TOWARD TIME-PREDICTABLE EXECUTION OF MULTI-TASK REAL-TIME SYSTEMS

BY

BACH DUY BUI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Abstract

Guaranteeing time-predictable execution in real-time systems involves the management of not only processors but also other supportive components such as cache memory, network on chip (NoC), memory controllers. These three components are designed to improve the system computational throughput through either bringing data closer to the processors (e.g cache memory) or maximizing concurrency in moving data inside the systems (e.g. NoC and memory controllers). We observe that these components can be sources of significant unpredictability in task executions if they are not operated in a deterministic manner. In particular, our analysis and experiments in [6, 35] show that with the standard cache and memory controller sharing mechanism, the execution time of a task may be unpredictably extended up to 33 to 44% in a single-core processor. We also show that analysis techniques and scheduling algorithms that have been proposed to account for and/or to mitigate this unpredictability often do not adequately address the problem at hand. As the consequence, those techniques and algorithms can only guarantee real-time execution in systems with under-utilized shared resources.

In this dissertation, we study the software and hardware infrastructure, optimization techniques and scheduling algorithms that guarantee predictable execution in real-time systems that use cache memory, network on chip (NoC), and memory controllers. The main challenge is how to guarantee system predictability in such a way that maximizes the benefits and the utilization of these components. We achieve that by carefully analyzing both theoretical and practical assumptions in the use of these components and deriving novel solutions based on this understanding. For cache memory, we propose the use of software-based cache partitioning techniques and a real-time optimization method to minimize the system real-time utilization. The proposed solution renders better performance because of its fully utilization of available cache area. For NoC scheduling, we proposed novel scheduling algorithms that are designed to cope directly with the unique assumption on resource sharing in NoC. As will be shown, in practical systems, these scheduling algorithms can achieve near optimal performance. For memory controllers, we propose a software and hardware infrastructure and coscheduling algorithms that are used to control the accesses of the DMA-enabled peripherals to the main memory. The goal is to prevent these accesses from delaying tasks' execution beyond the worst-case execution while still maximizing the I/O throughput.

Table of Contents

Chapter 1

Introduction

In the past few decades, the real-time system research community has been very successful in providing solid theoretical foundation for the development of real-time systems. Various scheduling algorithms and analysis techniques for guaranteeing the predictable execution of real-time systems have been introduced, which first deal with a simple task model on single-core processors [26], then with more complex task models on multi-core processors [4, 8, 52]. However, the development of modern processor architectures has resulted in many new challenges. Among these challenges are the fact that guaranteeing predictable execution now goes beyond processor scheduling: other shared resources in computer systems such as cache, Network-on-Chip (NoC), and the memory controller can significantly affect the execution time of real-time tasks. Nevertheless, the existing real-time system theory has not paid adequate attention to the effect that these components have on predictable execution. As shown in Fig. 1.1, in order to increase execution speed of competing concurrent tasks, besides using multiple processing elements, a typical modern computer system also uses other supportive components. The following paragraphs describe these components and the challenges they create with regards to real-time systems.

- Cache: cache is a small high-speed memory used to buffer data exchanged between the processing units and the main memory. Having been designed to maximize software development productivity and software execution speed, cache is shared between multiple tasks and most of its operations are automatic and hidden from software perspective. As the matter of fact, the sharing of self-controlled cache can greatly affect the predictable performance of real-time systems. We have shown in [6] that in a typical real-time embedded system, unsupervised cache sharing can increase task execution time by up to 33% in a single-core processor. It is because the preemptive execution of different tasks causes

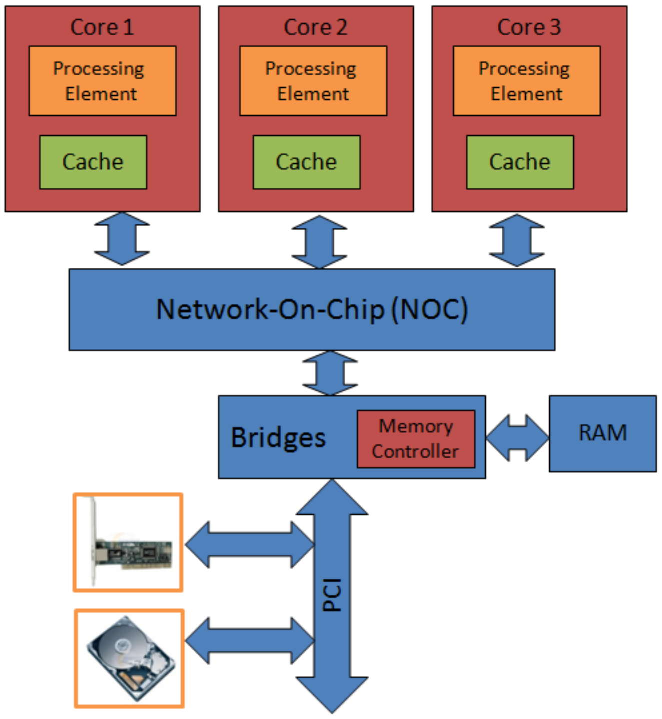


Figure 1.1: Computer Systems

cache lines of a task to be invalidated by the execution of another under the common assumption that cache is globally shared. Due to the random nature of cache access, without a predictable cache-sharing mechanism, identifying cache interference between tasks often results in very pessimistic estimation of task worst-case execution time.

- Network-on-Chip (NoC): NoC is high speed interconnect connecting cores with each other and with peripherals. In order to support an increasing numbers of cores, the focus of NoC design has been on improving NoC scalability and reducing transaction latency. This design focus results in NoC with high degrees of transaction concurrency and complex resource sharing model. The performance of a real-time system built on processors with NoC depends both on the scheduling of real-time tasks on cores, and on the scheduling of real-time data transactions on NoC. While the theory of the former has been mature, that of the later is not. Scheduling of real-time data transactions on a NoC while maximizing the NoC utilization poses new challenges which are different from those of traditional real-time systems. In particular, although the problem has the form of multiple resource scheduling, it cannot be formulated as multi-processor scheduling problem because a transaction may use multiple resources, i.e. physical links, at a time.

- Memory Controller: the memory controller is the component that arbitrates the main memory accesses between cores and peripherals. The memory controller is designed to maximize operation concurrency in the system. In particular, Direct-Memory-Access (DMA)-enabled peripherals can have access to the main memory while some computation tasks are running in the CPU cores. Like cache, the memory controller also operates without software supervision, which means its operations are not explicitly visible from the software perspective. As a consequence, the effect of memory controller on the execution of real-time systems is often hard to control. In [35], we observed that, due to contention for access to the shared infrastructure between a mono-core processor and a DMA-enabled peripheral, a task execution time may be unexpectedly extended up to 44%. Note that the processor is stalled when missed cache lines are being reloaded from the main memory. This stall time can further increase (affecting task execution time) if the memory controller is busy serving memory access of peripherals.

Given the unprecedented challenges raised by cache, NoC, and the memory controller, our major goal of the research in this dissertation is to provide a sound theoretical framework and system designs to cope with the unpredictability caused by these components in real-time systems. In the following paragraph we will summary the main contributions of this dissertation.

As aforementioned, managing cache sharing in real-time systems is crucial for the system predictability. However, a full software control of cache accesses is not desirable since it will require significant software and hardware modification. Also, giving software designers full control of cache accesses may be counter-productive as they will have to pay more attention to low-level operations. We believe a viable alternative is a cache sharing mechanism which requires a minimum change in the existing computer systems, yet provides more efficient real-time performance. In particular, the sharing mechanism should provides better real-time predictability than the existing systems without too much compromising in software execution speed. In Chapter 2, we analyze cache interference in several practical real-time systems. The analysis shows that the effective cache usage of real-time tasks is often much smaller than their memory footprint and than the total cache in the system. However, with the standard cache-sharing mechanism, cache of all tasks may fully overlap, which results in unnecessarily pessimistic worst-case execution time of the tasks. Given this analysis, we propose a cache-partitioning model which allows software designers to specify how cache is shared between real-time tasks. This cache-sharing mechanism can be implemented on existing computer systems and requires little change in the current software development practice and system architecture. We also propose an optimization method which identifies how much cache should be assigned to a certain task. Our simulation results and case studies show a counter-intuitive result that is by giving tasks less cache but explicitly controlling how cache is shared can actually improve the performance of real-time systems in terms of system utilization.

Real-time scheduling on Network-on-Chip (NoC) is a new and challenging problem, mainly because of the two reasons: 1) NoC supports high degrees of transaction concurrency; and 2) the pattern in which transactions share multiple resources is different with what has been considered in traditional real-time theory. In particular, a transaction in NoC, which spans multiple physical links, uses multiple resources simultaneously, while in processor scheduling, a task uses only one processor at a time. Existing works [44, 43, 24, 3] that extend the multi-processor real-time scheduling algorithms and analysis to NoC often do not fully consider transaction topological relationships. Not surprisingly, these proposals can only guarantee real-time deadlines in systems with low NoC utilization, especially when transactions overlap multiple non-overlapping transactions (i.e. non-transitive transaction dependence). It is because the non-transitive dependence is not an assumption of multi-processor scheduling. Higher utilization scheduling algorithms will obviously have to take into account the topological dependence between transactions. However, a general algorithm of this type, i.e one that works on all cases, is intractable. In Chapter 3, we first consider several special cases of transaction topologies and propose efficient algorithms for them. We show that the proposed algorithms have near optimum performance which surpasses that of existing works. The better performance of these algorithms comes precisely from the fact that they are able to exploit the topological dependence between transactions. These algorithms are then extended for some transaction sets on NoC with general topologies. The extension is still an efficient algorithm and has competitive performance.

In the state-of-the-art computer architecture, to maximize the throughput, many active components in the system, such as hard drives and network cards, are allowed to directly transfer data to or from memory without the control of software. The memory controller is used to manage these autonomous memory accesses. This controller operates on transaction-by-transaction basis therefore it can only make local-optimal decisions. As the effect of this sub-optimal operations and the random nature of memory accesses (i.e. ones from cache misses and peripherals), task execution time may be significantly extended under certain memory access patterns. Offline execution time estimation [35] that takes into account this effect can be used to predict the worst-case. However, the estimation is often very pessimistic and inefficient because worst-case situations may significantly affect the task execution but do not occur very often. In Chapter 4, we propose a hardware and software system to regulate the memory accesses of PCI-based peripherals. This system allows us to dictate when peripherals can transfer data in and out of the memory so that it has minimum effect on task execution. An algorithm is also proposed to co-schedule peripherals and real-time tasks. The algorithm strives to maximize the I/O traffic under the constraint that the run-alone worst-case execution time of every task is not violated.

1.1 Reference and Acknowledgment

This work has not been possible without the help of my advisers and the contributions of my colleagues. I would like to give a special appreciation to my adviser Prof. Marco Caccamo for all of the advises he has given to me. Those advises have not only helped me to shape my research direction but also to sharpen my research skill. Through many rough discussions with Marco, I have been able to recognize my drawbacks and to improve my critical thinking. I also want to give special thanks to Prof. Lui Sha. His deep insight knowledge of computer systems has always been the source of excellent research ideas, and of how to tackle hard problems.

I am also grateful to Prof. Tarek Abdelzaher and Prof. Sanjoy Baruah for serving on my thesis committee and for their kind comments to make my research complete.

Last but not least, I would like to thank my PhD fellow Rodolfo Pellizzoni. It has been a great pleasure to be working with him in many of my researches. His exceptional intelligence has helped me to significantly improve the quality of my works. His careful reviews and insightful comments were crucial for the success of our publications.

Note that material used in the dissertation previously appeared in the following publications.

- Bach D. Bui, Rodolfo Pellizzoni, Marco Caccamo, Real-time Scheduling of Concurrent Transactions in Multi-domain Ring Buses, IEEE Transactions on Computer, August, 2011.
- Bach D. Bui, Rodolfo Pellizzoni, Marco Caccamo, A Slot-based Real-time Scheduling Algorithm for Concurrent Transactions in NoC, in the Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2011.
- Bach D. Bui, Deepti K. Chivukula, Marco Caccamo, and Lui Sha, Real-time Communication for

Multicore Systems with Multi-domain Ring Buses, Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2010.

- Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez, Design and Evaluation of a Cache Partitioned Environment for Real-Time Embedded Systems, Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008. (Best paper award)

- R. Pellizzoni, Bach D. Bui, M. Caccamo, and L. Sha, Coscheduling of CPU and I/O Transactions in COTS-based Embedded Systems. Proceedings of the 29th IEEE Real-Time Systems Symposium, 2008.

Chapter 2

Cache Partitioning Techniques and Optimization for Real-time Systems

In this chapter, we will detail the issue of inter-task cache interference in real-time systems through experiments and analytical analysis. As will be shown, this interference may inflate the worst-case system utilization if the cache shared between tasks is not predictably managed. We will then propose and analyze the use of cache partitioning techniques for managing cache sharing. The proposed technique is evaluated through case studies and simulated experiments.

2.1 The Impact of Cache Interference

To understand the impact of cache interference in real-time systems, we conducted an experiment with a real system. In our test-bed we used a DELL with Intel Pentium 4 $1.5GHz$ processor, memory bus speed $400MHz$, and 256KB L2 cache. LinuxRK [32], a real-time kernel developed at CMU, was used as the test-bed operating system. To measure the effect of multi-task cache interference, we used a fixed priority scheduler and two processes running at different priorities: 1) the low priority process τ_{low} has two different pairs of execution times[1] and periods: $(5ms, 10ms)$ and $(10ms, 20ms)$; 2) the high priority one τ_{high} has three different pairs of execution times and periods: $(1ms, 2ms)$, $(2ms, 4ms)$, $(5ms, 10ms)$. For each experiment, there were two runs: during the first one the high priority task limited the number of operations involving a memory access; during the second one, the high priority task was run such that it invalidated as

[1]Note that these execution times were measured in an experimental setup with very limited cache interference.

τ_{high} (ms) / τ_{low} (ms)	$(1, 2)$	$(2, 4)$	$(5, 10)$
$(5, 10)$	13.65%	6.1%	$-$
$(10, 20)$	13.6%	6.15%	2.35%

Table 2.1: Task utilization increment.

many cache lines as possible of the low priority process. The execution time of τ_{low} was measured during both runs and τ_{low}'s utilization increment was computed for each experiment. The results when using a MPEG decoder application as the low priority process are shown in Table 2.1. As can be seen, the task utilization increment can be as high as 13% even though the system has a small L2 cache of size 256KB. It is worth noticing that the utilization increment of τ_{low} is independent of its period (assuming constant task utilization) while it increases inversely proportional to τ_{high}'s period. The results of this simple experiment are in agreement with the case study and extensive simulations of Section 2.5; in fact, 13% task utilization increment can occur even when τ_{low} is suffering a rather limited number of preemptions due to a higher priority task.

By using analytical analysis on a practical real-time systems, we can also show the significant increase in task worst-case execution time (WCET) due to cache interference. Consider the PowerPC processor MPC7410, a popular embedded processor used in real-time systems, which has $2MB$ two-way associative L2 cache (see Table 2.2 for further details). The time taken by the system to reload the whole L2 cache is about $655\mu s$. This reloading time directly affects task execution time. A typical partition size of an avionic system compliant to the ARINC-653 standard [1] can be as small as $2ms$. Under this scenario, the execution time increment due to cache interference can be as big as $655\mu s/2ms \approx 33\%$. In general, the effect of cache interference on task execution time is directly proportional to cache size and CPU clock frequency and inversely proportional to memory bus speed. It is worth noticing that since CPU speed, memory bus speed, and cache size are constantly increasing in modern computer architectures, it is unlikely that this problem will be less severe in the near future.

The above impact analysis has shown that it is crucially important in real-time system design to have reliable techniques to calculate the worst-case cache interference. Such a technique has been proposed by Ramaprasad et la. in [39, 38] for fixed-priority scheduling. With the standard cache-sharing mechanism, this technique assumes that in the worst-case cache of all tasks fully overlap. However, embedded system

Processor	MPC7410
CPU Speed	1000Mhz
L2 cache	2MB two-way set associative
Memory bus speed	125Mhz
Instruction/Data L1 miss + L2 hit latency	9/13 CPU-cycles
Memory access latency	17 Memory-cycles / 32 bytes

Table 2.2: PowerPC configuration

applications [14] usually have smaller memory footprint (i.e. from 50KB - 250KB) than typical size of the cache memory (i.e 1MB - 4MB). Therefore the cache areas of different tasks are not necessary overlap. Furthermore, let define the cache size of an application to be the smallest size of the cache needed by the application to have minimum execution time. It has been shown in [14] that the cache size of a task is usually much smaller than its memory footprint. Our measurement with experimental avionic applications also supports this argument. Table 2.3 shows the memory footprint and cache size of all tasks in our experimental avionic systems. There are eight tasks in total and all of them have cache sizes that are multiple times smaller than their memory footprints. As a consequence, assuming that the cache areas of all tasks fully overlap results in unnecessarily pessimistic WCET estimation.

In order to improve the system efficiency with respect to cache interference, we propose using cache partitioning technique to eliminate and/or minimize the inter-task cache interference. In the next sections, we will describe two different approaches to partition cache and an optimization method to identify the size for cache partitions that minimizes system utilization.

2.2 Cache Partitioning Techniques

The cache partitioning techniques can be largely classified into two main types: hardware-based techniques and software-based ones.

SMART proposed by Kirk [19] is a typical hardware-based cache partitioning. In SMART, the cache memory is divided into equal-sized small segments and one large segment is referred to as shared pool.

Task	Memory footprint (KB)	Cache size (KB)
1	8000	508
2	33000	644
3	668	104
4	9000	416
5	280	28
6	140	60
7	28	8
8	230	216

Table 2.3: Task parameters

The large segment is shared by non-critical tasks while the small ones are dedicated to real-time tasks or groups of real-time tasks. Hardware-based cache partitioning has the benefit of being transparent to higher layers thus requiring little software modification. However its major disadvantages, such as having only fix partition sizes and requiring custom-made hardware, make software-based approaches better choices in practice.

The idea of software-based cache partitioning techniques was first proposed by Wolfe in [50]. By means of software, the code and data of a task are logically restricted to only memory portions that map into the cache lines assigned to the task. In essence, if the task memory footprint is larger than its cache partition, its code and data must reside in memory blocks that are regularly fragmented through out the address space. In [25], Liedtke extended Wolfe's idea exploring the use of operating systems (OS) to manage cache memory. By mapping virtual to physical memory, an operating system determines the physical address of a process, thus also determines its cache location. In contrast, Mueller [30] investigated using compilers to assign application code into physical memory. During compilation process, code is broken into blocks, blocks are then assigned into appropriate memory portions. Since the code address space is no longer linear, the compiler has to add branches to skip over gaps created by code reallocation. Due to the observation that tasks at the same priority level are scheduled non-preemptively with respect to each other, the author suggested that all tasks can be accommodated by using a number of partitions which is no more than the number of priority levels. The main disadvantage of software-based partitioning techniques is that they tend to have larger partition granularity than the hardware-based ones. For example, a technique that use the OS's supports has the smallest partition size to be a page size (i.e. 4KB). However, a software-based technique is much easier to implement in practice and allows much more flexibility in terms of cache partition configuration than a hardware-based one. Furthermore, since the effective memory footprints of practical embedded applications are usually multiple of KB, the partition granularity problem is often not a significant issue. For these reasons, in this research, we advocate the use of software-based OS-controlled techniques like the one in [25]. In the following sections, we will focus on optimizing the software-based cache partitioning mechanism to maximize real-time system schedulability.

