprocessor system, as described in Section II-A, the problem for achieving predictability is that the duration of a bus transfer depends on the bus congestion. Since bus conflicts depend on the task schedule, WCET analysis cannot be performed before that is known. However, task scheduling traditionally assumes that the worst-case execution times of the tasks are already calculated. To solve this circular dependency, we have developed an approach based on the following principles [6]:

1) A TDMA-based bus access policy (Section II-C) is used for arbitration. The bus schedule, created at design time,

is enforced during the execution of the application.
2) The worst-case execution time analysis is performed with respect to the bus schedule, and is integrated with the task scheduling process, as described in Figure 8.

We illustrate our overall approach with a simple example. Consider the application in Figure 7a. It consists of three tasks; τ_1, τ_2 and τ_3 mapped on two processors. The static cyclic scheduling process is based on a list scheduling technique [14] and is performed in the outer loop described in Figure 8. Let us, as is done traditionally, assume that worst-case execution times have been obtained using techniques where each task is considered in isolation, ignoring conflicts on the bus. These calculated worst-case execution times are 156, 64, and 128 time units for τ_1, τ_2, and τ_3 respectively. The deadline is set to 192 time units, and would be considered as satisfied according to traditional list scheduling, using the already calculated worst-case execution times, as shown in Figure 7b. However, this assumes that no conflicts, extending the bus transfer durations (and implicitly the memory access times), will ever occur on the bus. This is, obviously, not the case in reality and thus results obtained with the previous assumption are wrong.

In our predictable approach, the list scheduler will start by scheduling the two tasks τ_1 and τ_2 in parallel, with start time 0, on their respective processor (line 2 in Figure 8). However, we do not yet know the end times of the tasks, and to gain this knowledge, worst-case execution time analysis has to be performed. In order to do this, a bus schedule which the worst-case execution times will be calculated with respect to (line 6 in Figure 8) must be selected. This bus schedule is, at the moment, constituted by one bus segment ω, as described in Section II-C. Given this bus schedule, worst-case execution times of tasks τ_1 and τ_2 will be computed (line 7 in Figure 8). Based on this output, new bus schedule candidates are generated and evaluated (lines 5-8 in Figure 8), with the goal of obtaining those worst-case execution times that lead to the shortest possible worst-case global delay of the application.

Assume that, after selecting the best bus schedule, the corresponding worst-case execution times of tasks τ_1 and τ_2 are 167 and 84 respectively. We can now say the following:

- Bus segment ω_1 is the first segment of the bus schedule, and will be used for the time interval 0 to 84.
- Both tasks τ_1 and τ_2 start at time 0.
- In the worst case, τ_2 ends at time 84 (the end time of τ_1 is still unknown, but it will end later than 84).

Now, we go back to step 3 in Figure 8 and schedule a new task, τ_3, on processor one. According to the previous worst-case execution time analysis, task τ_3 will, in the worst case, be released at time 84, scheduled in parallel with the remaining part of task τ_1. A new bus segment ω, starting at time 84, will be selected and used for analyzing task τ_3. For task τ_1, the already fixed bus segment ω_1 is used for the

```
01: θ=0
02: while not all tasks scheduled
03:   schedule new task at t ≥ θ
04:   Ψ=set of all tasks that are active at time t
05:   repeat
06:     select bus segment ω for the
           time interval starting at t
07:     determine the WCET of all tasks in Ψ
08:   until termination condition
09:   θ=earliest time a task in Ψ finishes
10: end while
```

Figure 8. Overall Approach

time interval between 0 and 84, after which the new segment ω is used. Once again, several bus schedule candidates are evaluated, and finally the best one, with respect to the worst-case global delay, is selected. Assume that the segment ω_2 is finally selected, and that the worst-case execution times for tasks τ_1 and τ_3 are 188 and 192 respectively, making task τ_3 end at 276. Now, ω_2 will become the second bus segment of the application bus schedule, ranging from time 84 to 188, and this part of the bus schedule will be fixed. Now, we repeat the same procedure with the remaining part of τ_3 (which now ends at time 242 instead of 276, since ω_3 assigns all bus bandwidth to CPU_2). The final, predictable schedule is shown in Figure 7c, and leads to a WCGD of 242.

An outline of the algorithm can be found in Figure 8. We define Ψ as the set of tasks active at the current time t, and this is updated in the outer loop. In the beginning of the loop, a new bus segment ω, starting at t, is generated and the resulting bus schedule candidate is evaluated with respect to each task in Ψ. Based on the outcome of the WCET analysis, the bus segment ω is improved for each iteration. The bus segments previously generated before time t remain unaffected. After selecting the best segment ω, θ is set to the end time of the task in Ψ that finished first. The time t is updated to θ and we continue with the next iteration of the outer loop.

Since our approach requires knowledge about not only the number of cache misses for a certain program path, but also their location with respect to time, this must be taken into consideration by the WCET analysis on line 7 in Figure 8. Consequently, we must search through all feasible program paths and match each possible bus transfer to slots in the actual bus schedule, keeping track of exactly when a bus transfer is granted the bus in the worst case. Since the number of program paths grows exponentially, the number of possible search paths in the control flow graph quickly becomes very large. Fortunately, efficient search-tree pruning techniques dramatically reduce the search space, and allow for quick analyses also for big and complex tasks.