2.3 Cache Partitioning Optimization

In this section, we formulate the partitioning problem as an optimization problem (minimizing worst-case utilization) whose solution is expressed by the size of each cache partition and the assignment of tasks to partitions. Since the problem is NP-Hard, a genetic algorithm is used to find a near optimal solution and an approximate utilization lower bound is presented that will be used as a comparison metric to evaluate the effectiveness of our genetic algorithm.

2.3.1 Terminology and assumptions

We consider a single processor multi-tasking real-time system S as a pair $S = \{\mathcal{T}, K^{size}\}$, where $\mathcal{T}$ is a task set of size N: $\mathcal{T} = \{\tau_i : i = [1, N]\}$, K^{size} is total number of cache partition units available in the system. Let δ be the size of a cache partition unit. Note that the value of δ depends on the cache partitioning technique employed: for example, considering an OS-controlled technique, the value δ is the page size, e.g. 4KB. In this case, if CPU has $2MB$ cache, then $K^{size} = 2048KB/4KB = 512$ units. Denote as U_{wc} the worst-case system utilization.

Regarding task parameters, each task τ_i is characterized by a tuple $\tau_i = \{p_i, exec_i^C(k), CRT_i(k)\}$ where p_i is the task period, $exec_i^C(k)$ is cache-aware execution time, $CRT_i(k)$ is cache reloading time, and k is an integer index that represents cache size $k * \delta$. Functions $exec_i^C(k)$ and $CRT_i(k)$ will be formally defined in the following paragraphs.

Definition 2.3.1 *The cache-aware execution time $exec_i^C(k)$ of task τ_i is the worst case execution time of the task when it runs alone in a system with cache size $k * \delta$.*

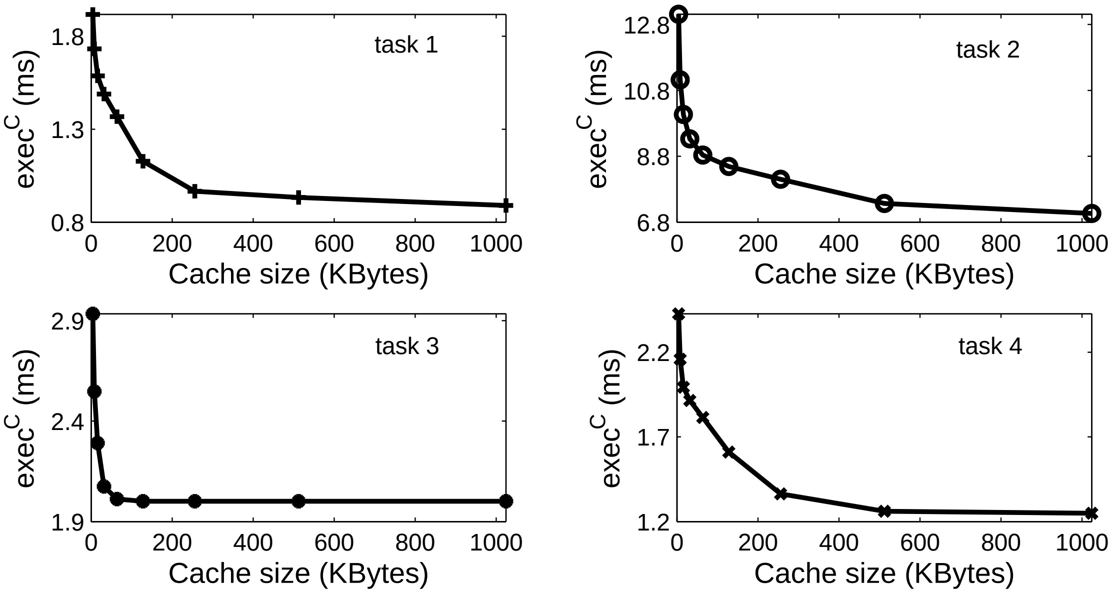


Figure 2.1: $exec^C$ of avionic applications

There have been many techniques proposed for measuring single task worst-case execution time including those that take into account cache effect [48]. In this research, we assume that function $exec_i^C$ can be obtained by measuring τ_i's execution time for different cache sizes by using an available tool. Figure 2.1 depicts $exec^C$ functions of four tasks in an experimental avionic system. According to the figure, it can be noticed that $exec_i^C(k)$ is composed of two sub-intervals. More precisely, the cache-aware execution time is decreasing in the first sub-interval and becomes almost constant in the second one. Intuitively, we can explain this phenomenon as follows: when the cache size is smaller than a task's memory footprint, the execution time diminishes as the cache size increases due to a reduction of cache misses; on the other hand, when the cache size is larger than task's memory footprint (k_i^0), the execution time is no longer cache-size dependent since the number of cache hits does not change.

Next, we define the cache reloading time function.

Definition 2.3.2 *The cache reloading time function $CRT_i(k)$ is total CPU stall time due to cache misses caused by the preemptions that τ_i can experience within a period p_i.*

$CRT_i(k)$ is the overhead that occurs only in multi-tasking systems and takes into account the effect of cache lines invalidated by preempting tasks. In general, $CRT_i(k)$ depends on the maximum number of times that τ_i can be preempted and the size of cache partition that τ_i is using. A good technique to estimate a safe upper bound of $CRT_i(k)$ for fixed priority scheduling is proposed in [39]. This technique can be easily extended for cyclic executive. In this research, we assume that given a task set $\mathcal{T}$, a technique as described in [39] can be used to find $CRT_i(k)$ for each task τ_i.

With the problem at hand, two unknown variables need to be determined for each task τ_i:

- k_i: size of the cache partition assigned to the task τ_i
- a_i: an indication variable whose value is $a_i = 1$ if task τ_i uses a private cache partition or 0 otherwise.

For convenience, we call a tuple $[k_i, a_i]$ as an *assignment* for task τ_i, and $\{a_i : \forall i \in [1, N]\}$ as an *arrangement* of tasks. Having defined all elements, it is now possible to compute the $WCET$ of a task τ_i given an assignment $[k_i, a_i]$:

$$WCET_i(k_i) = exec_i^C(k_i) + (1 - a_i) \times CRT_i(k_i) \tag{2.3.1}$$

2.3.2 Genetic algorithm

The cache partitioning problem is now formulated as an optimization problem whose objective function is to minimize the worst-case system utilization under the constraint that the sum of all cache partitions cannot exceed K^{size} (total available cache size) and all the safety critical tasks (i.e., level A and B of task criticality) should be assigned to a private cache (i.e., $a_i = 1$). Note that a less critical task can use either a shared or a private cache partition.

- **Problem Statement**: Given a real-time system $S = \left\{\mathcal{T}, K^{size}\right\}$, $\mathcal{T} = \{\tau_i : i = [1..N]\}$, and $\forall i \in [1, N] : \tau_i = \left\{p_i, exec_i^C(k), CRT_i(k)\right\}$, find a system configuration $C^{opt} = \{[k_i, a_i] : \forall i \in [1, N]\}$ that minimizes system worst-case utilization U_{wc}.

The mathematical definition of the mentioned optimization problem follows:

$$\begin{aligned} \text{min} \quad & U_{wc} = \textstyle\sum_i \frac{WCET_i(k_i)}{p_i} \\ \text{s.t.} \quad & \textstyle\sum_i a_i \cdot k_i + k^{share} \leq K^{size} \\ & \forall i \in [1, N] : \ 0 < k_i \leq K^{size} \\ & \forall i \in [1, N] : \ k^{share} = k_i \ \mid \ a_i = 0 \\ & \forall i \in [1, N] : \ a_i = 1 \mid \tau_i \text{ is a safety critical task} \end{aligned} \tag{2.3.2}$$

Let U_{wc}^{opt} be the minimum value of the objective function. This problem always has a feasible solution. In general, two questions need to be answered for each task: 1) should we put the task in private or shared cache? (except for the case of safety critical tasks). 2) what is the size of the cache partition? Notice that all tasks that are assigned to the shared partition will have the same cache size as a solution for the optimization problem and it is set equal to k^{share}. This cache partitioning problem is NP-hard since it can be reduced to knapsack problem in polynomial time.

In the remaining of this section, we describe our genetic algorithm that is used to solve the optimization problem. The algorithm is based on the GENOCOP framework [51] which has been shown to perform surprisingly well on optimization problems with linear constrains [2]. The constraint-handling mechanism of GENOCOP is based on specialized operators that transform feasible individuals into other feasible individuals.

Algorithm 1 Cache Partitioning Genetic Algorithm

```
Input: $S = \{\mathcal{T}, K^{size}\}$
Output: $\{[k_i, a_i] : \forall \tau_i \in \mathcal{T}\}$
1: $g \leftarrow 0$ Initialize $P(g)$
2: while $g \leq G_{max}$ do
3:     mutate some individuals of $P(g)$
4:     cross over some individuals of $P(g)$
5:     locally optimize $P(g)$
6:     evaluate $P(g)$
7:     $g \leftarrow g + 1$
8:     select $P(g)$ from $P(g-1)$
```

Our solution is shown in Algorithm 1. Specific information of the problem at hand is employed to design the operators which are local optimizer, mutation and crossover. At each generation g, a set of individuals (i.e., population $P(g)$) is processed. Individuals are feasible solutions of the optimization problem. A non-negative vector X of fixed length $N+1$ is used to represent an individual, where: N is number of tasks; and $\forall i \in [1, N]$ if $X[i] > 0$, then $X[i]$ is the size of the private cache of τ_i, if $X[i] = 0$, then τ_i uses the shared

[2]Notice that the considered optimization problem involves the evaluation of a non-monotonic function $CRT_i(k)$. As a consequence, solvers as hill climbing or simulated annealing could be easily trapped within a local optimum. Hence, a genetic algorithm was chosen to circumvent this problem.

partition of size $X[N+1]$. For example, $X = [2, 3, 0, 0, 5]$ means τ_1 and τ_2 use two private partitions with $k_1 = 2$ and $k_2 = 3$, whereas τ_3 and τ_4 use the shared partition with $k_3 = k_4 = k^{share} = 5$. Individuals are evaluated (line 6) using Equation 2.3.2. The outcome of the evaluation i.e. U_{wc} of each individual is then used to select which individual will be in the next generation. The lower U_{wc} an individual has, higher is the probability that it will survive. Let U^h_{wc} be the utilization of the output configuration i.e. the lowest utilization found by the algorithm. The three operators (i.e., local optimizer, mutation, and crossover) are described in the following sections. In the following algorithms, all random variables generated by instruction $random$ have uniform distribution.

2.3.3 Local Optimizer

Since $exec^C_i(k)$ is a non-increasing function, increasing $X[i]$ always results in a smaller or equal utilization of τ_i thus a smaller or equal system utilization. This observation leads to the design of a simulated annealing local optimizer as showed in Algorithm 2. Each individual of population $P(g)$ undergoes the local improvement before being evaluated. The original individuals are then replaced by the local optima. This heuristic reduces the search space of the genetic algorithm to only those solutions that are locally optimal.

Algorithm 2 Simulated Annealing Local Optimizer

```
Input: $X[1..N+1]$
Output: $X[1..N+1]$
 1: $T \leftarrow T_{max}$
 2: while $T > T_{min}$ do
 3:     $tries \leftarrow 0$
 4:     while $tries < tries_{max}$ do
 5:         $X' \leftarrow X$
 6:         $i \leftarrow random[1, N]$ such that $X[i] > 0$
 7:         $X'[i] \leftarrow X'[i] + random(X'[i], K^{size} - \sum_{j=1, j \neq i}^{j=N} X'[j])$
 8:         $\Delta U \leftarrow U'_{wc} - U_{wc}$
 9:         if $\Delta U < 0$ and $random[0, 1] < 1/(1 + e^{\Delta U/T})$ then
10:             $X[i] \leftarrow X'[i]$
11:         $tries \leftarrow tries + 1$
12:     reduce $T$
```

At each trial, an $X[i]$ is chosen at random (line 6) and its value is increased by a random amount within its feasible range (line 7) . The new value is accepted with a probability proportional to temperature T. Only variables representing private cache are considered (i.e. $X[i] > 0 \ \forall i \in [1, N]$). Note that enlarging the size of the shared partition, $X[N + 1]$, does not necessarily result in reducing utilization since it may increase the cache reloading time. Obviously, this heuristic can only find approximate local optima with respect to a certain *arrangement* of tasks; the global optimum might require a task to use a shared partition instead of a private one (or vice-versa) as assigned by the local optimizer. The mutation and crossover operators are designed to allow the algorithm to search beyond local optima.

2.3.4 Mutation Operator

Algorithm 2 minimizes the utilization by enlarging size of private cache partitions thus the solution is only locally optimal with respect to a certain *arrangement* of tasks. The mutation operator described in Algorithm 3 helps to search beyond local optima by randomly rearranging tasks into a private or shared cache partition. In other words, it creates new landscape where the local optimizer can work.

Algorithm 3 Mutation Operator

```
Input: $X[1..N + 1]$
Output: $X[1..N + 1]$
1: $flip \leftarrow random\{TAIL, HEAD\}$
2: if $flip = TAIL$ then
3:     $i \leftarrow random[1, N + 1]$
4:     assign $X[i]$ to a random value in its feasible range
5: else
6:     $i \leftarrow random[1, N]$ such that $X[i] > 0$
7:     $X[i] \leftarrow 0$
```

The operator takes as input an individual at the time. The operator randomly chooses a task τ_i for either modifying its private cache size or rearranging the task into shared cache by assigning 0 to $X[i]$. The size of shared cache partition may also be changed. The operator guarantees that the generated individual is feasible.

2.3.5 Crossover Operator

The crossover operator (Algorithm 4) aims to transfer good chromosome of the current generation to the next one. Algorithm 2 finds, for each individual, a local optimum with respect to its *arrangement* of tasks. After undergoing local optimization, if one individual has lower utilization than another, there is a high probability that the former has a better *arrangement*. In other words, the genetic advantage of an individual over another is implicitly expressed by its *arrangement*. Our design of crossover operator exploits this information.

The operator spawns descendants by randomly combining the *arrangement*s of pairs of predecessors. The algorithm takes two parents as input and produces two children. If τ_i of one of the parents uses shared cache (i.e. $X[i] = 0$), the *assignment* of the child's τ_i is the *assignment* of either of the parent's τ_i, otherwise it is the arithmetic crossover (i.e. $b * X_1[i] + (1 - b)X_2[i]$ where b is a continuous uniform random variable taking values within $[0, 1]$). The operator guarantees that if parents are feasible then their children are also feasible.

Algorithm 4 Crossover Operator

```
Input: $X_1[1..N+1], X_2[1..N+1]$
Output: $X'_1[1..N+1], X'_2[1..N+1]$
1: $a \leftarrow random[0, 1]$
2: for $i = 1$ to $N + 1$ do
3:   if $X_1[i] = 0$ or $X_2[i] = 0$ then
4:     $flip \leftarrow random\{TAIL, HEAD\}$
5:     if $flip = TAIL$ then
6:       $X'_1[i] \leftarrow X_1[i]$
7:       $X'_2[i] \leftarrow X_2[i]$
8:     else
9:       $X'_1[i] \leftarrow X_2[i]$
10:      $X'_2[i] \leftarrow X_1[i]$
11:  else
12:    $X'_1[i] \leftarrow b * X_1[i] + (1 - b) * X_2[i]$
13:    $X'_2[i] \leftarrow b * X_2[i] + (1 - b) * X_1[i]$
```

2.4 Approximate Utilization Lower Bound

In this section we derive an approximate utilization lower bound for the cache partitioning problem which will be used to evaluate our heuristic solution. The bound gives a good comparison metric to evaluate the effectiveness of a given partitioning scheme and is approximate because it is computed by using $exec_i^C(j)$ and $CRT_i(j)$ functions. In realistic scenarios, these two functions are derived experimentally and have some approximation errors. Hence, the bound is approximate too. A definition of approximate utilization lower bound U_b follows.

Definition 2.4.1 *An approximate utilization lower bound U_b of the cache partitioning problem (defined in Section 3.2) is a value that satisfies $U_b \leq U_{wc}^{opt}$.*

The bound U_b is easily computed starting from an initial phase that assumes unlimited cache size and all tasks have a private cache size of their memory footprint. Then, at each step the total cache size is reduced either by shrinking the size of a private partition or by moving a task from private cache to shared one: the decision is made in such a way that the increment of total task utilization is minimized at each step. This technique is similar[3] to Q-RAM [37], and the iterative algorithm is executed until the total cache size is reduced to K^{size}.

Although the bound value is not exactly U_{wc}^{opt}, as will be shown in Section 2.5, it is still a good measure for the performance of any heuristic solving the analyzed optimization problem. In addition, system designers can also use this bound to predict how much utilization at most can be saved when applying any partitioning heuristic. According to equations 2.3.1 and 2.3.2, the utilization of a task may increase or decrease depending on its cache attributes like size, private/shared partition and cache reloading time. The derivation of U_b is based on the notion of two distinct utilization differentials per cache unit: $\Delta_i^e(j)$ and $\Delta_i^r(j)$. In fact, $\Delta_i^e(j)$ indicates how a task's utilization varies as a function of the size of its private cache partition, and $\Delta_i^r(j)$ is a normalized index on how task's utilization varies when switching from private to shared cache. The formal definitions of $\Delta_i^e(j)$ and $\Delta_i^r(j)$ follow.

Definition 2.4.2 *The utilization differential $\Delta_i^e(j)$ of task τ_i is the difference in utilization per cache unit when τ_i's private cache is reduced from j to $j-1$ units.*

[3]Q-RAM starts always from a feasible state with minimum resource allocation, while our algorithm starts from an infeasible state by assuming unlimited resource availability.

$$\Delta_i^e(j) = \frac{exec_i^C(j-1) - exec_i^C(j)}{p_i} \tag{2.4.1}$$

Task τ_i's utilization may increase when changing from private to shared cache due to the presence of cache reloading time. If τ_i's private cache size is j then when the switching takes place, the amount of freed cache is j units. Thus we have the following definition of utilization differential $\Delta_i^r(j)$ caused by cache reloading time.

Definition 2.4.3 *The utilization differential $\Delta_i^r(j)$ of task τ_i is the difference in utilization per cache unit caused by cache reloading time when τ_i is moved from a private partition of size j to a shared partition of the same size.*

$$\Delta_i^r(j) = \frac{CRT_i(j)}{j * p_i} \tag{2.4.2}$$

Having introduced the basic notions, we now describe the algorithmic method for estimating U_b (Algorithm 5). Since $exec_i^C$ is a non increasing function, τ_i has its smallest utilization when its cache is private and has maximal size (i.e. $[k_i = k_i^0, a_i = 1]$). Consequently, the system utilization is minimized since every task has its own private cache of maximum size. That absolute smallest system utilization (U_b^{abs}) and the total needed cache size (K) are calculated in line 6 and 7, respectively. This configuration, however, is not feasible when $K > K^{size}$ which holds in most practical systems. Note that after line 8, $U_b = U_b^{abs}$ is a lower bound following Definition 2.4.1. Nevertheless, it is not the tightest bound that can be found in polynomial time. A tighter bound is estimated using procedure starting from line 11. Essentially, it reduces the value of K toward that of K^{size} (line 13 and 17). Then for each unit of cache size taken from K, the value of U_b is increased (line 14 and 18) such that U_b approaches but **never exceeds** U_{wc}^{opt}. This is done by using the smallest values of Δ^e and Δ^r to update U_b at each step. The correctness of the procedure is proven in the following paragraphs.

Consider a configuration $C = \{[k_i = k_i^0, a_i = 1] : \forall i \in [1, N]\}$ that uses total cache size $K = \sum_i k_i^0 > K^{size}$, there are three basic operations that can be executed to let C to converge toward C^{opt}:

1. reducing the size of any private partition by 1 unit thus reducing K by 1 unit and increasing U_b by a value $\Delta_i^e(j)$.

2. moving a task from its private partition of size j to a shared partition of the same size, thus reducing K by j units and increasing U_b by a value $\Delta_i^r(j) * j$.

3. reducing the size of the shared partition by 1 unit thus reducing K by 1 unit.

Lemma 2.4.1 shows that since operation 3 is equivalent to a sequence of the other two, only operation 1 and 2 are needed to compute U_b.

Lemma 2.4.1 *Every sequence of operations used to compute U_b can be converted to a sequence of operations 1 and 2.*

Proof.
We only need to prove that any sequence of operation 3 can be represented as a sequence of operations 1 and one operation 2. Assume that the final size of the shared partition is k^{share}. We can always reduce the size of any task's private partition to k^{share} using operation 1, then by applying operation 2 those tasks can be moved to the shared partition and U_b can be computed without using operation 3. □

Using Lemma 2.4.1, we can prove the following theorem that implies the correctness of Algorithm 5

Theorem 2.4.1 *The output of Algorithm 5 (U_b) is smaller or equal to U_{wc}^{opt}*

Proof.
Lemma 2.4.1 proves that every transformation applied to compute U_b is composed of sequences of operation 1 and 2. Consider a task τ_i currently using a private cache of size j: for each operation 1 applied to τ_i, U_b increases by $\Delta_i^e(j)$ and K decreases by 1; for each operation 2 applied to τ_i, U_b increases by $\Delta_i^r(j) * j$ and K decreases by j. Since Algorithm 5 uses the smallest value among Δ^e and Δ^r to update U_b at each step, after the while loop (when $K = K^{size}$), $U_b \leq U_{wc}^{opt}$. The time complexity of the while loop is bounded by $\sum_i k_i^0 - K^{size}$ □

Note that U_b is an utilization lower bound and Algorithm 5 does not produce a feasible solution since it might end up splitting the cache of a task into a private part and a shared one.

Algorithm 5 U_b Estimation

Input: $S = \{\mathcal{T}, K^{size}\}$

Output: U_b

```
1: for $i = 1$ to $N$ do
2:     for $j = 1$ to $k_i^0$ do
3:         $EXE[\sum_{l=1}^{i-1} k_l^0 + j] \leftarrow \Delta_i^e(j)$
4:         $REL[\sum_{l=1}^{i-1} k_l^0 + j] \leftarrow \Delta_i^r(j)$
5: sort $EXE$ and $REL$ in decreasing order
6: $U_b^{abs} \leftarrow \sum_i exec_i^C(k_i^0)/p_i$
7: $K \leftarrow \sum_i k_i^0$
8: $U_b \leftarrow U_b^{abs}$
9: $e \leftarrow$ size of $EXE$
10: $r \leftarrow$ size of $REL$
11: while $K > K^{size}$ do
12:     if $EXE[e] < REL[r]$ then
13:         $K \leftarrow K - 1$
14:         $U_b \leftarrow U_b + EXE[e]$
15:         $e \leftarrow e - 1$
16:     else
17:         $K \leftarrow K - \min(K - K^{size}, j)$
18:         $U_b \leftarrow U_b + \min(K - K^{size}, j) * REL[r]$
19:         $r \leftarrow r - 1$
```

2.5 Evaluation

This section evaluates the proposed algorithm by using input data taken from real and simulated systems. Although the solution can be applied to systems with any scheduling policy, all the experiments in this section assume a cyclic executive scheduler. This assumption is motivated by the fact that the cyclic executive scheduler is commonly used by avionic industry due to the high criticality of the developed real-time systems. In our evaluation, we are concerned only with the last level of cache as it affects system performance the most. However, it is noted that taking into account other cache levels would not change the performance of the proposed algorithm. We start first by describing the methodology used to generate input data, then we show the experimental results.

2.5.1 Methodology

This section discusses our method to simulate cache access patterns and to generate functions $exec^C$ and CRT. In [49], Thiébaut developed an analytical model of caching behavior. The model is provided in the form of a mathematical function (Equation 2.5.1) assigned to each task that dictates the cache miss rate for a given cache size k. Although it was originally developed for a fully associative cache, it has been verified that the model generates synthetic miss rates that are very similar to the cache behavior of actual traces across the complete range of cache associativities [49].

$$MissRate(k) = \begin{cases} \frac{1-\frac{k}{A_1}(1-\frac{1}{\theta})-A_2}{1-A_2} & \text{if } k \le A_1 \\ \frac{\frac{A^\theta}{\theta}k^{(1-\theta)}-A_2}{1-A_2} & \text{if } A_1 < k \le k^0 \\ 0 & \text{if } k^0 < k \end{cases} \tag{2.5.1}$$

$$A_1 = A^{\theta/(\theta-1)} \tag{2.5.2}$$

$$A_2 = \frac{A^\theta}{\theta}k^{0^{(1-\theta)}} \tag{2.5.3}$$

Cache behavior is function of three task-dependent parameters, A, θ, k^0. A determines the average size of a task's neighborhood (i.e., working set). The higher A is, the larger the task's neighborhood becomes. Notice that a larger working set increases the probability of a cache miss. θ is the locality parameter which is inversely proportional to the probability of making large jumps. The probability that a memory access visits a new cache line diminishes as θ increases. It has been shown statistically that real applications have θ ranging from 1.5 to 3 [49]. It is worth noticing that Equation 2.5.1 only models the capacity cache miss. A complete model needs to also encapsulate the effect of compulsory cache miss (i.e., cold cache miss). A system with single or non-preemptive tasks has cache miss rate converges to that of Equation 2.5.1. However this is not the case in preemptive multi-tasking systems where compulsory cache miss becomes significant due to frequent preemptions. Equation 2.5.4 proposed by Dropsho [9] modifies Equation 2.5.1 to calculate the miss rate occurring at the warm-up phase too. The improved model calculates the instantaneous miss rate by using the current number of unique cache entries as the instantaneous effective cache size.

$$\begin{aligned} &InstantRate(m,k) \\ &\quad = \begin{cases} MissRate(m) & \text{if } m < k \\ MissRate(k) & \text{otherwise} \end{cases} \end{aligned} \tag{2.5.4}$$

The inverse of instantaneous miss rate is the average number of memory references required to arrive at the next miss. This information is used to calculate the number of references required to have M misses (Equation 2.5.5).

$$Ref(M,k) = \sum_{m=1}^{M} 1/InstantRate(m,k) \tag{2.5.5}$$

We are now ready for the calculation of $exec^C$ and CRT. Considering task τ that has total number of memory references R, the task's $exec^C$ is calculated according to Equation 2.5.6. Note that, in this case we can directly use Equation 2.5.1 to approximate cache miss rate since by definition $exec^C$ is the execution time of τ when running non-preemptively. Hence, $exec^C$ can be computed as it follows:

$$\begin{aligned} exec^C(k) = \quad & (1 - Missrate(k)) * R * HitDelay \\ & + Missrate(k) * R * MissDelay, \end{aligned} \tag{2.5.6}$$

where $HitDelay$ and $MissDelay$ are the execution times of an instruction when a cache hit or cache miss occur, respectively. The values of these constants depend on the adopted platform. Assume now that τ runs on a multi-tasking system scheduled by cyclic executive with a time slot of size s (in general, $exec^C$ is longer than s): since no assumption is made on the memory access pattern of other tasks, in the worst case τ has to warm-up the cache again at each resumption. In other words, in order to calculate the number of memory references (Ref) taken place in each time slot s, Equation 2.5.5 must be used and the value of M is such that the induced execution time in that time slot $((Ref(M,k) - M) * HitDelay + M * MissDelay)$ is equal to s. Note that in a multi-tasking system, the number of memory references (i.e. instructions) executed in a time slot is less than what can be executed within the same time slot in a non-preemptive or single-task system due to inter-task cache interference problem. Therefore, task τ would take more than $exec^C$ time to complete its total number of references. By definition, that additional time is captured by CRT.

In summary, to generate simulated input data, we generate a set of task-dependent parameters for each task: A, θ, k^0, the size of scheduling time slot s, and the total number of task's memory references. $exec^C$

Task	Number of references	Memory usage (KB)	Time slot	Cache size (KB)
1	61560	8000	4	508
2	259023	33000	5	644
3	76364	668	1	104
4	90867	9000	3	416
5	32544	280	2	28
6	6116	140	1	60
7	41124	28	1	8
8	217675	230	6	216

Table 2.4: Task parameters

and CRT of each task are then calculated accordingly using Equation 2.5.5 and 2.5.6. We emphasize that the generated data is only for the purpose of evaluating the performance of Algorithm 1; they do not replace techniques that estimate $exec^C$ and CRT. The proposed heuristic can be applied to any practical system whenever information of $exec^C$ and CRT is available by any means.

2.5.2 Case Study

In this section, an experimental avionic system with eight tasks is used to verify the effectiveness of the proposed approach. The system parameters are the same as those in Table 2.2. The scheduler runs with a major cycle of six time slots: five tasks have their own time slot and three others share one time slot. The size of each time slot is shown in column 2 of Table 2.5. Tasks' parameters, including number of memory references, memory usage and time slot allocation are shown in Table 2.4. All tasks have the same period of $16.67ms$. The number of memory hits and misses were measured as a function of the cache size for each task. The traces were then used to find $exec^C$ and CRT functions. $exec^C$ of tasks 1 to 4 are plotted in Figure 2.1. Although some tasks may use a large amount of memory, the range of cache sizes at which its $exec^C$ function differential is significant may be smaller. This is because, in most cases, the worst case execution time path accesses only a small part of all the memory allocation. For example, memory size of task 4 is about 9000KB but its $exec^C$ is subject to big variations only at cache sizes smaller than 512KB. In other words, it is possible to assign to a task an amount of cache smaller than its memory allocation without

Time slot	Size (ms)	Baseline: slot utilization (U_{share})	Heuristic: slot utilization (U_{wc}^h)
1	2.3	103.8	70.4
2	1.2	45.9	35.8
3	3.2	48.0	42.2
4	1.9	54.8	48.9
5	4.4	99.5	80.5
6	3.67	100.2	77.4

Table 2.5: Task worst-case utilization

increasing its execution time much. This suggests that cache partitioning can still be useful in practice even with big memory-footprint applications.

To calculate CRT function, we use the method discussed in Section 2.5.1. In addition, the task's parameters, i.e. A, θ, k^0, are given by fitting $Missrate(k)$ (Equation 2.5.1) into the measured trace. The correctness of this model is verified by the fact that CRT is always smaller than 30% of $exec^C$ (see Section 2.1 for more details). In this case study, the algorithm outputs a partitioned configuration where all tasks use a private cache. Column 5 of Table 2.4 reports the resulting size of each cache partition. As expected, in many cases the partition size is much smaller than task's memory size.

Table 2.5 reports time slot's utilization U_{share} for the baseline configuration (that uses only a shared cache) and the improved time slot's utilization (by using the proposed heuristic) on columns 3 and 4, respectively. The utilization of a time slot is the percentage of slot duration used by the task(s) assigned to that slot; tasks running in a time slot are not schedulable if the slot utilization is greater than 100%. In this case study, slots 1 and 6 are not schedulable under the baseline configuration but they are under the partitioned configuration. The baseline utilization U_{share}, the partitioned one U_{wc}^h, and the utilization bound U_b are 79.7%, 64.2%, and 61.3%, respectively. The utilization gain is 15.4% while U_{wc}^h is only 2.9% greater than U_b.

2.5.3 Evaluation with Simulated Systems

The same system parameters shown in Table 2.2 are used for simulations. Tasks' parameters are randomly chosen with uniform distribution within the intervals presented in Table 2.6. The range of A and θ is selected

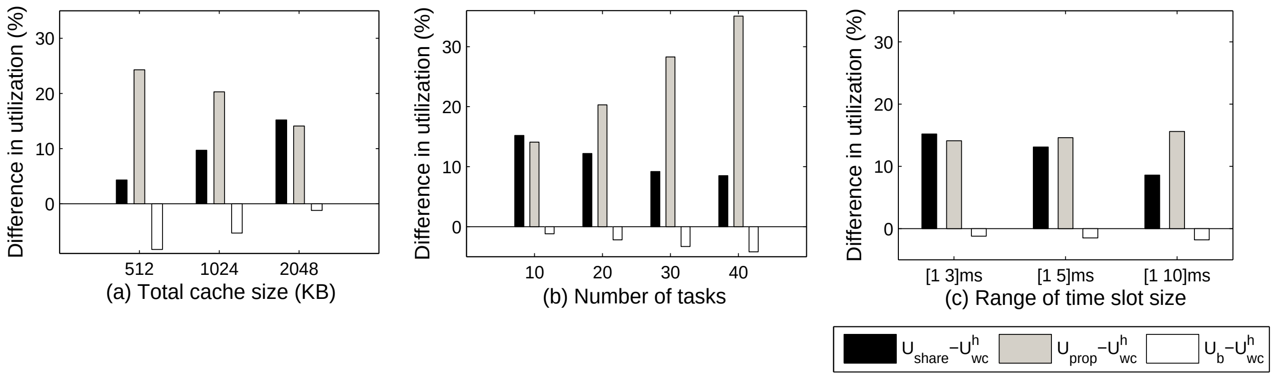


Figure 2.2: Effect of the three factors

A (KB)	θ	k^0 (KB)	number of memory references
$[1, 10]$	$[1.5, 3]$	$[A_1, 1024]$	$[10^3, 10^6]$

Table 2.6: Simulation parameters

based on values given by [49] which also fit well with those used in the case study. All experiments use the range of k^0 shown in Table 2.6 except where noted otherwise. A_1 is calculated using Equation 2.5.2. Each task has utilization smaller than 100% when it runs with the baseline configuration. The following simulations describe how different factors (i.e., the total cache size, the number of tasks, and the size of time slots) affect the performance of the proposed algorithm. The performance measures are the average utilization gain defined as $U_{share} - U^h_{wc}$ and the average value of $U_b - U^h_{wc}$. For comparison purposes, we also estimate the worst case utilization of proportional cache partitioned systems (U_{prop}). In such a system, a task is assigned a private cache partition of size proportional to its memory footprint, i.e. $k_i = \frac{k^0_i}{\sum_1^N k^0_j} K^{size}$. The average value of $U_{prop} - U^h_{wc}$ is reported. Simulated system parameters are as follows except where stated otherwise: $N = 10$, $K^{size} = 2MB$, and time slot range from 1 to $3ms$. All results are the average over the outputs of 30 different task sets.

Effect of the total cache size: in this experiment, we measured systems having cache sizes of 512KB, 1MB, and 2MB. The performance measures plotted in Figure 2.2(a) show that the heuristic performs very well especially when the total cache size is large, i.e. 2MB: the gain in utilization is about 15% and the difference between the heuristic utilization and the bound is less than 2%.

Effect of the number of tasks: in this experiment, the number of simulated tasks is 10, 20, 30, and 40.

Results are plotted in Figure 2.2(b). Notice that two phenomena occur when the number of tasks increases: 1) the gain in utilization $U_{share} - U_{wc}^{h}$ is smaller since more tasks are forced to use the shared partition, 2) the poor performance of proportional cache partitioning is more prominent since there are more tasks running on smaller private cache partitions.

Effect of the size of time slots: in this experiment, tasks' time slot sizes are randomly chosen with uniform distribution within the following three ranges: $1-3ms$, $1-5ms$, $1-10ms$. Intuitively, when the time slot size increases, the effect of cache-related preemption delay is reduced since tasks experience less preemptions during each execution. Consequently, the utilization gain due to cache partitioning is lower. This behavior is shown in Figure 2.2(c): as expected, the utilization gain $U_{share} - U_{wc}^{h}$ is reduced to 8% at the largest range whereas the gap between U_{wc}^{h} and U_b is almost constant. Notice that time slot variation has no effect on $U_{prop} - U_{wc}^{h}$.

2.6 Conclusion

In this research, we have shown that cache partitioning can be used to improve system schedulability. This is because in real-time systems a schedulability test has to always assume the worst-case inter-task cache interference. This assumption often implies a low cache area utilization in practical systems. The proposed cache partitioning optimization is designed to optimize cache area usage so that the cache interference is minimized when necessary with regards to real-time guarantee.

Chapter 3

Network-on-Chip Real-time Scheduling

As mentioned in Chapter 1, scheduling of real-time transactions in a NoC while maximizing its utilization poses new challenges which are different from those of traditional real-time systems. In particular, although the problem has the form of multiple resource scheduling, it is not the same as multi-processor scheduling because a transaction may use multiple resources, i.e. physical links, simultaneously. This pattern of resource sharing create *non-transitivity dependence* between transactions. For example, while two transactions τ_1 and τ_2 which do not share any links can be transmitted concurrently, they both will be delayed by the transmission of transaction τ_3 which uses links shared with both τ_1 and τ_2. To the best of our knowledge, this real-time resource sharing problem has not received adequate attention. In related works [43, 44], Shi et al. proposed a method to calculate worst-case latency of transactions scheduled by a *fixed-priority* scheduling algorithm. Since these works extend the traditional real-time scheduling paradigm to NoC, they are not able to take full advantage of the parallelism available between non-overlapping transactions.

In this research, we propose novel real-time scheduling algorithms specifically designed for NoC. Through understanding of the dependence between transactions, the algorithms make dynamic scheduling decisions so that transaction concurrency is maximized. Due to the problem complexity, we will first focus on NoC with ring topology (Section 3.3). Transaction sets on ring-topology NoC are classified into acyclic and cyclic transaction sets. We propose scheduling algorithms first for the former then extend them for the latter. We will then extend our theory for ring-topology NoC to the general NoC. More specifically, in Section 3.4, we propose scheduling algorithms for acyclic transaction sets on general NoC. All proposed algorithms are evaluated through simulated experiments and implementation.

3.1 Related Works

Many of the early works on hard real-time communication [22, 46, 31] focus on communication between computers on *single-domain* bus networks. In these networks, only one transaction can be transferred on a bus at any time because the bus is shared between all transactions. A system with multiple buses is considered in [13]. However, each bus in the system still has one domain. Since a single-domain bus bears a similarity to single-processor systems, the traditional real-time scheduling theory for single-processor systems [26] is applied or extended to solve the problem in these works. The many-core SoC in which we are interested have multi-domain buses where non-overlapping transactions can be transferred concurrently. In addition, the number of domains on a bus is determined by the topology of bus transactions.