Given whatever TDMA bus schedule, our WCET analysis calculates a *safe* corresponding worst-case execution time. An integrated worst-case cache miss analysis, supporting set associative instruction and data cache models of various sizes, is used in order to collect information about the

possible bus transfers. The analysis technique is applicable to both compositional and noncompositional hardware architectures, as defined by Wilhelm et al. [15], and is of the same computational complexity as traditional methods. We refer to our previous work for an extensive coverage of the used WCET analysis framework [16][6].

IV. Average-Case Execution Time Estimation

When calculating the WCET of a task, one tries to find the worst-case program path with respect to the specified bus schedule. The bus optimization algorithm then locates where, with respect to the worst-case program path, to allocate bandwidth. This technique is not directly applicable to average-case execution time (ACET) analysis, since there is generally no particular program path corresponding to the ACET of a task.

To evaluate how good a bus schedule is from the point of view of the ACET, the application has to be executed a large number of times so that the end time of each run can be recorded and used to calculate a mean. This is a rather time-consuming process and, therefore, using this method repetitively inside an optimization loop leads to excessively long analysis times. Also, in order for the optimization algorithm to know where to allocate bus bandwidth for a certain task, the locations, with respect to time, of the cache misses for an average execution of the task have to be approximated. We solve these two problems by using a histogram-based technique, where simulation data is used to create task profiles then given as input to the optimization algorithm.

In order to build the memory access histogram, N sets of input data are generated for each task. This data is randomized with respect to a distribution representing typical input patterns for the particular task in question. Every task is then simulated, in isolation, N times and, for each simulation, a trace file containing the locations of the cache misses is generated. Using this information, we want to find out where, in time, cache misses are most likely to occur so that bus bandwidth can be assigned accordingly. This is done by building, for each task, a histogram over bus accesses in time (excluding the time spent using the bus), with respect to all N simulations.

Figure 9 shows an example of a histogram based on 1000 simulations. The y-value of the histogram denotes how many of the N simulations of the task requested the bus during the time interval represented by the corresponding x-value. For instance, in this example, it can be seen that all simulations request the bus at the very start of the task due to instruction cache misses. During the time regions denoted by t_1, t_2, t_3 and t_4, most simulations request the bus. Consequently, making sure that the task gets a lot of bandwidth during these time periods is most likely a good idea, from the point of view of the ACET.

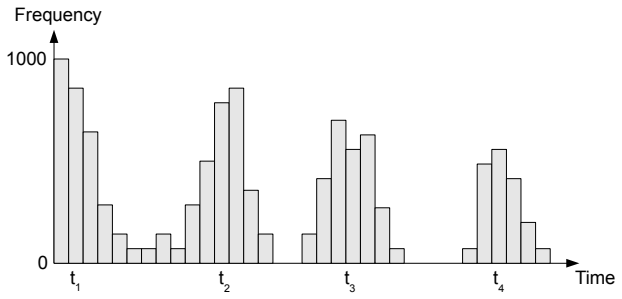

Figure 9. Memory Access Histogram

Given the histogram and a specified bus schedule, we can obtain an estimation of the average-case task execution time.

By using the frequency data on the y-axis, a hypothetical program path can be constructed, corresponding to the average-case memory access pattern. To get an average-case execution time estimation, we can then apply our technique outlined in the last paragraph of Section III, using this hypothetical program path.

We want to design a bus schedule that produces a good ACGD. However, since we want to keep the WCGD small, we must also consider the worst-case program path during the optimization process. This requires a new optimization technique, which will be outlined in the next section.

V. Combined Average and Worst Case Optimization Approach

We assume that the steps in Figure 8 have been carried out, and that we have the result in the form of a task schedule, s_0^{worst}, and a bus schedule, B_0, corresponding to the smallest possible WCGD. These are, together with a designer-specified limit on the maximum allowed WCGD and the memory access histogram data for the tasks $\tau_i \in G(\Pi, \Gamma)$, taken as input parameters to our combined optimization approach, as illustrated in Figure 10. As output from the algorithm, a bus schedule B_{final}, optimized for both ACGD and WCGD, is returned together with the final worst-case task schedule $s_{\text{final}}^{\text{worst}}$ and average-case task schedule length ACGD_{est}.

In the first step of our algorithm in Figure 10, the average-case schedule s_0^{avg} is calculated with respect to B_0. Then an iterative function, denoted as `improve` on line 4 in Figure 10, tries to improve the bus schedule with respect to both the average and the worst case. The termination condition is reached when no more improvements can be found, and the algorithm then exits and returns the best bus schedule B_{final} and corresponding worst-case task schedule $s_{\text{final}}^{\text{worst}}$.