There has also been significant research focused on real-time communication on multi-domain buses. Most of these works [43, 44, 3, 24, 28] are concerned with the *fixed-priority* scheduling paradigm. In these algorithms, each transaction is given a fixed priority at design time and higher priority transaction τ_i always preempts its overlapping transactions which have lower priority (the preemption occurs at the flit level). Most recently, Zheng et al. propose in [43, 44] a solution to optimally assign fixed priorities to real-time transactions and a method to analyze the worst-case transaction latency (WTL) under a fixed-priority scheduling algorithm. Although our work has the same assumption about multi-domain buses, our proposed scheduling algorithm is based on the *dynamic-priority* scheduling paradigm: that is a transaction may be assigned with different priorities at runtime. To the best of our knowledge, our research is the first to do so. As will be shown in the evaluation section, the performance of our approach on a typical NoC is better than that of related works. We also note that since our proposed algorithm dynamically computes the schedule, it has higher runtime overhead than that of the fixed-priority ones. This overhead may adversely affect the algorithm performance. However, as we analyzed and demonstrated by experiments in Section 3.3.4, this overhead is, in fact, relatively small.

3.2 Real-time NoC Model

We consider a system comprising multiple Processing Elements (PEs) interconnected by a NoC. The NoC is composed of routers connected by communication links. Each router has several links directly connecting it with neighbors. The number of the neighbors of each router depends on the network topology. For example, a router on a ring-topology NoC has two neighbors, while one on a 2D-topology NoC has four. Links can

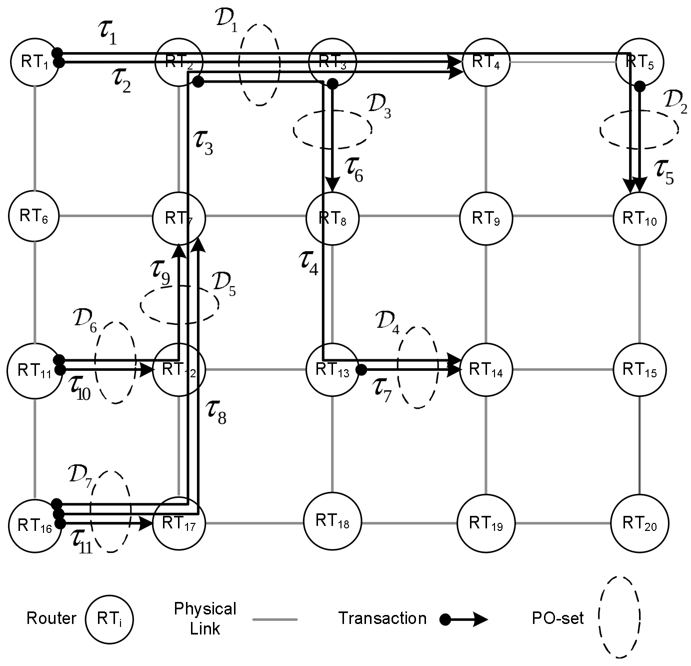


Figure 3.1: A transaction set in NoC

be either unidirectional or bidirectional, but each link is single-duplex which supports only a *single* data transmission at a time. Most NoC employ full-duplex connections; in our model, they are represented by two unidirectional links. Figure 3.1 shows an example of a 4×5 mesh NoC.

We assume that real-time applications running on multiple processing elements exchange data through the NoC. A *data* transaction is defined as a request made by an application to transfer a certain amount of data between two processing elements. We consider a scheduling problem where applications request *periodic* data transactions, each comprising an infinite sequence of jobs. Each data transaction is transferred hop-by-hop from the source to the destination. We assume that each data transaction has a fixed route which consists of routers through which it reaches the destination. Also, when a transaction is transmitted, it uses all links on its route at the same time. This is the assumption used in NoC with wormhole switching [7, 10], which are the most popular switching protocol used in NoC. Note that this assumption does not imply that the link resource is wasted because, in practice, a packet will be split into multiple flits, which then will be sent in pipeline along its route. The above assumption means that two data transactions *overlap* and can not be transferred concurrently if their routes share a same link. However, multiple non-overlapping data transactions can be sent at the same time.

In this research, we propose the use of *slot-based* scheduling model, in which the time line is divided into consecutive equal slots and transactions are scheduled on contention-free slots. This type of scheduling has been used to build PFair [4] and BF [52], which are the optimal scheduling algorithms for multi-processors. Although this approach requires synchronization between routers and computation at the end node, it can significantly reduce implementation complexity of real-time NoC because it eliminates the need of buffers and arbiters at the routers. The slot-based scheduling model has been successfully implemented in the Aethereal NoC [12], a guaranteed-service NoC developed at Phillips Laboratory.

3.3 Real-time Scheduling for Ring-topology NoC

In this section, we will discuss the scheduling problem for NoC with ring topology. An example of NoC of this type is represented in Figure 3.2. In the followings, we will first introduce some terminologies that will be used throughout this section. Then, we will propose a dynamic scheduling algorithm that has better performance than existing works. The algorithm performance is evaluated through simulation and implementation.

Transaction Model

Let routers on the ring-topology NoC be indexed with a unique number in $[1, N^\epsilon]$ where N^ϵ is the number of routers. We define $\mathcal{T}$ as the set of data transactions: $\mathcal{T} = \{\tau_i : i = [1, N]\}$. A data transaction τ_i is characterized by a tuple $\tau_i = (e_i, p_i, \epsilon_i^1, \epsilon_i^2)$ where e_i is the time that the NoC spends to transmit a job of τ_i, p_i is the period of τ_i. Each job must complete within its period, i.e. relative deadlines are equal to periods. ϵ_i^1 and ϵ_i^2, where $\epsilon_i^1 \neq \epsilon_i^2$, are the indexes of the source and destination routers of the transactions. ϵ_i^1, ϵ_i^2 are called the *first* and *second* endpoint of τ_i, respectively. A transaction has two endpoints ϵ_i^1 and ϵ_i^2 if its route uses all consecutive routers from element ϵ_i^1 to ϵ_i^2 in the clockwise direction. Transaction τ_i is said to *go through* element ϵ if $\epsilon \neq \epsilon_i^1$, $\epsilon \neq \epsilon_i^2$ and element ϵ is on the route of τ_i. The NoC utilization u_i of τ_i is calculated as: $u_i = e_i/p_i$. We assume that all data transactions arrive at time 0. Let hyper-period h of $\mathcal{T}$ be the least common multiple of the periods of all transactions in $\mathcal{T}$.

Two transactions are said to *overlap* and can not be transferred concurrently if they use a same link. Given a data transaction set $\mathcal{T}$, we define an overlap indicating function $OV : \mathcal{T} \times \mathcal{T} \mapsto \{0, 1\}$ where $OV(\tau_i, \tau_j) = 1$ if τ_i and τ_j overlap, and 0 otherwise. Figure 3.2 shows a transaction set where τ_1, τ_2, τ_3, and τ_4 overlap each other but they do not overlap τ_7.

A *pairwise overlap set* (PO-set) $\mathcal{D}$ is defined as a maximal subset of $\mathcal{T}$ such that $\forall \tau_i, \tau_j \in \mathcal{D} : OV(\tau_i, \tau_j) = 1$. For convenience, we consider that a transaction that does not overlap any transactions

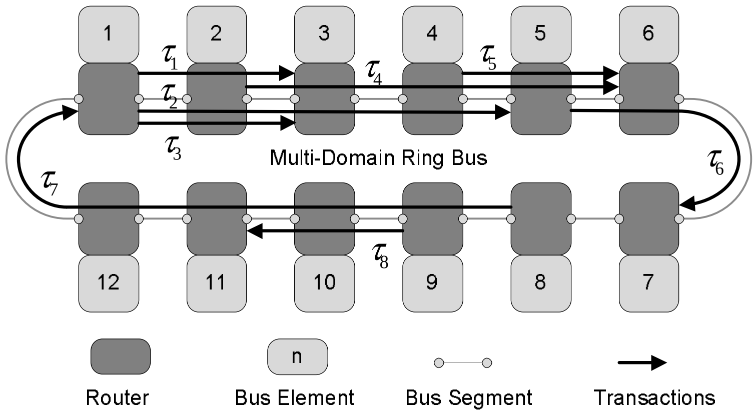


Figure 3.2: Bus Architecture and Acyclic transaction set

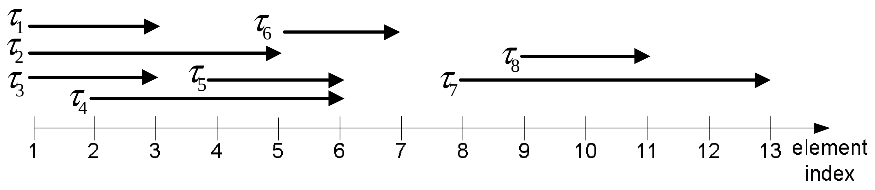


Figure 3.3: Indexed straight line representation

belongs to a PO-set that contains only that transaction. In general a transaction may belong to more than one PO-set. Figure 3.2 shows an example of a transaction set with four PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Let the total number of PO-sets in a transaction set be $N^{\mathcal{D}}$. Since each PO-set contains at least one element different from those of other PO-sets and transactions are arranged in an one dimensional space, $N^{\mathcal{D}} \leq N$.

A transaction set is said to be *acyclic* if there exists a router which has no transaction going through. The transaction set is *cyclic*, otherwise. Note that, the cyclic transaction set creates a cycle on the ring-topology NoC: the cycle comprises of several overlapping transactions. Figure 3.2 shows an example of an acyclic where element 1, 7, and 8 have no transaction going through, whereas Figure 3.4 shows an example of a cyclic transaction set. For ease of identifying the first and second endpoints in the figure, we depict each transaction τ_i as an arrow which always directs from the first endpoint to the second endpoint of τ_i. The direction of the arrow does not imply the direction of the transaction.

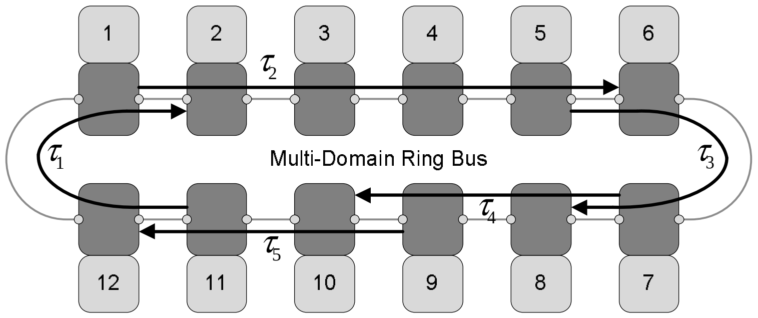


Figure 3.4: Cyclic transaction set

Scheduling Model

We adopt a *slot-based, contention-free* scheduling model similar to the model used in [4, 52] [1]. In this model scheduling decisions are made at integral values, starting from 0. The real interval between time $t \in \mathbb{N}$ and time $t + 1$ i.e. $[t, t + 1)$ is called *slot t*. We assume that every transaction's execution time and period are multiples of slots. Thereafter, we will use a slot as a time unit unless specified otherwise. A schedule S is defined as a function S: $\Gamma \times \mathbb{N} \mapsto \{0, 1\}$ where $S(\tau_i, t) = 1$ if and only if τ_i is scheduled at slot t. A schedule S is *valid* if and only if according to S, it *never* happens that a transaction is scheduled in the same slot together with one or more other transactions that overlap with it.

Given the constraint on overlapping transactions, a necessary condition on the schedulability of a transaction set can be easily derived as in Theorem 3.3.1.

Theorem 3.3.1 *A transaction set $\mathcal{T}$ is schedulable only if:*

$$\forall \mathcal{D} \subseteq \mathcal{T} : u^{\mathcal{D}} = \sum_{\forall \tau_i \in \mathcal{D}} u_i \leq 1 \tag{3.3.1}$$

Proof.
Since, by definition, no two transactions of a PO-set $\mathcal{D}$ can be scheduled concurrently, all transactions of $\mathcal{D}$ must be scheduled in sequence. In other words, the transactions of $\mathcal{D}$ can be considered to be sharing one resource. Therefore, Inequality 3.3.1 must be satisfied. □

Let $E(\epsilon_k)$ be a set of all transactions in $\mathcal{T}$ that use the link between router ϵ_k and $\epsilon_k + 1$. The following lemma is necessary for later discussion.

[1]A hardware implementation of this model is presented in[12].

Lemma 3.3.1 *Given a transaction set $\mathcal{T}$ that satisfies the necessary condition, the following inequality holds.*

$$\sum_{\forall \tau_i \in E(\epsilon_k)} u_i \leq 1$$

Proof.

Since transactions in $E(\epsilon_k)$ pairwise overlap, there exists $\mathcal{D}$ such that $E(\epsilon_k) \subseteq \mathcal{D}$. Therefore the lemma is implied by Theorem 3.3.1. □

Indexed Straight-line Representation of Acyclic Transaction Sets

For ease of presentation, thereafter, we use the indexed straight-line representation described below to model acyclic transaction sets. Given an acyclic transaction set, we select a router which has no transaction going through to be the *first element*. Then, the routers are indexed ascendingly from 1 to N^ϵ in clockwise direction in which the first element's index is 1. Bus elements in Figure 3.2 are indexed following this definition. Since there are no transactions going through element 1, the overlaps between transactions in the acyclic transaction set remains the same if we do the following transformation: 1) let router N^ϵ, instead of connecting to router 1, connect to an additional router which is indexed $N^\epsilon + 1$; 2) change every transaction which has the second endpoint at 1, i.e. $\tau_i = (e_i, p_i, \epsilon_i^1, 1)$, to be $\tau_i = (e_i, p_i, \epsilon_i^1, N^\epsilon + 1)$. For example, in Figure 3.2, $\tau_7 = (e_7, p_7, 8, 1)$ will be changed to be $\tau_7 = (e_7, p_7, 8, 13)$. Since the overlaps between transactions are still the same after the transformation, a valid schedule of the transformed transaction set is also a valid schedule of the original transaction set and vice versa. Given this transformation, the acyclic transaction set can be represented as a set of overlapping line intervals on an indexed straight line where each line interval corresponds to a transaction and the straight line is indexed from 1 to $N^\epsilon + 1$. Figure 3.3 shows the indexed straight line representation of the transaction set shown in Figure 3.2. The following properties are obvious in the straight-line representation of an acyclic transaction set.

Property 3.3.1 *For every transaction τ_i, we have $\epsilon_i^1 < \epsilon_i^2$.*

Property 3.3.2 *For every transaction τ_i and τ_j, τ_i and τ_j overlap if and only if $\epsilon_i^1 < \epsilon_j^2$ and $\epsilon_j^1 < \epsilon_i^2$.*

We study the scheduling problem for acyclic transaction sets in Section 3.3.1. We then extend our solution to cyclic transaction sets in Section 3.3.2.

3.3.1 Scheduling Algorithms for Acyclic Transactions

In this section we present our scheduling algorithms for the proposed real-time transaction sets on the ring-topology NoC. The discussion is divided into two parts.

First, we propose an algorithm, namely POBase, which schedules every acyclic transaction set whose transactions have the *same* period. We will prove that the necessary condition (Theorem 3.3.1) is also the sufficient condition for same-period acyclic transaction set to be schedulable by POBase. Therefore, POBase is optimal (in term of schedulability) for these transaction sets. The algorithm is pseudo-polynomial

Second, a scheduling algorithm, namely POGen, is proposed to schedule acyclic transaction sets whose transactions do not have the same period. POGen, which is built based on POBase, is an online algorithm. POGen can schedule all transaction sets whose PO-set utilizations satisfy the following utilization bound:

$$\forall \mathcal{D} \subseteq \mathcal{T} : u^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}, \tag{3.3.2}$$

where L is defined as the greatest common divisor (GCD) of all transaction periods measured in number of slots. Note that the bound approaches 0 when L is small; for example, when transaction periods are mutually prime numbers. The bound, however, approximates 1 when L is large. We believe that this assumption holds in most practical real-time applications [27]. As we will show in the implementation section, with the speed of the state of the art many-core SoC [2], the practical slot size is about $100us$ to $10us$ (which is also the size of a time unit in our definition). Meanwhile, the periods in practical real-time applications [27] are usually measured in millisecond units. Therefore, the smallest value in time unit of the GCD of all transaction periods is $1ms$. Because $1ms = 10 \times 100us = 100 \times 10us$, we have that L has practical values ranging from 10 to 100 slots. This results in the utilization bound between 0.9 and 0.99. We also note that this bound is only a sufficient bound and we plan to improve this bound in our future works.

The POBase algorithm

The problem of acyclic same-period transaction set scheduling is similar to the Interval Graph Coloring Problem (IGCP) [20] (see Chapter 4.1). An interval graph is a graph constructed from a set of intervals on the real line where each vertex represents an interval and there is an edge between two vertices if the two corespondent intervals overlap. The IGCP is the problem of assigning a color in a minimum set of colors to each vertex in the interval graph such that two adjacent vertices do not have a same color. We note that the IGCP is a special case of the our problem and the coloring algorithm in [20] can only handle this special

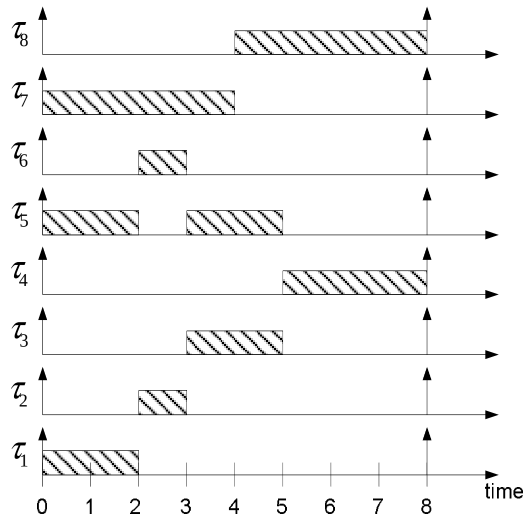


Figure 3.5: An example of the POBase algorithm

case. Our proposed algorithm POBase is a new algorithm to solve the problem at hand. POBase is a first-fit algorithm with respect to a transaction ordering. More specifically, in POBase, the transactions are ordered ascendingly by their first endpoint (stored in list $\mathcal{L}$). Then, each transaction in $\mathcal{L}$ is assigned to the earliest slots where no smaller-ordered overlapping transaction has been already assigned to[2]. This condition is enforced by the use of array lastEndpoint (Step 6). lastEndpoint has size equal to the transactions' period p. The initial values of all items of lastEndpoint is 1 which is also the smallest index of the routers. Except for the initial value, during the algorithm execution, the value of item t of lastEndpoint will be the second endpoint (Step 8) of the last transaction that has been assigned to slot t. Since transactions are being assigned in ascending order of their first endpoints, if condition $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$ in Step 6 is satisfied then τ_i does not overlap with all transactions that have been assigned to t before τ_i. We will formally prove this statement in Lemma 3.3.3. This proof requires Lemma 3.3.2. Finally, the proof of POBase's optimality will be shown in Theorem 3.3.2.

Figure 3.5 shows an example of the schedule generated by POBase for the transaction set shown in Figure 3.2 whose transactions have period equal to 8 and execution times: $e_1 = 2, e_2 = 1, e_3 = 2, e_4 = 3, e_5 = 4, e_6 = 1, e_7 = 4, e_8 = 4$. Consider the schedule of transactions of $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$. τ_5 is scheduled in slots $\{0, 1, 3, 4\}$, because its smaller-ordered overlapping transactions τ_2 and τ_4 are scheduled

[2]The transactions can also be ordered by their second endpoint and the schedule is generated in descending ordered of the order list.

in slots $\{2, 5, 6, 7\}$.

Algorithm 6 POBase

Input: $\mathcal{T}$ such that $\forall \tau_i \in \mathcal{T} : p_i = p$

Output: schedule S for period p

```
1:  $\mathcal{L} \leftarrow$ list of $\forall \tau_i \in \mathcal{T}$ in ascending order of $\epsilon_i^1$
2:  $\forall t \in [0, p)$ : $\mathsf{lastEndpoint}[t] \leftarrow 1$
3:  for each $\tau_i \in \mathcal{L}$ do
4:      $r \leftarrow e_i$
5:      for each $t \in [0, p)$ do
6:          if $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$ then
7:              $S(\tau_i, t) \leftarrow 1$
8:              $\mathsf{lastEndpoint}[t] \leftarrow \epsilon_i^2$
9:              $r \leftarrow r - 1$
10:             if $r = 0$ then
11:                 break //complete schedule assignment of $\tau_i$
```

Lemma 3.3.2 *At each iteration of Step 6, for every τ_j which has been assigned to slot t, $\epsilon_j^2 \leq \mathsf{lastEndpoint}[t]$.*

Proof.

We prove by induction.

Base case: Consider the first iteration, the lemma holds because there is not any transaction being assigned.

Induction case: Assume that the lemma holds at iteration k where τ_j is being assigned, we will prove that it also holds at iteration $k + 1$. Consider slot t to which τ_j is assigned. Due to the condition at Step 6, we have $\mathsf{lastEndpoint}[t] \leq \epsilon_j^1$. Then by the induction assumption, we have: for every τ_l that has been assigned to t before iteration k, $\epsilon_l^2 \leq \epsilon_j^1$. Furthermore, if τ_j is assigned to t, we have $\mathsf{lastEndpoint}[t] = \epsilon_j^2$ after Step 8. Since by Property 1 of the indexed straight-line presentation, $\epsilon_j^1 < \epsilon_j^2$, we have the lemma holds at iteration $k + 1$. □

Lemma 3.3.3 *Slot t has not been assigned to any overlapping transaction of τ_i if and only if $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$.*

Proof.

Necessary condition: We prove this condition by showing that if $\mathsf{lastEndpoint}[t] > \epsilon_i^1$ then slot t has been assigned to a transaction that overlaps τ_i. Since $\mathsf{lastEndpoint}[t] > \epsilon_i^1$, there must exist a transaction τ_j that has been assigned to t before τ_i where $\epsilon_j^2 = \mathsf{lastEndpoint}[t] > \epsilon_i^1$. And since τ_j has been assigned to the slot before τ_i, $\epsilon_i^1 \geq \epsilon_j^1$. Therefore, we have: $\epsilon_j^2 > \epsilon_i^1$ and $\epsilon_i^2 > \epsilon_j^1$. Then, by Property 2 of the indexed straight-line presentation, τ_i and τ_j overlap.

Sufficient condition: If $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$, then by Lemma 3.3.2 we have that: for every τ_j that has been assigned to slot t before τ_i, $\epsilon_j^2 \leq \mathsf{lastEndpoint}[t] \leq \epsilon_i^1$. Therefore, by Property 2 of the indexed straight-line presentation, τ_j and τ_i do not overlap. $\square$

Theorem 3.3.2 *If the same-period acyclic transaction set $\mathcal{T}$ satisfies the necessary condition 3.3.1, then* POBase *generates a feasible schedule for $\mathcal{T}$.*

Proof.

The generated schedule is valid because a transaction is not scheduled in the same slot with its overlapping transactions (Lemma 3.3.3) . It remains to show that if a transaction set satisfies the necessary condition, then at the end of the algorithm,

$$\forall \tau_i : \sum_{x \in [0,p)} S(\tau_i, x) = e_i. \tag{3.3.3}$$

We will prove this by induction.

Base case: Consider the first iteration of the for-loop starting at Step 3. In this iteration, the schedule of τ_1 in $\mathcal{L}$ is generated. Since all items of $\mathsf{lastEndpoint}$ have value 1, the condition at Step 6 is satisfied for every t. Furthermore, we have $e_1 \leq p$. Therefore, at the end of the iteration, Equation 3.3.3 must hold for τ_1 i.e. $\sum_{x \in [0,p)} S(\tau_1, x) = e_1$.

Induction case: Assume after iteration k of the for-loop starting at Step 3, Equation 3.3.3 holds for all transactions $\{\tau_i : i \in [1, k]\}$. We will prove that Equation 3.3.3 also holds for τ_{k+1} after iteration $k + 1$. By contradiction, assume that at the end of the iteration $k + 1$, $\sum_{x \in [0,p)} S(\tau_{k+1}, x) < e_{k+1}$. Let $E(\epsilon_{k+1}^1)$ be the set of transactions that use the link between router ϵ_{k+1}^1 and $\epsilon_{k+1}^1 + 1$. By the way the transactions are ordered and Property 2 of the indexed straight-line presentation, we have that $\forall \tau_i \in \mathcal{T}$ if the schedule of τ_i has been generated before τ_{k+1} and τ_i overlaps τ_{k+1} then $\epsilon_i^1 \leq \epsilon_{k+1}^1 < \epsilon_i^2$. Therefore $\tau_i \in E(\epsilon_{k+1}^1)$. In other words, among all the transactions that overlap with τ_{k+1}, only transactions in $E(\epsilon_{k+1}^1)$ have their schedule generated. Therefore, the contradiction assumption occurs only when:

$$\sum_{\tau_i \in E(\epsilon_{k+1}^1)} \sum_{x \in [0,p)} S(\tau_i, x) = p. \tag{3.3.4}$$

Since the following is true:

$$\forall \tau_i \in E(\epsilon_{k+1}^1) \setminus \{\tau_{k+1}\} : \sum_{x \in [0,p)} S(\tau_i, x) = e_i,$$

by the contradiction assumption and Equation 3.3.4 we have: $\sum_{\tau_i \in E(\epsilon_{k+1}^1)} e_i > p$. This contradicts with Lemma 3.3.1 which implies that $\sum_{\tau_i \in E(\epsilon_{k+1}^1)} e_i \leq p$. Therefore, at the end of the iteration, Equation 3.3.3 must hold for τ_{k+1}. This completes the proof. □

POBase Analysis: If an efficient sorting algorithm is used at Step 1, the time complexity of this step will be $O(N \log(N))$. Furthermore, Step 6 to 11 require constant number of operations. Therefore the time complexity of POBase to build a schedule of p slots (where p is the common period) for N transactions is $O(N * \max(\log(N), p))$. We note that the time complexity of POBase is pseudo-polynomial, and when $p \geq N$, it is equivalent to that of PFair [4] and BoundaryFair [52], which takes $O(N * p)$ to generate the schedule of p slots.

The POGen algorithm

In this subsection we propose an online scheduling algorithm (POGen) for acyclic transaction sets whose transactions do not have the same period. Our proposed solution is inspired by the work in [52], in which the execution time line is divided into intervals and the number of slots in each interval which is assigned to a task is proportional to the task's utilization. However, since work in [52] does not have the transaction overlap assumption, their proposed algorithms can not be used for the problem at hand.

In POGen, the execution time line from 0 to the hyper-period h, i.e. $[0, h)$, is divided into a set of consecutive *scheduling intervals*: $\{\mathsf{int}^k = [t^k, t^{k+1}) : k \in \mathbb{N} \wedge 0 \leq t^k < t^{k+1} < h\}$. Let $|\mathsf{int}^k| = t^{k+1} - t^k$. Scheduling intervals must respect two fundamental properties: 1) the arrival time (also deadline) of any transaction must coincide with the finishing time of a scheduling interval and the start time of the next one; 2) the minimum length of any scheduling interval must be at least L where L is the greatest common divisors

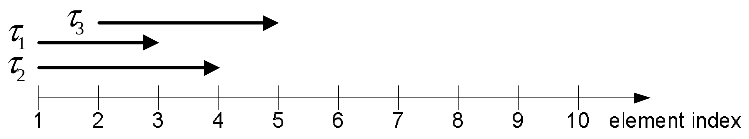


Figure 3.6: Example of three transactions

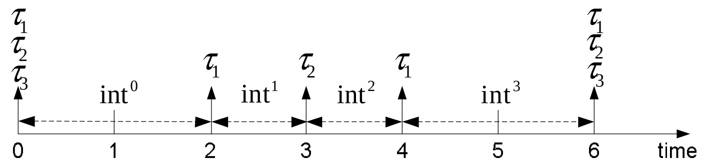


Figure 3.7: Scheduling intervals on the execution time line

of all transaction periods. As we will show in Theorem 3.3.3, the second property is essential to induce a feasible utilization bound: intuitively, longer scheduling intervals allow POGen to better approximate the fluid scheduling model. There are multiple feasible assignments of scheduling intervals that respect the two properties. For example, Figure 3.7 shows the scheduling intervals induced by the set of three transactions shown in Figure 3.6, where $\tau_1 = \{e_1 = 1, p_1 = 2, \epsilon_1^1 = 1, \epsilon_1^2 = 3\}$, $\tau_2 = \{e_2 = 1, p_2 = 3, \epsilon_2^1 = 1, \epsilon_2^2 = 4\}$ and $\tau_3 = \{e_3 = 1, p_3 = 6, \epsilon_3^1 = 2, \epsilon_3^2 = 5\}$. In this example, the scheduling intervals are the intervals between two closest arrival times of any two transactions. Note that by definition of L and since all transactions arrive at time 0, it follows that the minimum length of scheduling intervals in the example is indeed L. An alternative feasible definition for scheduling intervals consists in assigning $t^k = k\mathsf{L}$, e.g. all scheduling intervals have fixed length L. By definition of L, it then follows that the arrival time of any transaction coincides with the start time t^k of some interval int^k. In the rest of this section, we will not restrict ourselves to any specific interval assignment, instead only assuming that scheduling intervals respect the two fundamental properties.

In each scheduling interval int^k, each transaction τ_i is assigned an *interval load* l_i^k which is the number of slots in the interval allocated to schedule τ_i. The interval loads of each transaction are calculated such that at the end of each interval, the transaction's execution approximates its execution in the fluid scheduling model [15]. The interval load of a PO-set is the sum of the interval loads of its transactions. Given the interval loads of all transactions in interval int^k, POBase is used to generate the schedule of int^k.

As shown in the previous subsection, the interval schedule given by POBase will be feasible if and only if:

$$\forall \mathcal{D} \subseteq \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} l_i^k \leq |\mathsf{int}^k|.$$

A schedule of a transaction set, which is generated by POGen, is feasible if it satisfies the following two conditions:

Condition 3.3.1 *for each transaction τ_i, the sum of the interval loads over the transaction period is equal to e_i.*

Condition 3.3.2 *there is a feasible schedule for every scheduling interval.*

In the following paragraphs, we will discuss our solution to identify the scheduling intervals and the interval loads which induces a feasible schedule.

With regard to the interval loads, we define for each transaction τ_i and scheduling interval int^k a lag function:

$$\mathsf{lag}(\tau_i, \mathsf{int}^k) = u_i * t^{k+1} - \sum_{x \in [0, t^k)} S(\tau_i, x).$$

The function calculates how much time τ_i must be executed in interval int^k such that at the end of int^k it is scheduled according to the fluid scheduling model [15]. We also define for each PO-set $\mathcal{D}$ a similar lag function:

$$\mathsf{lag}(\mathcal{D}, \mathsf{int}^k) = u^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x).$$

The goal of POGen is to generate a *feasible load set* for each interval int^k, that is, a set of transaction loads that satisfy the following inequalities:

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil, \tag{3.3.5}$$

$$\begin{aligned} \forall \mathcal{D} \subseteq \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor &\leq \sum_{\tau_i \in \mathcal{D}} l_i^k \\ &\leq \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rceil). \end{aligned} \tag{3.3.6}$$

Inequality 3.3.5 sets conditions on the interval load for each transaction, based on the closest integral values of the lag functions. Inequality 3.3.6 sets conditions on the total interval load of each PO-set. Note that the right side of Inequality 3.3.6 guarantees that each PO-set with feasible loads is schedulable in int^k by POBase, that is, Condition 2 is satisfied for int^k. Similarly, if all loads satisfy the lower bounds of

Inequalities 3.3.5, then the generated schedule satisfies Condition 1. The reason is as follows. Consider the last scheduling interval of a period of transaction τ_i: $\mathsf{int} = [t, a * p_i)$ where t and a are some integers, the lag function of τ_i is:

$$\mathsf{lag}(\tau_i, \mathsf{int}) = a * u_i * p_i - \sum_{x \in [0,t)} S(\tau_i, x).$$

Since $u_i * p_i = e_i$ is an integer, and so is $S(\tau_i, x)$, $\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor = \mathsf{lag}(\tau_i, \mathsf{int})$. That means the total interval load of τ_i up to slot $a * p_i$, which is calculated as:

$$\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor + \sum_{x \in [0,t)} S(\tau_i, x),$$

is equal to $a * e_i$ and satisfies Condition 1. However using only the lower bound loads does not guarantee that Inequality 3.3.6 can be satisfied at the same time. This is also true if only upper bound loads are used. The following example illustrates this point. Consider again the example of the transaction set in Figure 3.6 and 3.7. If the algorithm runs with interval loads to be their lower bound loads, then we have the sets of interval loads in four intervals $\mathsf{int}^0, \mathsf{int}^1, \mathsf{int}^2, \mathsf{int}^3$ respectively are $\{l_1^0 = 1, l_2^0 = 0, l_3^0 = 0\}$, $\{l_1^1 = 0, l_2^1 = 1, l_3^1 = 0\}$, $\{l_1^2 = 1, l_2^2 = 0, l_3^2 = 0\}$, $\{l_1^3 = 1, l_2^3 = 1, l_3^3 = 1\}$. This means the schedule in int^3 is not feasible because its total load is 3 while $|\mathsf{int}^3| = 2$. If otherwise, the upper bound loads are used only, then the schedule of interval int^0 is also not feasible because the total load in this interval is 3. An algorithm that generates feasible schedules must use a combination of these values and computing this is not trivial.

POGen achieves this feature by iteratively computing a feasible load set for each scheduling interval. It is an online algorithm which is invoked at the beginning of each interval and generates the schedule for that interval. In Lemma 3.3.4, we first show that the following inequalities initially hold at the beginning of the first interval int^0:

$$\forall \mathcal{D} \subseteq \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq |\mathsf{int}^k|, \tag{3.3.7}$$

$$\forall \mathcal{D} \subseteq \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0,t^k)} S(\tau_i, x) \geq \lfloor u^{\mathcal{D}} * t^k \rfloor. \tag{3.3.8}$$

A feasible load set is then computed in Step 1 of POGen by the GenerateLoad procedure, which is assumed to satisfy the following proposition:

Proposition 3.3.1 *Assume that all PO-sets satisfy the utilization bound in Inequalities 3.3.2. If Inequalities 3.3.7, 3.3.8 hold for* int^k*, then* GenerateLoad *computes a feasible load set for* int^k.

Given a feasible load set for interval int^0, Lemma 3.3.4 guarantees that Inequalities 3.3.7, 3.3.8 again hold for int^1. Hence, in the next execution of POGen at t^1, GenerateLoad can compute a feasible load set for int^1, and so on and so forth for all scheduling intervals in the hyper-period. Since a feasible load set is obtained for all scheduling intervals, Condition 1 and Condition 2 are satisfied and thus POGen generates a feasible schedule of $\mathcal{T}$. In the next Section 3.3.1, we will prove that GenerateLoad indeed satisfies Proposition 3.3.1.

Algorithm 7 POGen

Input: transaction set $\mathcal{T}$, interval int^k

Output: schedule S for int^k

```
1: $\{l_i^k : \forall i \in [1, N]\} \leftarrow$ GenerateLoad($\mathcal{T}$,$\mathsf{int}^k$)
2: $\mathcal{T}' \leftarrow \{\{l_i^k, |\mathsf{int}^k|, \epsilon_i^1, \epsilon_i^2\} : \forall i \in [1, N]\}$
3: $S$ for $\mathsf{int}^k \leftarrow$ POBase($\mathcal{T}'$)
```

Lemma 3.3.4 *If* GenerateLoad *satisfies Proposition 3.3.1, then Inequalities 3.3.7, 3.3.8 hold for every scheduling intervals.*

Proof.

We prove by induction.

Base step: Consider the first scheduling interval $\mathsf{int}^0 = [0, t^1)$. Inequalities 3.3.7 for this interval hold because

$$\forall \mathcal{D} \subseteq \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^0) \rfloor = \lfloor u^{\mathcal{D}} * t^1 \rfloor \leq |\mathsf{int}^1|,$$

and Inequalities 3.3.8 hold because

$$\forall \mathcal{D} \subseteq \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0,0)} S(\tau_i, x) = 0 = \lfloor u^{\mathcal{D}} * 0 \rfloor.$$

Induction step: Assume that Inequalities 3.3.7, 3.3.8 hold in every scheduling interval up to int^k. We prove that Inequalities 3.3.7, 3.3.8 also hold before the execution of GenerateLoad at interval int^{k+1}. Since Inequalities 3.3.7, 3.3.8 are satisfied at interval int^k, GenerateLoad generates a feasible load set and POBase generates a feasible schedule for the interval. Therefore after Step 3, we have:

$$\forall \mathcal{D} \subseteq \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [t^k, t^{k+1})} S(\tau_i, x) = \sum_{\tau_i \in \mathcal{D}} l_i^k.$$

Then by the left side of Inequalities 3.3.6, we obtain the following which proves that Inequalities 3.3.8 hold for int^{k+1}.

$$\begin{aligned}
&\forall \mathcal{D} \subseteq \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&= \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) + \sum_{\tau_i \in \mathcal{D}} l_i^k \\
&\geq \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) + \\
&\left\lfloor u^{\mathcal{D}} * t^{k+1} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k})} S(\tau_i, x) \\
&= \lfloor u^{\mathcal{D}} * t^{k+1} \rfloor
\end{aligned}$$

Now consider Inequalities 3.3.7. Notice that since $S(\tau_i, x)$ is integer, we have:

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor \\
&= \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x).
\end{aligned}$$

Since Inequalities 3.3.8 hold for int^{k+1}, Inequalities 3.3.7 also hold because (the last inequality holds because of Inequality 3.3.2):

$$\begin{aligned}
&\forall \mathcal{D} \subseteq \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor \\
&= \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&\leq \lfloor u^{\mathcal{D}} * t^{k+2} \rfloor - \lfloor u^{\mathcal{D}} * t^{k+1} \rfloor \\
&\leq \lceil u^{\mathcal{D}} * (t^{k+2} - t^{k+1}) \rceil \leq |\mathsf{int}^{k+1}|.
\end{aligned}$$

This completes the proof. □

The GenerateLoad procedure

As we mentioned, procedure GenerateLoad searches for a feasible load set of each scheduling interval. There are two questions that have to be answered: (Q1) is there a feasible load set? (Q2) is there an efficient algorithm to find it? We will show that the problem at hand is equivalent to the problem of *Circulations in flow Graphs with Demands and Lower bounds* (CGDL) [20] where all the demands are zero. A flow graph is a directed graph where each edge has a capacity and there is a flow going through each edge in its specified direction. The magnitude (a.k.a value) of a flow must be smaller than the capacity of the edge it goes through. If each edge also has a lower bound then the flow value must be higher than this bound. Furthermore, each vertex may have a demand d. The values of the flows must satisfy the *conservation constraint* that is: the total value of the flows entering a vertex must be d units higher than that of flows exiting the same vertex. The CGDL problem is the problem of finding a set of flows that satisfies all the constraints (a.k.a feasible flows) in a given graph. It has been proved in [20] that the Ford-Fulkerson algorithm can be used to generate a set of feasible flows in a CGDL problem if there exists one. More specifically, if the graph has integer edge capacities, lower bounds and vertex demands, then the generated feasible flows are also integers. The Ford-Fulkerson algorithm is a well-known algorithm in graph theory which is originally designed to solve the max-flow min-cut problem [20].