VI. Bus Access Optimization for ACGD and WCGD

The `improve` function (line 4 in Figure 10) takes as input parameters a bus schedule B_k, the initial worst-case task schedule s_0^{worst}, the initial average-case task schedule s_0^{avg} and WCGD_{max}. As output, we get the improved bus

Algorithm: **Combined ACGD and WCGD optimization**

Input: - Initial bus schedule B_0
- Initial worst-case schedule s_0^{worst}
- Maximum worst-case global delay WCGD_{max}
- Memory access histograms of the tasks $\tau_i \in G(\Pi, \Gamma)$

Output: - The resulting bus schedule B_{final}
- The resulting worst-case task schedule $s_{\text{final}}^{\text{worst}}$

01: Calculate initial average-case schedule s_0^{avg}
02: $k = 0$
03: **repeat**
04: $[B_{k+1}, s_{k+1}^{\text{worst}}, s_{k+1}^{\text{avg}}]$ = improve(B_k)
05: k=k+1
06: **until** termination condition
07: $B_{\text{final}} = B_k$
08: $s_{\text{final}}^{\text{worst}} = s_k^{\text{worst}}$

Figure 10. Combined Optimization Approach

schedule B_{k+1} together with the corresponding worst-case task schedule s_{k+1}^{worst} and average-case task schedule s_{k+1}^{avg}. The goal is to modify the bus schedule so that the ACGD of the application is reduced, while the WCGD is increased as little as possible. To do so, the effects on both the ACGD and WCGD have to be considered for each possible modification. However, performing average-case and worst-case execution time analysis with respect to several bus schedule candidates is time-consuming. Therefore, it is desirable to identify the most interesting parts of the bus schedule, where a modification is likely to result in positive effects for the global delay, and then perform execution time analysis with respect to modifications of these parts only. Consequently, we start the `improve` function by investigating which parts of the bus schedule to modify for a decreased ACGD, without initially considering the effects on the WCGD. Only the most interesting parts are then investigated with respect to both the ACGD and WCGD.

A. Task and Bus Segments

The first step of the `improve` function is to generate the average-case task schedule s_k^{avg} by performing ACET analysis (Section IV), for each task, with respect to the bus schedule B_k. From the execution time analysis, we can extract interesting properties, such as bus transfer times and the number of memory accesses, of certain time intervals, and these properties are then used to determine how much the corresponding parts of the bus schedule can be improved with respect to the ACGD (and later also how to modify the bus schedule).

In order to find suitable time intervals, we first divide both the bus schedule B_k and the average-case task schedule s_k^{avg} into segments. For this, we distinguish between two different kind of segments: *task segments* and *bus segments*. A task segment is defined as the longest time interval in which a specific set of tasks, with respect to a specific task schedule, are executing concurrently. Every task schedule can be seen as a disjunctive set of task segments, Ξ. This is a natural division, since the execution time analysis operates on these kind of sets. Bus segments are defined just as in Section II-C, i.e. as intervals of the bus schedule where the same

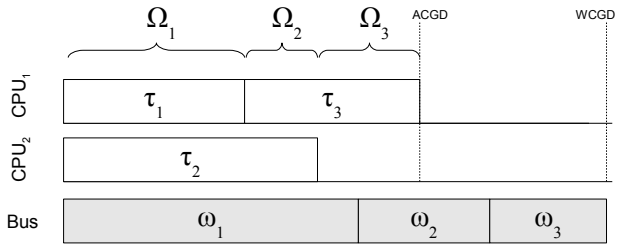

Figure 11. Division of an Application Into Segments

TDMA round is repeated. Consequently, the bus schedule can be regarded as a disjunctive set of bus segments, B.

Figure 11 shows a task schedule for an application with three tasks mapped on two processors, and the corresponding bus schedule. The task schedule is divided into three task segments: Ω_1, Ω_2 and Ω_3. The three dashed areas of the bus schedule represent the bus segments ω_1, ω_2 and ω_3. What we, initially, would like to do is to identify the areas of the bus schedule, represented by task segments and bus segments, for which modifications can result in the most beneficial change of the ACGD. In order to perform this identification, we start with an investigation of how the bus bandwidth is distributed to the task segments and bus segments, in the average case.

B. Bus Bandwidth Distribution Analysis

Based on the information from the average-case execution time analysis, for each task segment $\Omega_i \in \Xi$ and bus segment $\omega_j \in B$, we want to determine the following:

1) The desired bus bandwidth, that, when given to this particular interval of the bus schedule, minimizes the global delay in the average case. For a specific task segment $\Omega_i \in \Xi$, this bandwidth is denoted $\overline{P}_{\Omega_i}$ and is a vector of n elements, $\overline{P}_{\Omega_i} = p_{\Omega_i}(1), \ldots, p_{\Omega_i}(n)$, where n is the number of processors in the system and each element represents the desired bandwidth for the corresponding processor. Similarly, $\overline{P}_{\omega_j} = p_{\omega_j}(1), \ldots, p_{\omega_j}(n)$ is the corresponding vector for a bus segment $\omega_j \in B$. The bandwidth is represented as the fraction of the total bus bandwidth, thus satisfying:
$$\sum_k p_{\Omega_i}(k) = 1, \qquad \sum_k p_{\omega_j}(k) = 1$$
Detailed descriptions for how to perform these calculations can be found in AppendixA-A and A-B.
2) The bandwidth currently given to each processor during the specific interval $\Omega_i \in \Xi$ and $\omega_j \in B$, represented by the vectors $\overline{P}_{\Omega_i}^{bus} = p_{\Omega_i}^{bus}(1), \ldots, p_{\Omega_i}^{bus}(n)$ and $\overline{P}_{\omega_j}^{bus} = p_{\omega_j}^{bus}(1), \ldots, p_{\omega_j}^{bus}(n)$. As for the desired bandwidth, the elements are representing fractions of the total bandwidth, therefore summing up to one. Section A-C describes how these two vectors are calculated.