Having proved that the problem of searching for a feasible load set is equivalent to the CGDL problem, we will then prove that if the utilization of each PO-set is smaller than the utilization bound expressed by Inequalities 3.3.2, there always exists a feasible solution therefore answering Q1. Then, since the Ford-Fulkerson algorithm [20] can be used to solve the problem, Q2 is also answered.

In the following, we will intuitively describe the construction of a directed graph from the input of $\mathsf{GenerateLoad}$. Each vertex of the constructed graph represents a PO-set $\mathcal{D}_j$. We define for each vertex, a *PO-set edge* $g_j^{\mathcal{D}}$ whose real-valued flow $f_j^{\mathcal{D}}$ represents the interval load of the corresponding PO-set. An integer-valued lower bound $b_j^{\mathcal{D}}$ and an integer-valued capacity $c_j^{\mathcal{D}}$ are defined for each of the PO-set edges such that Inequalities 3.3.6 are imposed on their flow values, i.e.:

$$\forall \mathcal{D}_j \subseteq \mathcal{T} : b_j^{\mathcal{D}} \leq f_j^{\mathcal{D}} \leq c_j^{\mathcal{D}}. \tag{3.3.9}$$

where

$$\begin{aligned} b_j^{\mathcal{D}} &= \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor, \\ c_j^{\mathcal{D}} &= \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rceil). \end{aligned}$$

Furthermore, for each transaction τ_i, a *transaction edge* is defined whose real-valued flow f_i represents the interval load of the corresponding transaction. A lower bound value b_i and a capacity c_i are defined for each of the transaction edges such that Inequalities 3.3.5 are imposed on their flow values:

$$\forall \tau_i \in \mathcal{T} : b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq f_i \leq c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil. \tag{3.3.10}$$

The flow of a transaction edge entering a vertex represents the contribution of the corresponding transaction's interval load to the corresponding PO-set's interval load. The endpoints and the direction of each edge are defined in such a way that the values of the flows in and out a vertex preserve the relationship between the interval load of the corresponding PO-set and that of its transactions. The graph has a feasible circulation flow which represents a feasible load set.

The following definition is necessary for the graph construction. Let the *index PO-set order* of a transaction set $\mathcal{T}$ be an ordered list of all PO-sets in $\mathcal{T}$ where PO-set $\mathcal{D}$ with smaller $\min_{\tau_i \in \mathcal{D}_j} \epsilon_i^2$ has smaller index. Ties are broken arbitrarily. Since each PO-set has only one value $\min_{\tau_i \in \mathcal{D}_l} \epsilon_i^2$, the order is well-defined. The transaction set in Figure 3.2 has the index PO-set order be $\{\mathcal{D}_j : j \in [1,4]\}$ where $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Figure 3.8 shows the graph G constructed from the transaction set in Figure 3.2. Transaction edges are represented by solid lines while PO-set edges are represented by dotted lines.

Graph construction: let us define a tuple $G = (V, E)$ as follows:

- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a vertex v_j.
- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a directed edge $g_j^{\mathcal{D}}$ with capacity $c_j^{\mathcal{D}} = \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rceil)$ and lower bound $b_j^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor$. Let $g_j^{\mathcal{D}}$ be a *PO-set edge*.
- For each transaction τ_i, define a directed edge g_i with capacity $c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$, and lower bound $b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor$. Let g_i be a *transaction edge*.
- $\{g_i : \tau_i \in \mathcal{D}_1\}$ are edges that enter v_1; $g_1^{\mathcal{D}}$ is an edge that exits v_1.
- $\forall j : 1 < j \leq N^{\mathcal{D}}$, $\{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\}$ and $g_{j-1}^{\mathcal{D}}$ are edges that enter v_j; $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ and $g_j^{\mathcal{D}}$ are edges that exits v_j. This construction step deals with the situation where two PO-sets $\mathcal{D}_{j-1}, \mathcal{D}_j$ share some transactions. More specifically, to preserve the relationship between the interval loads of the PO-sets and that of its transactions, the transaction edge of a transaction common to two PO-sets would have to enter the two corresponding vertexes v_{j-1}, v_j. Since in a qualified graph, each directed edge can enter at most one vertex, this situation must be avoided. This can be accomplished by representing the interval loads of the common transactions on the second PO-set (v_j) as the interval load of the first PO-set (i.e., $g_{j-1}^{\mathcal{D}}$ enters v_j) minus the interval load of the transactions that are only in the first set (i.e., $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ exit v_j). Lemma 3.3.5 will detail the proof of this argument.
- $V = \{v_j : j \in [1, N^{\mathcal{D}}]\}$

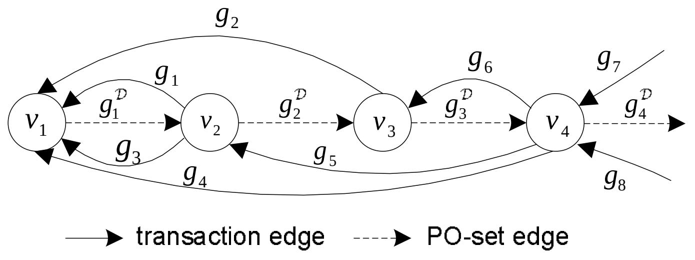


Figure 3.8: Constructed graph G

- $E = \{g_i : j \in [1, N]\} \cup \{g_j^{\mathcal{D}} : j \in [1, N^{\mathcal{D}}]\}$

Finally, the graph flow is subject to the flow conversation constraint [20] in which given a vertex, the sum of the flow values entering it minus the sum of the flow values exiting it is zero. As a graph construction example, consider PO-set $\mathcal{D}_2$. Vertex v_2 has an output PO-set edge $g_2^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_2$. Since $\mathcal{D}_1$ has τ_2 and τ_4 in common with $\mathcal{D}_2$ but not τ_1 and τ_3, v_2 has an input PO-set edge $g_1^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_1$ and two output transaction edges g_1 and g_3 that represent the interval loads of τ_1 and τ_3, respectively. Furthermore, note that edges g_2 and g_4 for the common transactions τ_2 and τ_4 enter v_1, not v_2. g_2 exits v_3, but g_4 exits v_4 because τ_4 also belongs to $\mathcal{D}_3$. Finally v_2 has an input transaction edge g_5 that represents the interval load of τ_5. Since τ_5 is in $\mathcal{D}_2$ and $\mathcal{D}_3$ but not $\mathcal{D}_1$ and $\mathcal{D}_4$, g_5 exits v_4. Lemma 3.3.5 shows that G is indeed a directed graph.

Lemma 3.3.5 *G is a directed graph.*

Proof.

Since every edge of G is directed, it remains to show that each edge has only one or two endpoints. There is one edge defined for each PO-set and one edge defined for each transaction.

For each PO-set $\mathcal{D}_j$, the PO-set edge g_j^D exits only v_j. In addition, g_j^D enters only v_{j+1} when $j < N^{\mathcal{D}}$. Therefore each PO-set edge exits exactly one vertex and enters at most one vertex.

By the index PO-set ordering, if $\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}$, then $\tau_i \notin \mathcal{D}_k \setminus \mathcal{D}_{k-1}$ where $j < k \leq N^{\mathcal{D}}$. Therefore, the elements of the following set are disjoint: $\mathcal{A} = \big\{\{g_i : \tau_i \in \mathcal{D}_1\}, \{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\} : j \in (1, N^{\mathcal{D}}]\big\}$. By definition, $\mathcal{A}$ contains the transaction edges of G that enter some vertices. Also the union of the elements of $\mathcal{A}$ is $\{g_i : \tau_i \in \mathcal{T}\}$. Therefore, each transaction edges enters exactly one vertex.

By a similar proving technique, we can show that each transaction edge exits at most one vertex. Due to space constraints, we skip the detailed proof. In conclusion, every edge of G has at most two endpoints and is directed. □

It remains to show that GenerateLoad satisfies Proposition 3.3.1 and therefore POGen generates feasible schedules for all transaction sets that satisfy the utilization bound of Inequalities 3.3.2. For simplicity of exposition, we split the proof in multiple lemmas. First, Lemma 3.3.6 proves an important property of graph G regarding the flow values. Then, this property will be used to prove in Lemma 3.3.7 that graph G has a feasible flow set if Inequalities 3.3.7, 3.3.8 are satisfied for interval int^k and furthermore all PO-sets satisfy an utilization constraint based on $|\mathsf{int}^k|$. Note that we know from [20] that if graph G has a feasible flow set, then it has an integral feasible flow set (i.e. the values of all flows in the set are integers) which can be found by the Ford-Fulkerson algorithm [20]. Therefore, to complete the proof, we will have to prove that a feasible load set can be derived from an integral feasible flow set of G (Lemma 3.3.8). Finally, we will show that the utilization bound of Inequalities 3.3.2 implies the utilization bound used in Lemma 3.3.7. Hence, Proposition 3.3.1 holds.

Lemma 3.3.6 *A flow in graph G satisfies the flow conservation constraint at every vertex v_j if and only if the following equalities hold for every PO-set $\mathcal{D}_j$:*

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}. \tag{3.3.11}$$

Proof.

Proof of Lemma 3.3.6: We prove this by induction over the ordered set of vertices.

<u>Base case</u>: By the graph construction rules regarding the edges that enter and exit v_1, the flow conservation constraint holds at vertex v_1 if and only if:

$$\sum_{\tau_i \in \mathcal{D}_1} f_i = f_1^{\mathcal{D}}$$

<u>Induction case</u>: Assume that the lemma holds for every PO-set from $\mathcal{D}_1$ to $\mathcal{D}_{j-1}$. We will prove that the lemma also holds for $\mathcal{D}_j$. By the graph construction rules regarding the edges that enter and exit v_j and the definition of the flow conservation constraint at vertex v_j, we have

$$\sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i = \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}}.$$

Since

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = \sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i,$$

we have the flow conservation constraint holds at vertex v_j if and only if

$$\begin{aligned}\sum_{\tau_i \in \mathcal{D}_j} f_i &= \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i \\ &= \sum_{\tau_i \in \mathcal{D}_{j-1}} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}}\end{aligned}$$

Finally, by the induction hypothesis, the above equation is equivalent to

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_{j-1}^{\mathcal{D}} + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} = f_j^{\mathcal{D}}$$

This completes the proof. □

Lemma 3.3.7 *There exists a feasible flow set in graph G if Inequalities 3.3.7, 3.3.8 are satisfied for interval* int^k *and furthermore the PO-set utilizations satisfy the following condition.*

$$\forall \mathcal{D}_j \subseteq \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{|\mathsf{int}^k| - 1}{|\mathsf{int}^k|} \tag{3.3.12}$$

Proof.

First note that Inequalities 3.3.7 are necessary for the edge constraints on each PO-set edge (Inequality 3.3.9) to be satisfied. Let us construct a flow set as follows.

$$\begin{aligned}&\forall \tau_i \in \mathcal{T} : f_i = \mathsf{lag}(\tau_i, \mathsf{int}^k) \\ &\forall \mathcal{D}_j \subseteq \mathcal{T} : f_j^{\mathcal{D}} = \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k)\end{aligned}$$

We will have to prove that the constructed flow set satisfies the edge constraints and the flow conservation constraints. Given the constructed flow, it is easy to verify that the edge constraints of each transaction edge (Inequality 3.3.10) and the left-side edge constrains of each PO-set edge (Inequality 3.3.9) are satisfied. The right-side edge constraints of each PO-set edge are satisfied because by the definition of the lag function and by Inequalities 3.3.8, before the execution of $\mathsf{GenerateLoad}$ for interval int^k we have the following:

$$\begin{aligned}&\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) = u_j^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_j} \sum_{x \in [0, t^k)} S(\tau_i, x) \\ &\leq u_j^{\mathcal{D}} * t^{k+1} - \lfloor u_j^{\mathcal{D}} * t^k \rfloor \\ &< u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1.\end{aligned}$$

Now by Inequalities 3.3.12, the following holds:

$$\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) < u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1 \leq |\mathsf{int}^k|.$$

It remains to verify that the flow conservation constraint is honored at every vertex. Since the constructed flow set satisfied Equation 3.3.11, the sufficient condition of Lemma 3.3.6 proves this statement. □

Lemma 3.3.8 *If there is an integral feasible flow set in graph G, then there is a feasible load set where* $\forall \tau_i \in \mathcal{T} : l_i^k = f_i$ *.*

Proof.
Given an integral feasible flow set, $\forall \tau_i \in \mathcal{T}$ let $l_i^k = f_i$. The following inequality holds thanks to Inequalities 3.3.10.

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$$

Thus the interval loads satisfy Inequality 3.3.5. We now have to prove that the interval loads also satisfy Inequality 3.3.6. By the necessary condition of Lemma 3.3.6, we have $\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}$. Then since $\sum_{\tau_i \in \mathcal{D}_j} l_i^k = \sum_{\tau_i \in \mathcal{D}_j} f_i$ and $f_j^{\mathcal{D}}$ is subject to PO-set edge constraints in Inequality 3.3.9, Inequality 3.3.6 is satisfied. □

Theorem 3.3.3 *The acyclic transaction set* $\mathcal{T}$ *is schedulable by* POGen *if:*

$$\forall \mathcal{D}_j \subseteq \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}.$$

Proof.
Since $\mathsf{L} \leq \min_k(|\mathsf{int}^k|)$, Inequalities 3.3.12 hold. Assume that Inequalities 3.3.7, 3.3.8 hold for a specific interval int^k. Then by Lemma 3.3.7 and [20], the constructed graph G has an integral feasible flow set. Hence, by Lemma 3.3.8 algorithm GenerateLoad computes a feasible load set, which proves Proposition 3.3.1. Since furthermore, according to Lemma 3.3.4, Inequalities 3.3.7, 3.3.8 hold for every interval int^k, it follows that Inequalities 3.3.5 and 3.3.6, and therefore feasibility Conditions 1 and 2, also hold for every interval. This concludes the proof. □

Algorithm analysis: Since $\forall g_i \in E : c_i - b_i \leq 1$ and $\forall g_j^{\mathcal{D}} \in E : c_j^{\mathcal{D}} - b_j^{\mathcal{D}} \leq 1$, we have $\Delta = \sum_{g_i \in E} (c_i - b_i) + \sum_{g_j^{\mathcal{D}} \in E} (c_j^{\mathcal{D}} - b_j^{\mathcal{D}}) \leq N + N^{\mathcal{D}} \leq 2N$. The time complexity of the Ford-Fulkerson algorithm in finding a feasible circulation in graph G is $O(|E| * f^{max})$ where f^{max} is the maximum flow value of a graph derived from G and $f^{max} \leq \Delta$ (see [20] for details). Since $\Delta \leq 2N$, the time complexity of GenerateLoad is $O(N^2)$. Finally, since the time complexity of POBase is $O(N * \max(\log(N), |\mathsf{int}^k|))$, the time complexity of POGen to generate the schedule for $|\mathsf{int}^k|$ slots is $O(N * \max(N, \max_k |\mathsf{int}^k|))$. The worst-case time complexity to generate the schedule for a hyper-period is $O(N^2 * h)$, which occurs when every scheduling interval has size 1.

3.3.2 Scheduling Algorithms for Cyclic Transaction Sets

The cyclic transaction set scheduling problem is NP-complete because the special case where all transmission times are the same and all periods are equal is equivalent to the Circular-Arc Coloring Problem (CACP) [20](Chapter 10.3). A Circular-Arc is a set of overlapping arcs that create a cycle. The CACP is the problem of assigning a color in a minimum set of colors to each arc such that two overlapping arcs have different colors. The CACP has been shown to be NP-complete [20](Chapter 10.3). In this section we will propose a heuristic algorithm for this problem. The proposed solution uses the transaction buffer at a router to transform a cyclic transaction set into an acyclic one such that the latter's schedule can be used to execute the former. More specifically, we select a router ϵ_k and split each transaction τ_i that goes through ϵ_k into two *pseudo transactions* τ_i' and τ_i''. τ_i' transfers data of τ_i from ϵ_i^1 to ϵ_k, and τ_i'' transfers the data which is stored in ϵ_k by τ_i' to ϵ_i^2. We said that τ_i' and τ_i'' is *feasibly transferred* data of τ_i if data of every job of τ_i is transferred to ϵ_i^2 before its deadline. The new transaction set is acyclic since there is no transaction going through ϵ_k. However, there is a precedence constraint between the pseudo transactions i.e. if τ_i'' transferred data in slot t then that data must be stored in ϵ_k by τ_i' before t. Since this constraint is not an assumption of transaction sets that can be scheduled by POGen, τ_i' and τ_i'' may not feasibly transfer data of τ_i. More specifically, note that POBase does not, in general, guarantee any order of slots used by two different transactions. As the result, the POBase procedure in POGen may generate a schedule where τ_i'' uses only slots that are before those of τ_i'. Therefore, some execution slots of τ_i'' (at least ones in the first interval) are wasted since there has been no data in ϵ_k to transfer. Figure 3.9 shows an example of this situation where in int^0 all the execution slots of τ_i'' are wasted (because τ_i' only starts executing at time 2). Note that, however, in the next interval τ_i'' can transfer what has been transfered by τ_i' in the previous interval and so on. This essentially means that if a period p_i has m scheduling intervals, we "waste" one out of m intervals both for τ_i' (which must complete transferring data before the last interval within the period) and for τ_i'' (which may only transfers data from the second interval within the period). To guarantee that both τ_i' and τ_i'' can effectively transfer all data in $m-1$ intervals instead of m, we have to inflate the utilization of the two transactions. This, in effect, means that the execution time of the two transactions within each period is higher than that of τ_i. The question now is: what is the minimum necessary increment in the execution time needed. We will address this question in Lemma 3.3.10 after we formally describe the problem in the next paragraphs. We also note that our solution only works well when p_i is relatively larger than the size of every interval (as shown in Inequality 3.3.13).

As described above, we replace each transaction $\tau_i \in \mathcal{T}$ where $\tau_i = (e_i, p_i, \epsilon_i^1, \epsilon_i^2)$ and τ_i goes through ϵ_k with two *pseudo transactions* (p-transactions) τ_i' and τ_i'' where $\tau_i' = (e_i^+, p_i, \epsilon_i^1, \epsilon_k)$, $\tau_i'' = (e_i^+, p_i, \epsilon_k, \epsilon_i^2)$, and $e_i^+ > e_i$. τ_i' and τ_i'' have the same utilization $u_i^+ = e_i^+/p_i > u_i$. τ_i is called the *original transaction* (o-transaction) of τ_i' and τ_i''. The new transaction set is called *pseudo transaction set* denoted by $\mathcal{T}'$. The following (work conserving) rule is applied to the schedules of a p-transaction.

Rule 3.3.1 *A p-transaction* always *transfers data of the current job of its o-transaction in slot* t *if there is data of the job stored in the source element of the p-transaction at time* t.

Note that the execution of a p-transaction in slot t can transfer data of the current job of its o-transaction only if the data has already been stored in its source elements before time t, otherwise the execution does nothing. In the former case, we say that the execution slot of the p-transaction is *loaded*, and is *empty* in the latter. Note that τ_i' execution slots are always loaded until it transfers all the data of the current job of τ_i. It is because when a job of τ_i' is ready, a job of τ_i is also ready and all data of the job of τ_i has been stored in ϵ_i^1. Therefore, the statement is true by Rule 3.3.1. Figure 3.9 shows an example of the schedule of τ_i' and τ_i'' with the given transaction parameters and with every scheduling interval having size 5. Consider int^0 in which τ_i' has 2 execution slots and τ_i'' has 3 execution slots (this number is determined by function $\mathsf{GenerateLoad}$). Since τ_i' is scheduled in slot 2 and 3 (this schedule is determined by POBase), there is no data of τ_i stored in ϵ_k before time $t = 3$. Therefore, the 3 execution slots of τ_i'' in int^0 are empty.

We say that a p-transaction is *effective in execution slot* t when either of the following cases occurs: 1) the execution slot is loaded or 2) the execution slot is empty and the p-transaction has transferred *all* data of the current job of its o-transaction at time t. Note that τ_i' is always effective in all slots because its execution slots are always loaded until it transfers all the data of the current job of τ_i. However that is not the case for τ_i''. In Figure 3.9, τ_i'' is not effective in slot 0, 1, 2, and 12. In these slots, there is still data of the current job of τ_i stored at ϵ_i^1 but there is no data of the job stored at ϵ_k. The following lemma is obvious due to Rule

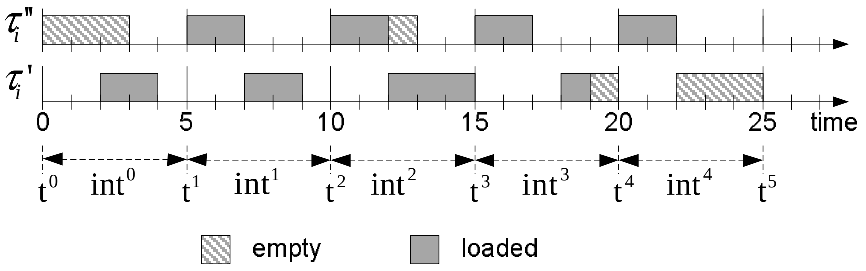


Figure 3.9: Schedule of p-transactions where $\forall k : \mathsf{int}^k = 5$, $\tau_i = (8, 25, \epsilon_i^1, \epsilon_i^2)$, and $\tau_i' = (12, 25, \epsilon_i^1, \epsilon_k)$, $\tau_i'' = (12, 25, \epsilon_k, \epsilon_i^2)$

.

3.3.1.

Lemma 3.3.9 *Consider job* j *of τ_i which is ready at t and has deadline at $t + p_i$, and consider scheduling interval* int^k *where $t \leq t^k < t^{k+1} \leq t + p$. If τ_i'' is not effective in one of its execution slot in* int^k*, then, at t^{k+1}, the amount of data of* j *that has been transferred by τ_i'' must be at least equivalent to $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$ slots.*

Proof.

By Rule 3.3.1, if τ_i'' is not effective in int^k, then by t^{k+1}, τ_i'' must have transferred all the data of j which had been transferred by τ_i' until t^k and there must be a portion of data of j still being stored at ϵ_i^1 after t^k. This also means that all execution slots of τ_i' between t and t^k are loaded. By POGen, the number of execution slots of τ_i' between t (i.e. when job j is ready) and t^k are $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$. Since τ_i' is always loaded in these slots, the amount of data of j that has been transferred by τ_i' until t^k is equivalent to at least $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$ slots. This completes the proof. □ The following lemma describes how to determine e_i^+

such that τ_i is schedulable and u_i^+ is minimized.

Lemma 3.3.10 *Every job of τ_i completes before its deadline if $\mathcal{T}'$ is schedulable by* POGen *and*

$$\forall \mathsf{int}^k : e_i^+ = \left\lfloor \frac{e_i + 1}{\alpha} \right\rfloor, \tag{3.3.13}$$

where $\alpha = \min_k (1 - |\mathsf{int}^k|/p_i)$.

Proof.

Since $\mathcal{T}'$ is schedulable by POGen, jobs of τ_i' and τ_i'' complete before their deadlines which are the same as the deadline of the correspondent job of τ_i. Thus, to complete the proof, we only need to show that the jobs of τ_i' and τ_i'' together transfer all the data of τ_i during their execution.

Consider job j of τ_i which is ready at t and has deadline at $t+p_i$. Let int^k where $t \le t^k < t^{k+1} \le t+p_i$ be the last scheduling interval where τ_i'' is not effective in at least one slot in int^k. We have:

- By POGen, the maximum number of execution slots of τ_i'' within $[t, t^{k+1})$ is $\lceil u_i^+ * t^{k+1} \rceil - u_i^+ * t$. Therefore, the smallest number of execution slots of τ_i'' within $[t^{k+1}, t+p_i)$ is $A = e_i^+ - \lceil u_i^+ * t^{k+1} \rceil + u_i^+ * t$.
- By Lemma 3.3.9, The amount of data of j that τ_i'' needs to transfer after t^{k+1} is equivalent to at most $B = e_i - \lfloor u_i^+ * t^k \rfloor + u_i^+ * t$ slots.

Since τ_i'' is always effective within $[t^{k+1}, t+p_i)$, all data of τ_i will be transferred to ϵ_i^2 before the deadline if

$$\begin{aligned} & A \ge B \Leftrightarrow \\ & e_i^+ - \lceil u_i^+ * t^{k+1} \rceil + u_i^+ * t \ge e_i - \lfloor u_i^+ * t^k \rfloor + u_i^+ * t \\ & \Leftrightarrow e_i^+ - e_i \ge \lceil u_i^+ * t^{k+1} \rceil - \lfloor u_i^+ * t^k \rfloor. \end{aligned} \tag{3.3.14}$$

Since

$$\begin{aligned} \lceil u_i^+ * t^{k+1} \rceil - \lfloor u_i^+ * t^k \rfloor &\le \lceil u_i^+ * t^{k+1} \rceil - \lceil u_i^+ * t^k \rceil + 1 \\ &\le \lceil u_i^+ * |\mathsf{int}^k| \rceil + 1, \end{aligned}$$

if the following inequality is true then Inequality 3.3.14 is also true:

$$e_i^+ - e_i \ge \lceil u_i^+ * |\mathsf{int}^k| \rceil + 1 \tag{3.3.15}$$

Since e_i^+ is integer, Inequality 3.3.15 is equivalent to

$$\lceil e_i^+ * \alpha \rceil \ge e_i + 1. \tag{3.3.16}$$

We complete the proof by showing in the following that if e_i^+ satisfies 3.3.13 then 3.3.16 is true:

$$\begin{aligned} & \lceil e_i^+ * \alpha \rceil = \left\lceil \left\lfloor \frac{e_i + 1}{\alpha} \right\rfloor * \alpha \right\rceil \ge \left\lceil \left(\frac{e_i + 1}{\alpha} - 1 \right) * \alpha \right\rceil \\ & = \left\lceil (e_i + 1) - \alpha \right\rceil = e_i + 1. \end{aligned}$$

The last equation in the above derivation comes from the fact that e_i is integer and $\alpha < 1$. □ Note that in

Equality 3.3.13, when $|int^k|$ approaches p_i, e_i^+ becomes larger and the proposed algorithm will be less likely to successfully schedule the transaction set. Despite this drawback, as we show through simulations (see Section 3.3.3), the proposed algorithm still performs significantly better than existing works on randomly-generated cyclic transaction sets.

The scheduling algorithm for cyclic transaction set $\mathcal{T}$, namely cPOGen, is summarized as follows.

Algorithm cPOGen:

- Step 1: Find ϵ_k such that the derived pseudo transaction set $\mathcal{T}'$, whose execution time of the p-transactions are calculated by Equation 3.3.13, passes the sufficient utilization bound test of POGen.
- Step 2: Schedule $\mathcal{T}'$ using POGen in which every scheduling interval has its length to be L, and schedule $\mathcal{T}$ accordingly following Rule 3.3.1.

Algorithm analysis: Note that we only need to execute Step 1 once and offline. This step can be implemented as: 1) visiting each router that has transactions going through (time complexity $O(B)$ where B is the number of routers) ; 2) at each visit, for each transaction τ_i going through the router, calculate e_i^+ (time complexity $O(N)$) and compare the new utilization of each PO-set with the bound (time complexity $O(N)$). Hence, time complexity of Step 1 is $O(N^2 * B)$. Step 2 is a POGen algorithm therefore it has the same time complexity as POGen.

3.3.3 Evaluation

Most of the previous related works [43, 44, 3, 24, 28] have focused on the Fixed-priority Scheduling Algorithm (FPA). These works deal with the methods for schedulability analysis and priority assignment. More specifically, Shi et al. have recently proposed in [43] a branch-and-bound algorithm that searches for a feasible priority set for a transaction set. If a feasible priority set exists, then the transactions set is guaranteed to be schedulable under the worst-case transaction latency (WTL) analysis proposed in [44]. The works in [44, 43] are the state of the art.

In this section, we are interested in comparing the performance of POGen/cPOGen on ring-topology NoC with the solution proposed in [44, 43]. The analyzed performance metric is the percentage of random transaction sets which are schedulable under POGen/cPOGen and FPA. Under POGen, an acyclic transaction set is schedulable if it passes the utilization bound test of Theorem 3.3.3. Meanwhile, under cPOGen,

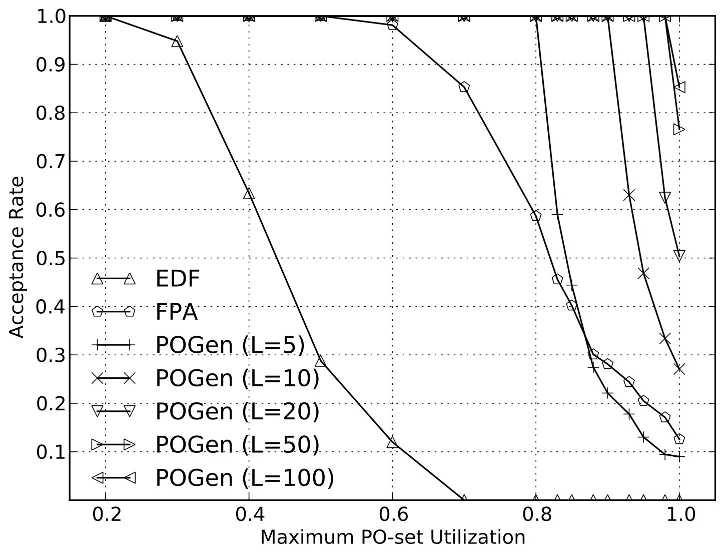


Figure 3.10: Acceptance rate of *acyclic* transaction sets with various maximum PO-set utilization

an cyclic transaction set is schedulable if its pseudo transaction set is schedulable. A transaction set is schedulable under FPA if it has a feasible priority set generated by the algorithm in [43].

In our experiments, we used three controlled parameters: 1) the maximum PO-set utilization of a transaction set[3]; 2) the size of transaction sets; and 3) the number of bus elements. The transactions' sources and destinations are randomly selected from the set of bus elements. Meanwhile, the transactions' utilization, transmission time and period are generated as follows. Given a maximum PO-set utilization u^{max}, the utilization of transaction τ_i is initially generated according to the uniform distribution algorithm in [5] such that the utilizations of all PO-sets are no larger than u^{max}. The transmission time e_i is generated as a uniformly-distributed random number in the range of 1 to 100 slots. The period p_i is then determined as $\lceil e_i/(u_i * \mathsf{L}) \rceil * \mathsf{L}$. Finally, given the pair of $\{e_i, p_i\}$, we recalculate u_i to be e_i/p_i.

The following graphs depict the average acceptance rate of POGen/cPOGen and FPA over 1000 different random transaction sets, in each of which two controlled variables are kept constant while the other one is varied. We report the acceptance rates of the algorithm on acyclic and cyclic transaction sets separately.

Figure 3.10 and 3.11 show the average acceptance rates of the algorithms on acyclic and cyclic transaction sets, respectively, (when L is 5, 10, 20, 50, or 100) under various maximum PO-set utilization. In these experiments, the size of transaction sets and the number of bus elements are set at 20 and 10,

[3]it is equivalent to "the maximum link utilization" in [43]

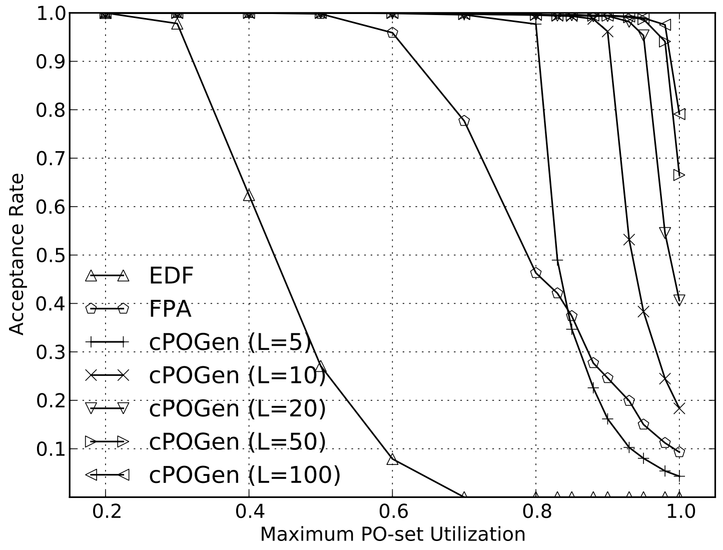


Figure 3.11: Acceptance rate of *cyclic* transaction sets with various maximum PO-set utilization

respectively. For comparison purpose, we also include in these figures the acceptance rates of the Earliest-Deadline-First (EDF); it is a well-known scheduling algorithm that can be used when the parallel execution of non-overlapping transactions is not allowed. It can be seen that in most cases the acceptance rate of POGen/cPOGen is better than that of FPA especially when PO-set maximum utilization is high and $\mathsf{L} > 5$. The better performance of POGen/cPOGen comes from the fact that the WTL analysis in [44] does not always take advantage of the parallelism between non-overlapping transactions. For example, consider the transaction set shown in Figure 3.2. Assume τ_3 and τ_5 have higher priority than τ_4. According to the WTL analysis in [44], the interference of transactions τ_3 and τ_5 on the execution of τ_4 is calculated as if all transactions were using a single-shared resource. However, POGen/cPOGen allows τ_3 and τ_5 to be executed in parallel as shown in Figure 3.5. In other words, the acceptance rate of FPA will be reduced when the number of PO-sets that contain a same transaction increases. Let denote the number of PO-sets that contain a same transaction as PcT. We also notice from Figure 3.10 and 3.11 that the acceptance rates of cyclic transaction sets is lower than that of acyclic ones. This is true both for our proposed algorithms and for FPA: with the former, the reduction is because of the utilization inflation we used to transform the transactions; with the latter, the reduction is because of the higher PcT when transaction sets are cyclic. In average, the maximum reduction of both POGen/cPOGen and FPA is about 10% and it is higher when the maximum utilization is higher. The figures also show (in the difference in performance of EDF and POGen/cPOGen) that allowing

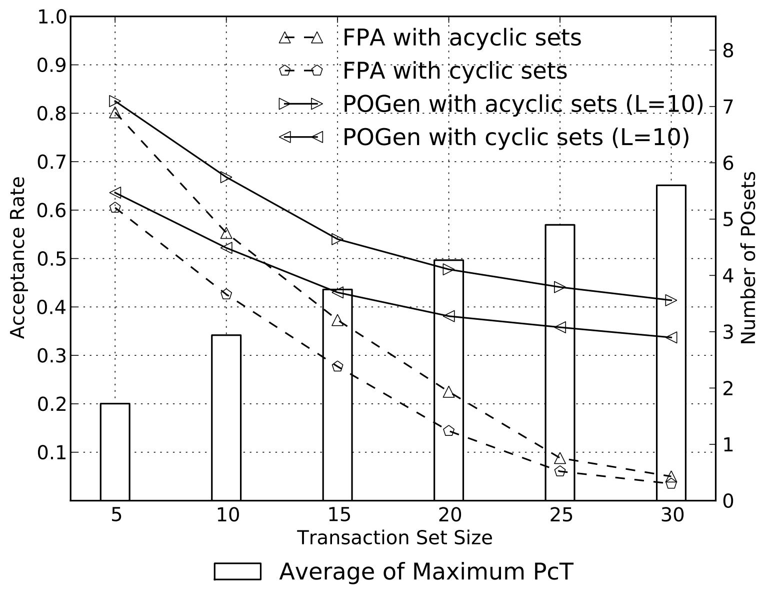


Figure 3.12: Acceptance rate with various size of transaction sets (maximum PO-set utilization is 0.95)

parallelism between transactions dramatically increases the bus utilization.

Figure 3.12 shows the acceptance rate with the maximum PO-set utilization to be 0.95, the number of bus elements to be 10, and various size of transaction sets. The results of acyclic and cyclic transaction sets are shown in the solid and dashed lines, respectively. We also draw in Figure 3.12 a bar graph which shows the average (over all transaction sets) of the maximum PcT in each set. The performance of POGen/cPOGen is better than FPA especially when the size of transaction sets are higher. The reason is that, when the number of bus elements is fixed, the higher the size of a transaction set, the bigger the maximum PcT (as shown in the bar graph). As a consequence, FPA suffers more from the effect described in the previous paragraph. The performance of POGen/cPOGen also reduces when the transaction number is higher because there are more transaction sets that do not meet the utilization bound. The reductions, however, are less than that of FPA.

The same reason explains the better performance of POGen/cPOGen in Figure 3.13 which shows the acceptance rate of POGen/cPOGen (with $\mathsf{L} = 10$) and FPA when the number of bus elements is varied. In these experiments, the maximum PO-set utilization is set at 0.95, the size of transaction sets are fixed at 20. When the number of bus elements increases, there will be more longer transactions which may belong to higher number of distinct PO-sets as shown with the bar graph.

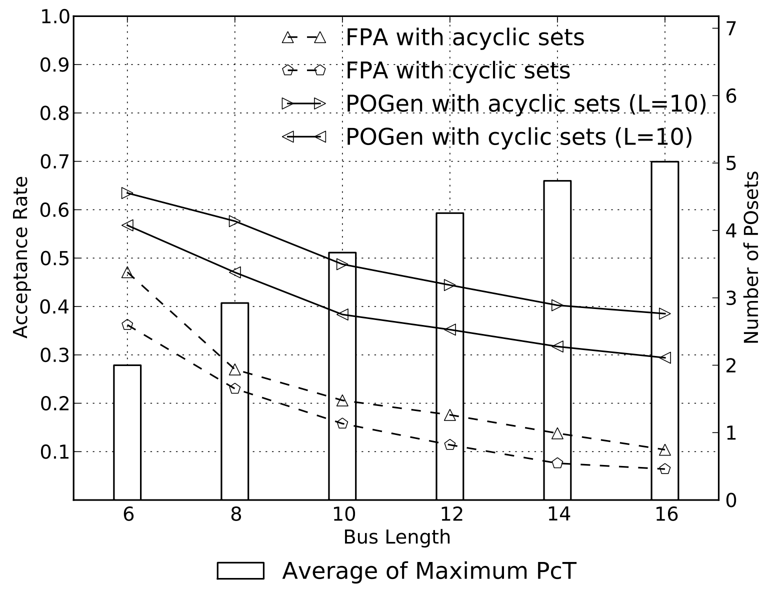


Figure 3.13: Acceptance rate with various bus length (maximum PO-set utilization is 0.95)

3.3.4 Implementation

The main drawback of POGen is that its overhead, in general, is higher than that of fixed-priority ones. This overhead heavily depends on specific implementations and platforms. To demonstrate the applicability of POGen, in this section, we discuss an implementation of POGen on a Cell Broadband Engine processor [7] and our measurement of its execution time overhead.