For a specific task segment $\Omega_k \in \Xi$, let $\alpha_{\Omega_k} \in [1..n] \subseteq \mathbb{Z}$, where n is the number of processors, denote the processor

on which the task on the critical path, with respect to the average-case task schedule, is executed during that particular time interval. For every task segment, there is always one such processor. Let us define the scalar $p^{\Delta}_{\Omega_k}$, for a task segment $\Omega_k \in \Xi$, as the difference between the desired task bandwidth and the provided bus bandwidth for processor α_{Ω_k}. That is, $p^{\Delta}_{\Omega_k} = p_{\Omega_k}(\alpha_{\Omega_k}) - p^{bus}_{\Omega_k}(\alpha_{\Omega_k})$.

Since many task segments can overlap a bus segment, several different processors can execute tasks that are on the (same) critical path during the particular time interval represented by the bus segment. Hence, let A_{ω_k} be the set of processors α_{ω_k} that are running tasks on the critical path during the interval represented by bus segment $\omega_k \in B$. We then define $p^{\Delta}_{\omega_k}$ as:

$$p^{\Delta}_{\omega_k} = max\Big(\bigcup_{\alpha_i \in A_{\omega_k}} \big(p_{\omega_k}(\alpha_i) - p^{bus}_{\omega_k}(\alpha_i)\big)\Big) \qquad (1)$$

A high $p^{\Delta}_{\Omega_i}$ for a task segment $\Omega_i \in \Xi$ or $p^{\Delta}_{\omega_k}$ for a bus segment $\omega_j \in B$ means that the corresponding interval of the bus schedule has room for improvement with respect to the average-case global delay. Therefore, time intervals with a high corresponding $p^{\Delta}_x, x \in (\Xi \cup B)$ are interesting from an optimization point of view, whereas intervals with a low $p^{\Delta}_x, x \in (\Xi \cup B)$ do not need further investigation. We can now limit the search space by just looking at parts of the bus schedule with a corresponding $p^{\Delta}_x, x \in (\Xi \cup B)$ exceeding a specified threshold.

If we wanted to just optimize for the average case, we would start by modifying the region represented by the segment (task or bus) with the highest $p^{\Delta}_x, x \in (\Xi \cup B)$. However, a large decrease in ACGD is not necessarily good, if that makes the WCGD increase too big. When deciding which region of the bus schedule to improve, one must also take into account the effect on the WCGD. In fact, what we want to improve is the ratio between average-case improvement and worst-case extension, with respect to the global delay. In order for our optimization algorithm to decide how good a bus schedule is for both the ACGD and WCGD, a cost function is specified in the next section.

C. Cost Function

We denote with $length(s)$ the length of schedule s. Let $s^{\text{worst}}_{\text{old}}$ and $s^{\text{avg}}_{\text{old}}$ be worst-case and average-case schedules, for the same application, generated with respect to a bus schedule B_{old}. After creating a new bus schedule B_{new}, by modifying a suitable interval of B_{old}, we obtain updated task schedules $s^{\text{worst}}_{\text{new}}$ and $s^{\text{avg}}_{\text{new}}$. If an improvement was made, the task schedules will satisfy $length(s^{\text{avg}}_{\text{new}}) < length(s^{\text{avg}}_{\text{old}})$, and $length(s^{\text{worst}}_{\text{new}}) > length(s^{\text{worst}}_{\text{old}})$ (since the WCGD is expected to grow[1]). If the bus schedule B_{new} does not lead to an improvement with respect to the ACGD, it is discarded and not considered further by the optimization algorithm. Provided that the new bus schedule B_{new} actually results in an improvement, a good measure of how good the bus modification is can be given by the ratio:

$$q_{s^{\text{worst}}_{\text{new}}, s^{\text{avg}}_{\text{new}}, s^{\text{worst}}_{\text{old}}, s^{\text{avg}}_{\text{old}}} = \frac{\text{length}(s^{\text{avg}}_{\text{old}}) - \text{length}(s^{\text{avg}}_{\text{new}})}{\text{length}(s^{\text{worst}}_{\text{new}}) - \text{length}(s^{\text{worst}}_{\text{old}})} \qquad (2)$$

Consequently, a suitable cost function for our optimization algorithm can be expressed as:

$$C(s^{\text{worst}}_{\text{new}}, s^{\text{avg}}_{\text{new}}, s^{\text{worst}}_{\text{old}}, s^{\text{avg}}_{\text{old}}) = -q_{s^{\text{worst}}_{\text{new}}, s^{\text{avg}}_{\text{new}}, s^{\text{worst}}_{\text{old}}, s^{\text{avg}}_{\text{old}}} \qquad (3)$$

With respect to this cost function, we can now evaluate a set of bus schedule candidates and choose the best one.

D. Bus Schedule Optimization

We create a new bus schedule candidate $B_{k'}$ from bus schedule B_k (taken as input parameter on line 4 in Figure 10) by modifying the part of B_k corresponding to the time interval represented by a task segment $\Omega_i \in \Xi$ or bus segment $\omega_j \in B$. To calculate the cost of $B_{k'}$, we must compute the schedules $s^{worst}_{k'}$ and $s^{avg}_{k'}$. This is done by invoking the execution time analysis framework twice, for the entire application, and that is relatively costly from a computation-time perspective. Therefore, as stated previously, the solution is to limit the search space and thus only generate candidates that are likely to perform good. It can be assumed that improving areas corresponding to segments with a low $p^{\Delta}_{\Omega_i}$ or $p^{\Delta}_{\omega_j}$, for $\Omega_i \in \Xi$ and $\omega_j \in B$ respectively, will not lead to the best results. Therefore, we define Ξ' as the set of the t task segments $\Omega_i \in \Xi$ which have the greatest corresponding $p^{\Delta}_{\Omega_i}$ values. Similarly, B' is defined as the set of b bus segments $\omega_j \in B$ with the greatest corresponding $p^{\Delta}_{\omega_i}$. The $t+b$ segments in $\Xi' \cup B'$ are selected for further investigation. High t and b values, set by the designer, allow the algorithm to evaluate more bus schedule candidates, but at the expense of computation time.

For each segment in $\Xi' \cup B'$, we generate several bus schedule candidates and evaluate them with respect to the cost function defined in Equation 3. When no more bus schedule candidates are left to evaluate for any segment in $\Xi' \cup B'$, the candidate associated with the lowest cost is kept and returned as bus schedule B_{k+1} (line 4 in Figure 10).

The first bus schedule candidate $B_{k'_0}$ for a specific segment in $\Xi' \cup B'$ is generated by inserting a new bus segment into the previously generated bus schedule B_k. This new bus segment is constituted by a TDMA round r, generated so that the bus bandwidth during the corresponding interval is assigned according to the desired bus bandwidth $\overline{P}_{\Omega_i}$ or $\overline{P}_{\omega_j}$, depending on if the segment being investigated is a task segment $\Omega_i \in \Xi'$ or a bus segment $\omega_j \in B'$. With respect to the bus schedule candidate $B_{k'_0}$, the schedules $s^{\text{worst}}_{k'_0}$ and $s^{\text{avg}}_{k'_0}$ are generated and $B_{k'_0}$ is then evaluated according to the cost function in Equation 3. To create the next bus

[1]For the special case when $length(s^{\text{worst}}_{\text{new}}) \leq length(s^{\text{worst}}_{\text{old}})$, the ratio in Equation 2 is set to ∞.

```
01: Perform an average-case execution time analysis.
02: Divide the resulting task schedule into a set of
    task segments Ξ.
03: Calculate current and desired bus bandwidth,
    P̄_{Ω_i} and P̄_{ω_j} with respect to the ACGD only, for
    each task segment and bus segment.
04: Calculate Ξ' and B'.
05: For each element in Ξ' ∪ B', generate a set of
    bus schedule candidates and evaluate them
    according to the cost function in Equation 3.
06: Return the candidate that generates the lowest cost,
    while keeping the WCGD below WCGD_max.
```

Figure 12. The `improve` Function

schedule candidate $B_{k'_i}$ (where now, in this case, $i = 1$) for the same segment, round r is modified according to the outcome of the execution time analysis, by assigning more bandwidth to the processor on the critical path. The cost is then recalculated. Other modifications, such as slot order permutations, can also be carried out depending on the restrictions imposed on TDMA complexity. The procedure of improving round r - each improvement resulting in a new bus schedule candidate - is repeated a specified number of times or until no further improvements are found, and then the next segment in $\Xi' \cup B'$ is investigated. The best bus schedule candidate $B_{k'}$ is then chosen as the new bus schedule B_{k+1} for the application, and the function returns. The `improve` function is summarized in Figure 12.

Note that adding new bus segments will increase the complexity of the bus schedule. Since the memory on the bus arbiter is limited, there might be a limit for how many bus segments we can allow. Once this maximum number of bus segments is reached, we cannot increase the number of segments of the bus schedule without first deleting at least one, already existing, bus segment. Therefore, immediately after inserting the new bus segment, resulting in the bus schedule candidate $B_{k'_0}$, and before making any improvements to the corresponding round r constituting it, we evaluate the effect of merging every pair of consecutive bus segments in the bus schedule using the ACGD and WCGD analyses and computing the resulting cost. The best merge is then kept, and we continue by generating more bus schedule candidates $B_{k'_i}$ (by trying to improve r, as usual). Note that this is only a problem when improving the bus schedule with respect to task segments $\Omega_i \in \Xi'$, since improving with respect to bus segments $\omega_j \in B'$ does not increase the number of bus segments.

VII. Experimental Results

We have evaluated our framework using an extensive set of generated C programs. The programs were constructed with respect to randomized task graphs consisting of between 20 and 200 tasks, mapped on 2 to 8 processors. The individual tasks were generated according to control flow graphs corresponding to programs for commonly used computations such as sorting, searching, matrix multiplications and DSP processing. In total, 8000 applications were generated and evaluated. To calculate the memory access histograms, as described in Section IV, 1000 simulations where carried out for each task.

As simulation environment, we have used the MPARM multiprocessor cycle-accurate simulator from Bologna University [17], configured according to the hardware model in Section II-A, using 8 ARM7 cores running at 200 MHz. An AMBA AHB-compliant bus arbiter, enforcing the bus model in Section II-C, was implemented and incorporated into the simulation framework. The bus speed was set to 100 MHz, resulting in a memory access taking 13 CPU clock cycles to serve. In order to restrict the amount of memory on the controller, we imposed the following restrictions on TDMA round complexity:

1) A processor can own at most one slot in a TDMA round.
2) The slot order is fixed, and cannot be changed during the optimization procedure.

The values of t and b, described in Section VI-D, were set to 100 and 50 respectively. We also limited the total number of bus segments allowed in the bus schedule to 1000.

Using the approach outlined in Section III, for each of the applications, we started by generating a bus schedule minimizing the worst-case global delay, completely ignoring the average case. Let us denote this initial, minimized, worst-case global delay by WCGD_0, and let ACGD_0 be the ACGD calculated with respect to the same bus schedule. The bus schedule, optimized for the worst case, and the corresponding worst-case task schedule were then sent as input parameters to the algorithm outlined in Figure 10, together with the generated memory access histograms for each task in the application task graph. A maximum allowed WCGD was also supplied.

We now investigated how much the ACGD can be decreased, given a maximum allowed increment (with respect to WCGD_0) of the resulting WCGD. For all applications, we performed the optimization procedure three times, allowing WCGD increments of 1%, 5% and 10% respectively. For each of these allowed increments, a corresponding average ACGD improvement was calculated. The result is found in Figure 13. For instance, for two processors, accepting an 1% extension with respect to WCGD_0 leads to an average ACGD improvement of 13.2%. Accepting a greater WCGD increment naturally results in a more substantial ACGD reduction. It can be observed that using a lower number of processors allows for a higher ACGD decrement, with respect to ACGD_0. This is explained by the fact that fewer competing processors leave more room for tailoring the bus schedule for a specific processor, allowing for a more flexible design.

In a second experiment, we investigated how optimizing for the ACGD, without considering the WCGD at all, affects the latter. The idea is to show that optimizing only for the ACGD leads to unreasonably high worst-case global delays,

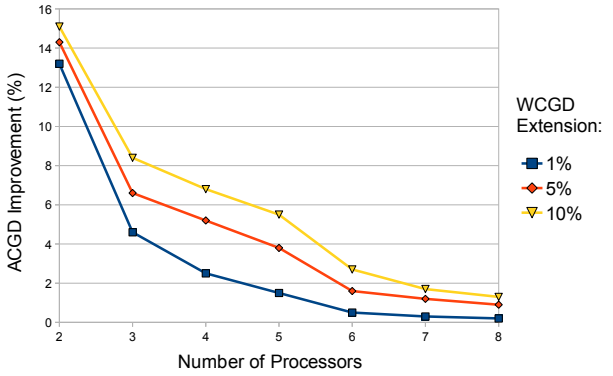

Figure 13. Relative ACGD Improvement

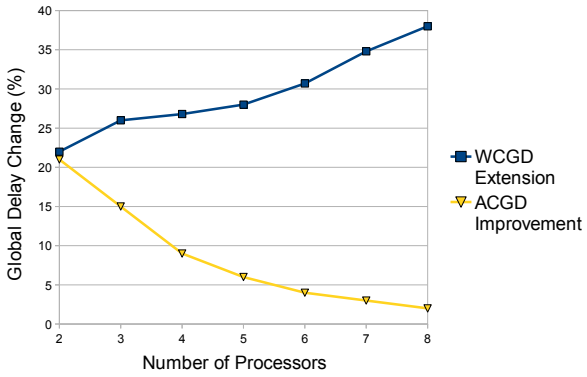

Figure 14. Relative ACGD Improvement and WCGD Extension

compared to when optimizing for both. For this second experiment, we used the very same generated test examples as in the first experiment, allowing for direct comparisons with the already calculated $WCET_0$ and $ACET_0$. Initially, an algorithm for optimizing the bus schedule, taking into account only the ACGD, was applied to the test applications, and then the WCGD was calculated with respect to that bus schedule. Let us denote the resulting ACGD and WCGD by $ACGD'_0$ and $WCGD'_0$ respectively. In Figure 14, we have plotted the relative average extension of $WCGD'_0$ compared to $WCGD_0$, and the average reduction of $ACGD'_0$ compared to $ACGD_0$. As can be seen, not taking the WCGD into consideration when optimizing the bus schedule leads to very high worst-case global delays, whereas the corresponding ACGD improvement is only slightly better than when also optimizing for the WCGD. For instance, for a 5 processor application, the WCGD extension compared to the optimal case ($WCGD_0$) is 28% whereas the improvement of the ACGD relative $ACGD_0$ is 6.0%. By looking in Figure 13, we can see that when optimizing for both ACGD and WCGD simultaneously, for 5 processors we can obtain a 5.5% (instead of 6%) improvement with only a 10% (compared to 28%) degradation of the WCGD.

All experiments were executed on a dual core Pentium 4 processor running at 2.8 GHz. The time to process one application ranged from 10 minutes to 4 hours, depending on the application complexity.

VIII. Conclusions

In this paper, we have presented an approach for bus design optimization, taking into consideration both the average-case and worst-case global delay for real-time applications running on multiprocessor systems-on-chip. Using our technique, the average-case global delay is reduced while the worst case is kept as small as possible. This is the first approach in literature to combine worst and average case optimization for real time systems, and the presented experimental results demonstrate its efficiency. It is important to mention that the proposed approach provides guarantees for worst-case predictability.

References

[1] K. Goossens, J. Dielissen, and A. Radulescu, "AEthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Design & Test of Computers*, vol. 2/3, pp. 115–127, 2005.

[2] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*. Morgan Kaufman, 2008.

[3] P. Puschner and A. Burns, "A Review of Worst-Case Execution-Time Analysis," *Real-Time Systems*, vol. 2/3, pp. 115–127, 2000.

[4] L. Thiele and R. Wilhelm, "Design for Timing Predictability," *Real-Time Systems*, vol. 28, no. 2/3, pp. 157–177, 2004.

[5] S. Schliecker, J. Rox, M. Ivers, and R. Ernst, "Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems," in *CODES+ISSS*, 2008.

[6] J. Rosén, A. Andrei, P. Eles, and Z. Peng, "Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip," in *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS)*, 2007, pp. 49–60.

[7] H. Falk, "WCET-aware Register Allocation based on Graph Coloring," in *DAC*, 2009.

[8] I. A. Khatib, D. Bertozzi, F. Poletti, L. Benini, et al., "A multiprocessor systems-on-chip for real-time biomedical monitoring and analysis: Architectural design space exploration," in *DAC*, 2006, pp. 125–131.

[9] P. Pop, P. Eles, Z. Peng, and T. Pop, "Analysis and Optimization of Distributed Real-Time Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. Vol. 11, pp. 593–625, 2006.

[10] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. L. Sangiovanni-Vincentelli, "Period Optimization for Hard Real-time Distributed Automotive Systems," in *DAC*, 2007, pp. 278–283.

[11] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hamalainen, "Overview of bus-based system-on-chip interconnections," in *ISCAS*, 2002, pp. 372–375.

[12] S. Pasricha, N. Dutt, and M. Ben-Romdhane, "Fast exploration of bus-based on-chip communication architectures," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004, pp. 242–247.

[13] A. Schranzhofer, J. Chen, and L. Thiele, "Timing Analysis for TDMA Arbitration in Resource Sharing Systems," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010.

[14] E. C. Jr and R. Graham, "Optimal Scheduling for two processor systems," *Acta Informatica*, vol. 1, pp. 200–213, 1972.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, , and P. Stenström, "The Worst-case Execution Time Problem – Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, 2008.

[16] C.-F. Neikter, "Cache prediction and execution time analysis on real-time mpsoc," 2008, Master Thesis, LIU-IDA/LITH-EX-A–08/046–SE.

[17] MPARM homepage, "http://www-micrel.deis.unibo.it/sitonew/research/mparm.html".

APPENDIX A.
BUS BANDWIDTH CALCULATIONS

A. Desired Bus Bandwidth Calculation for a Task Segment

In this subsection, we will describe how to calculate the desired bandwidth $\overline{P}_{\Omega_i}$ for a task segment $\Omega_i \in \Xi$, as required by the algorithm in Section V. It is assumed that an average-case execution time analysis has been performed with respect to a bus schedule, that partly or fully is tailored for the worst case. The desired bus bandwidth for a specific part of this bus schedule is, in this context, the distribution of bandwidth that will reduce the average-case global delay as much as possible.

Consider the task segment Ω_k. We would like to calculate the desired bus bandwidth for all processors executing during the corresponding interval of time. Let $T_j \subseteq G(\Pi, \Gamma)$ be the ordered set of the tasks running on processor $1, \ldots, n$ during the time interval specified by Ω_j, and denote these tasks by $\tau_1, \ldots, \tau_n$. Hence, $\tau_i \in T_j$ runs on processor i.

To estimate the desired bandwidth for the different processors, an approximation is needed for how much the current ACET of a task $\tau_i \in T_j$ contributes to the average-case global delay. Let D_i^1 be the set of all tasks $\tau_j \in G(\Pi, \Gamma)$ which have a direct dependency on $\tau_i \in T_j$ in the task graph $G(\Pi, \Gamma)$. Furthermore, let D_i^2 be the singleton set consisting of the first task, after $\tau_i \in T_j$, that is scheduled on the same processor. Combining these two sets, D_i is defined as $D_i = D_i^1 \cup D_i^2$. Now, we can calculate the length of the longest chain of tasks, with respect to their average-case execution times, that are affected by the execution time of $\tau_i \in T_j$. This longest chain of tasks is called the *tail* λ_i of task $\tau_i \in T_j$, and it is formally defined recursively as:

- $\lambda_i = 0$, if $D_i = \emptyset$
- $\lambda_i = \max_{\tau_j \in D_i} (\text{ACET}_{\tau_j} + \lambda_j)$, otherwise.

where ACET_{τ_j} is the average-case execution time of task $\tau_j \in G(\Pi, \Gamma)$ produced by the most recent average-case execution time analysis.

Let us now denote the start time and the end time of the task segment Ω_j by Ω_i^{start} and Ω_i^{end} respectively. For task $\tau_i \in T_j$, we define m_i as the number of cache misses on the average-case control flow path, counting from time Ω_j^{start} to Ω_j^{end}. Similarly, m_i^{end} is defined as the number of cache misses on the average-case path, starting from Ω_j^{end} and counting to the end of the task. Also, we define l_i as the sum of the executed cycles, excluding the time using or waiting for the bus, during the interval Ω_j. l_i^{end} is defined in the same way, but the cycles are now counted between the time Ω_j^{end} and the end of the task.

Now, with respect to the current bus schedule, let d_i denote the average time task $\tau_i \in T_j$ spends waiting, due to bus conflicts and the bus transfer time, for the bus during the time interval Ω_j, each time a cache miss is issued. Note that the following holds for any task $\tau_i \in T_j$:

$$l_i + m_i \cdot d_i = \Omega_j^{end} - \Omega_j^{start} \tag{4}$$

Remember that the desired bandwidth, represented as the fraction of the total bandwidth, for a task $\tau_i \in T_j$ running on processor β during the time interval represented by Ω_j is defined as $p_{\Omega_j}(\beta)$. Let us, for convenience, denote this fraction as p_i. The average waiting time can then be modeled in terms of the desired bandwidth as follows:

$$d_i = \frac{1}{p_i} k \tag{5}$$

According to this model, if we, for a task $\tau_i \in T_j$, approximate the part after time Ω_j^{end} by assuming that all cache misses take k cycles to serve, the average-case global delay can be expressed as:

$$\text{ACGD}_{\Omega_j} = \max_{\tau_i \in T_j} (\Omega_j^{\text{start}} + l_i + m_i \cdot \frac{1}{p_i} k + l_i^r + \lambda_{\tau_i})$$

where $l_i^r = l_i^{end} + m_i^{end} \cdot k$ is the approximation of the remaining part of task $\tau_i \in T_j$, starting from time Ω_j^{end}.

For the segment $\Omega_j \in \Xi$, we now want to find the bus bandwidth distribution that minimizes ACGD_{Ω_j}. This can be formulated as a system of inequalities:

$$\begin{aligned}
\Omega_j^{\text{start}} + l_1 + m_1 \cdot \frac{1}{p_1} k + l_1^r + \lambda_1 &\leq \text{ACGD}_{\Omega_j} \\
&\vdots \\
\Omega_j^{\text{start}} + l_n + m_n \cdot \frac{1}{p_n} k + l_n^r + \lambda_n &\leq \text{ACGD}_{\Omega_j} \\
p_1 + \cdots + p_n &= 1
\end{aligned}$$

Consequently, we want to find the $p_1, \ldots, p_n$ that results in the smallest possible ACGD_{Ω_j}. A very important observation is that for the minimum ACGD_{Ω_j}, the equations above are satisfied with equality, simplifying the calculations significantly. The resulting system of non-linear equations can be solved quickly using standard techniques.

B. Desired Bus Bandwidth Calculation for a Bus Segment

In addition to calculating the desired bus bandwidth for task segments, as done in Section A-A, we would like to do the same for bus segments, i.e. calculate $\overline{P}_{\omega_i}$ for all bus segments $\omega_i \in B$. Since the execution time analysis is used to extract the parameters needed in order to determine the desired bus bandwidth, and it operates on task segments, we must find ways to apply this information to bus segments instead. This has to be done differently depending on the size of the bus segment in relation to the (with respect to time) overlapping task segments. Note that the information needed from the execution time analysis is already stored in $\overline{P}_{\Omega_i}$ for the task segments $\Omega_i \in \Xi$, so we do not have to invoke it again.

For a bus segment $\omega_k \in B$, let O_{ω_k} denote the set of overlapping task segments in Ξ. We want to approximate the desired bus bandwidth for ω_k by using the already calculated $\overline{P}_{\Omega_i}$ vectors for all task segments $\Omega_i \in O_{\omega_k}$. Furthermore, for each $\omega_k \in B$, we define the function $f_{\omega_k} : O_{\omega_k} \rightarrow [0..1] \subseteq \mathbb{R}$, mapping every task segment $\Omega_i \in O_{\omega_k}$ to the fraction of how much it covers the time interval corresponding to ω_k (by overlapping it). Hence, for any $\omega_k \in B$, the following holds:

$$\sum_{\Omega_i \in O_{\omega_k}} f_{\omega_k}(\Omega_i) = 1 \tag{6}$$

Now, the desired bandwidth of a bus segment $\omega_k \in B$ can be calculated as:

$$\overline{P}_{\omega_k} = \sum_{\Omega_i \in O_{\omega_k}} f_{\omega_k}(\Omega_i) \cdot \overline{P}_{\Omega_i} \tag{7}$$

C. Current Bus Bandwidth Calculation

Finding the current bandwidth $\overline{P}^{bus}_{\omega_j}$ for a bus segment $\omega_j \in B$ is trivial, since it is alone determined by the TDMA round constituting the bus segment. For a task segment $\Omega_i \in \Xi$, $\overline{P}^{bus}_{\omega_i}$ can be calculated with a technique similar to the one previously used to derive Equation 7. For a task segment $\Omega_k \in \Xi$, let O_{Ω_k} denote the set of overlapping bus segments in B. Now, for each $\Omega_k \in \Xi$, let us define the function $g_{\Omega_k} : O_{\Omega_k} \rightarrow [0..1] \subseteq \mathbb{R}$, mapping every bus segment $\omega_j \in O_{\Omega_k}$ to the fraction of its total coverage of Ω_k, with respect to time. The current bandwidth of a task segment $\Omega_k \in \Xi$ can now be calculated as:

$$\overline{P}^{bus}_{\Omega_k} = \sum_{\omega_j \in O_{\Omega_k}} g_{\Omega_k}(\omega_j) \cdot \overline{P}^{bus}_{\omega_j} \tag{8}$$