The Cell processor has 1 PowerPC Processing Element (PPE) and 8 Synergy Processing Elements (SPE) each of which is an element on the Cell ring bus. There are also 3 additional bus elements which are a memory controller and two I/O controllers. POGen is implemented to run on SPEs as an online algorithm. It is invoked at the beginning of each scheduling interval by a timer-interrupt handler and generates the schedule of all transactions in that interval. Then, if the generated schedule has $S(\tau_i, t) = 1$, a slot scheduler will transfer data of τ_i in slot t using Direct-Memory-Access commands. The slot scheduler is also invoked by a timer-interrupt handler. Note that since POGen is only executed once at the arrival time of each transactions, the number of interrupts that invokes POGen is the same as that of the fixed-priority schedulers. Note that the execution time of the former, however, may be larger that of the latter. In our implementation (shown later), the execution time of POGen is typically no bigger than 3% in utilization. We also note that the execution time of the slot scheduler must also be accounted as the overall overhead of POGen. Since the slot scheduler is simply a table-lookup function, we expect that this scheduler can be implemented in hardware as part of the router. If so, its overhead will be negligible and will not affect the processing elements. Our software implementation of the slot scheduler shows that it adds about 1% to the total utilization. For comparison, the number of interrupts incurred by POGen is equivalent to that of the Boundary-Fair algorithm [52] (which is significantly smaller than that of PFair [4]).

POGen execution time was measured under various slot sizes which are $10us$, $20us$, $50us$, and $100us$. We assume that the period of every transaction is a multiple of $1ms$ which is also the smallest possible scheduling interval. We also selected the size of every scheduling interval to be equal to the GCD of all periods, which happened to be $1ms$ in all generated transaction sets. Given the different values of slot size and the GCD of all periods, the size of L measured in number of slots are 100, 50, 20 and 10 slots. We generated transaction sets with various sizes using the same methodology discussed in Section 3.3.3. The sizes of transaction sets are 10, 20, 30, and 40.

Figure 3.14 shows the execution time of POGen in ms under various conditions. This execution time also includes the latency of the timer-interrupt handler that invokes POGen. It can be seen that the algorithm overhead increases when L has a higher value. However the algorithm overhead is no more than $0.03ms$ even when $\mathsf{L} = 100$ slots. Since each scheduling interval is $1ms$, under the given conditions, the maximum algorithm overhead is less than 3% of the scheduling interval size.

Our measurement also shows that the execution time of the slot scheduler is no more than $0.125us$. In other words, if the slot size is $10us$, the overhead is less than 1.25% of the slot size. The overhead is smaller when the slot size is bigger.

3.4 Real-time Scheduling for General NoC

In Section 3.3, we introduce two scheduling algorithms for real-time transactions on a ring-topology NoC: Algorithm POBase is designed to schedule transactions whose periods are the same, whereas Algorithm POGen is designed for general transaction sets. The latter is built upon two algorithms POBase and GenerateLoad. Although the general framework of POGen can be used for transaction sets on general NoC, Algorithm POBase and GeneratedLoad can not be reused. The reason is that these two algorithms rely on the fact that for each transaction set on a ring-topology NoC, there exists a *total order* which is defined solely based on transactions' endpoints. This total order does not exist in a general NoC. Therefore, a different transaction order (more general) must be defined. Like with ring-topology NoC, we also classify transaction sets on general NoC into acyclic and cyclic transaction sets. This classification is a generaliza-

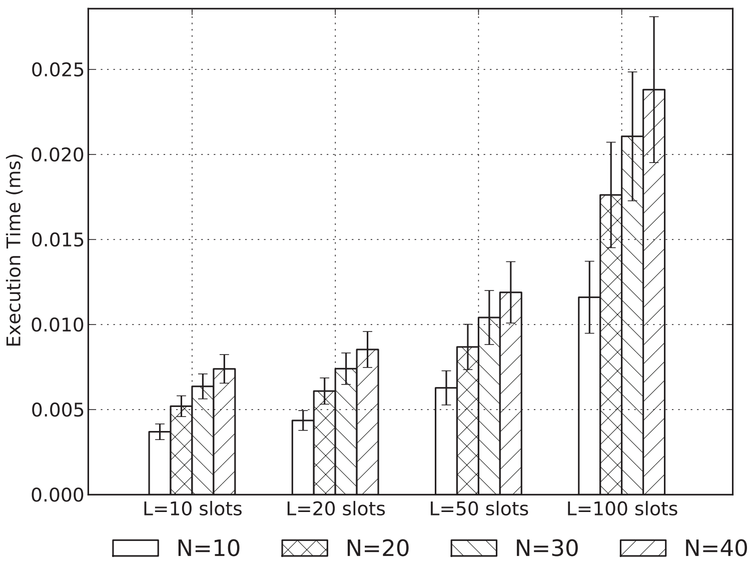


Figure 3.14: Average execution time of POGen

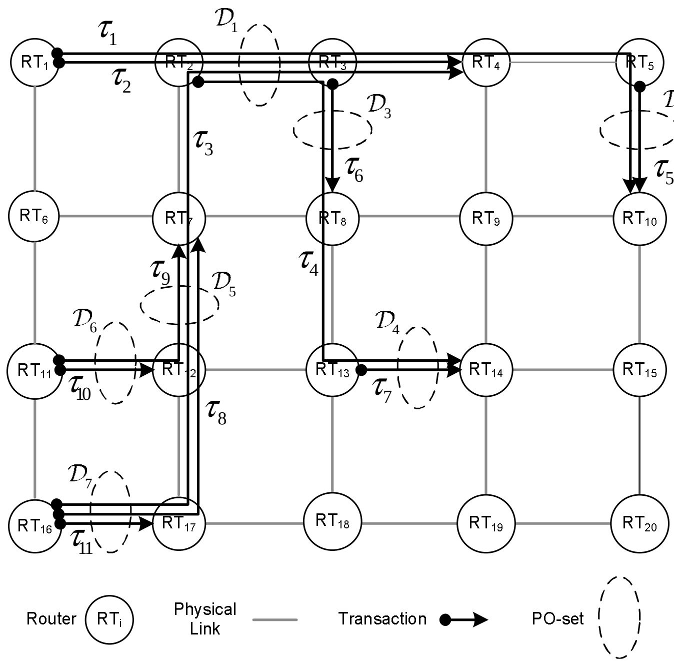


Figure 3.15: A transaction set in NoC

tion of the classification introduced in Section 3.3. In this research, we will focus only on the scheduling problem of *acyclic transaction sets* on general NoC. As shown later, transaction sets of this type often appear in practical applications.

3.4.1 Transaction Model

In this section, we introduce the transaction model used in general NoC. This model is a generalization of the model introduced in Section 3.3. We define $\mathcal{T}$ as the set of all data transactions: $\mathcal{T} = \{\tau_i : i = [1, N]\}$. A data transaction τ_i is characterized by a tuple $\tau_i = (e_i, p_i, \mathcal{R}_i)$ where e_i is the time required to transfer each job of τ_i; p_i is the period of τ_i; and $\mathcal{R}_i$ is the fixed route of τ_i through the network, that is the set of links that the transaction traverses. As an example, the route of transaction τ_4 in Figure 3.15 is $\mathcal{R}_4 = \{\mathrm{RT}_2 \rightarrow \mathrm{RT}_3 \rightarrow \mathrm{RT}_8 \rightarrow \mathrm{RT}_{13} \rightarrow \mathrm{RT}_{14}\}$, where $\rightarrow$ represents a physical link. Each job of τ_i must complete within its period, i.e. relative deadlines are equal to periods. The network utilization u_i of τ_i is calculated as: $u_i = e_i/p_i$. We assume that all data transactions arrive at time 0. Let hyper-period h of $\mathcal{T}$ be the least common multiple of the periods of all transactions in $\mathcal{T}$. Finally, two transactions τ_i and τ_j are said to *overlap* if their routes have any link in common, that is $\mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset$. Given a data transaction set $\mathcal{T}$, we define an overlap indicating function $OV : \mathcal{T} \times \mathcal{T} \mapsto \{0, 1\}$ where $OV(\tau_i, \tau_j) = 1$ if τ_i and τ_j overlap, and 0 otherwise.

PO-sets and incident tree: A *pairwise overlap set* (PO-set) $\mathcal{D}_j$ is defined as a maximal set of overlapping transactions, i.e. a subset of $\mathcal{T}$ that satisfies the following two conditions: 1) $\forall \tau_i, \tau_j \in \mathcal{D} : OV(\tau_i, \tau_j) = 1$; and 2) if $\tau_k \notin \mathcal{D}$ then $\exists \tau_i \in \mathcal{D} : OV(\tau_i, \tau_k) = 0$. By definition, every two transactions that overlap each other must belong to at least a same PO-set. For convenience, we consider that a non-overlapping transaction belongs to a PO-set that contains only that transaction. Let $N^{\mathcal{D}}$ be the total number of PO-sets of $\mathcal{T}$. In general a transaction may belong to more than one PO-set. As an example, the transaction set in Figure 3.15 comprises seven PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_1, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_6\}$, $\mathcal{D}_4 = \{\tau_4, \tau_7\}$, $\mathcal{D}_5 = \{\tau_3, \tau_8, \tau_9\}$, $\mathcal{D}_6 = \{\tau_9, \tau_{10}\}$, $\mathcal{D}_7 = \{\tau_3, \tau_8, \tau_{11}\}$. Two PO-sets $\mathcal{D}_i, \mathcal{D}_j$ are said to be *connected* if $\mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset$.

The *PO-graph* of $\mathcal{T}$ is defined as the incident graph of PO-sets of $\mathcal{T}$, i.e. the graph whose vertexes represent PO-sets and there is an edge between two vertexes if the two correspondent PO-sets are connected. Thereafter we shall use a PO-set and its correspondent vertex in PO-graph interchangeably. We assume that a PO-graph is a connected graph. The reason is that if the graph is disconnected, its components do not

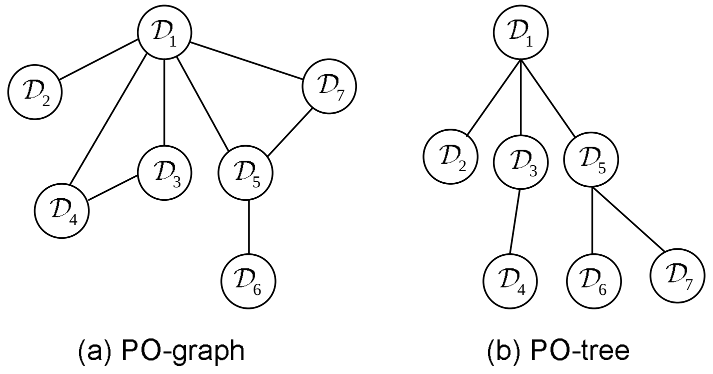


Figure 3.16: PO-graph, PO-tree of $\mathcal{T}$ shown in Figure 3.15

interfere with each other in terms of scheduling and therefore we can deal with them as separate transaction sets. Figure 3.16(a) shows the PO-graph of the transaction set showed in Figure 3.15.

Let *PO-tree* denote a spanning tree of the PO-graph and let PO-set $\mathcal{D}^r$ be the root of the PO-tree. Figure 3.16(b) shows a PO-tree rooted at $\mathcal{D}_1$ of the PO-graph in Figure 3.16(a). With respect to a specific PO-tree, PO-set $\mathcal{D}_i$ is an *ancestor* of PO-set $\mathcal{D}_j$, denoted by $\mathcal{D}_i \prec \mathcal{D}_j$, if $\mathcal{D}_i$ is on the path from the root $\mathcal{D}^r$ to $\mathcal{D}_j$ on the PO-tree. Let $\mathsf{ancestors}(\mathcal{D}_j)$ be the set of all ancestors of $\mathcal{D}_j$. For convenience, we use notation $\mathcal{D}_i \preceq \mathcal{D}_j$ to indicate that $\mathcal{D}_i \in \mathsf{ancestors}(\mathcal{D}_j) \cup \mathcal{D}_j$. Furthermore, $\mathcal{D}_i$ is the *parent* of $\mathcal{D}_j$, denoted by $\mathcal{D}_i = \mathsf{parent}(\mathcal{D}_j)$, if and only if $\mathcal{D}_i$ is the immediate ancestor of $\mathcal{D}_j$. Note that in a tree, a non-root node has a unique parent. Let $\mathsf{children}(\mathcal{D}_i)$ be the set of all PO-sets whose parent is $\mathcal{D}_i$. As an example, in Figure 3.16(b), $\mathsf{ancestors}(\mathcal{D}_6) = \{\mathcal{D}_1,\mathcal{D}_5\}$, $\mathcal{D}_5 = \mathsf{parent}(\mathcal{D}_6)$, and $\mathsf{children}(\mathcal{D}_5) = \{\mathcal{D}_6, \mathcal{D}_7\}$. We define the height of tree node $\mathcal{D}_i$, denoted by $\mathsf{height}(\mathcal{D}_i)$, as follows: if $\mathsf{children}(\mathcal{D}_i) = \emptyset$ then $\mathsf{height}(\mathcal{D}_i) = 0$, otherwise $\mathsf{height}(\mathcal{D}_i) = 1 + \max_{\mathcal{D}_j \in \mathsf{children}(\mathcal{D}_i)} \mathsf{height}(\mathcal{D}_j)$.

In Section 3.3, we proved that there exists a category of transaction sets in uni-dimensional buses, called *cyclic* transaction sets, for which the scheduling problem is NP-complete. Since uni-dimensional buses are a special case of NoC, the same result applies in our case. In this paper, we extend the definition of acyclic/cyclic transaction sets in Section 3.3 to encompass transaction sets on multi-dimensional NoC. We say that a NoC transaction set is acyclic if it has a PO-tree that satisfies Property 3.4.1 and 3.4.2, and it is cyclic otherwise.

Property 3.4.1 *If $\mathcal{D}_l \preceq \mathcal{D}_m \preceq \mathcal{D}_n$ and $\tau_i \in \mathcal{D}_l$ and $\tau_i \in \mathcal{D}_n$, then $\tau_i \in \mathcal{D}_m$.*

Property 3.4.2 *If $\mathcal{D}_l \npreceq \mathcal{D}_m$ and $\mathcal{D}_m \npreceq \mathcal{D}_l$, then $\mathcal{D}_l \cap \mathcal{D}_m = \emptyset$.*

Note that a transaction set that is cyclic or acyclic by the definition in Section 3.3 is also cyclic or acyclic, respectively, by the current definition. In this research, we will focus only on acyclic transaction sets because of two reasons: 1) as we discuss later, many real-time applications exhibit data-flow topologies that are acyclic; 2) for this instance of the problem, we can derive an efficient and near-optimum solution in practical settings. We believe that having a good solution for this specific problem will provide a good theoretical foundation for solving the general NP-complete problem. Thereafter, we use PO-tree to denote the spanning tree that satisfies the aforementioned properties. The transaction set shown in Figure 3.15 is acyclic because given its PO-tree shown in Figure 3.16(b), all transactions and PO-sets satisfy Property 3.4.1 and 3.4.2. For example, since $\mathcal{D}_1 \prec \mathcal{D}_3 \prec \mathcal{D}_4$, any transaction that is in both $\mathcal{D}_1$ and $\mathcal{D}_4$ (e.g. τ_4) is also in $\mathcal{D}_3$ (Property 3.4.1), or since $\mathcal{D}_2 \npreceq \mathcal{D}_3$ and $\mathcal{D}_3 \npreceq \mathcal{D}_2$, $\mathcal{D}_2 \cap \mathcal{D}_3 = \emptyset$ (Property 3.4.2).

For ease of presentation, we define the following index scheme for the PO-sets in the PO-tree: select any Depth-First-Search (DFS) travel over PO-tree. Then each PO-set is indexed by a unique number from 1 to $N^{\mathcal{D}}$ in the order it is visited in the selected DFS. PO-sets in Figure 3.16(b) are indexed following this definition. Note that, with this index scheme, if $\mathcal{D}_l \preceq \mathcal{D}_m$ then $l \leq m$ but $l \leq m$ does not always imply $\mathcal{D}_l \preceq \mathcal{D}_m$ (see $\mathcal{D}_2$ and $\mathcal{D}_3$ in Figure 3.16). The following property, however, is true.

Property 3.4.3 *If $l \leq m \leq n$ and $\mathcal{D}_l \preceq \mathcal{D}_n$, then $\mathcal{D}_l \preceq \mathcal{D}_m$.*

Proof.

Assume by contradiction that $\mathcal{D}_l \npreceq \mathcal{D}_m$. If $\mathcal{D}_m \prec \mathcal{D}_l$, then $m < l$. If otherwise $\mathcal{D}_m \nprec \mathcal{D}_l$, then either $m < l$ or $m > n$. Both cases contradict the property's assumption. □

Given an indexed PO-sets, let maxid_i and minid_i denote the indexes with maximum and minimum value, respectively, among all PO-sets to which τ_i belongs. Also denote these PO-sets as maxPO_i and minPO_i, respectively. As an example, $\mathsf{maxid}_3 = 7$, $\mathsf{minid}_3 = 1$, $\mathsf{maxPO}_3 = \mathcal{D}_7$, $\mathsf{minPO}_3 = \mathcal{D}_1$. We have the following property.

Property 3.4.4 *For every $\mathcal{D}_l$ which contains τ_i, $\mathsf{minPO}_i \preceq \mathcal{D}_l$.*

Proof.

Assume by contradiction that there exists $\mathcal{D}_l$ where $\tau_i \in \mathcal{D}_l$ and $\mathsf{minPO}_i \npreceq \mathcal{D}_l$. Since, by definition, $\mathsf{minid}_i < l$, we have $\mathcal{D}_l \npreceq \mathsf{minPO}_i$. Then, by Property 3.4.2, $\mathcal{D}_l$ cannot share τ_i with minPO_i, which contradicts the property's assumption. □

Finally, we have the following lemma which will be used for calculating the proposed algorithms complexity.

Lemma 3.4.1 *If $\mathcal{T}$ is acyclic, then its number of PO-sets $N^{\mathcal{D}} \leq N$.*

Proof.

By the defined PO-set index scheme and by Property 3.4.1 and 3.4.2, we have transactions in $\mathcal{D}_j \setminus \mathsf{parent}(\mathcal{D}_j)$ do not belong to $\bigcup_{i \in [1,j-1]} \mathcal{D}_i$. Therefore sets in collection $A = \{\mathcal{D}_j \setminus \mathsf{parent}(\mathcal{D}_j) : j \in [1, N^{\mathcal{D}}]\}$ are mutually disjoint. Since the size of $\mathcal{T}$ is N, by pigeonhole principle, we have $|A| = N^{\mathcal{D}} \leq N$. $\square$

Schedulability Necessary Condition

Since by definition, no two transactions of a PO-set $\mathcal{D}_l$ can be scheduled concurrently, all transactions of $\mathcal{D}_l$ must be scheduled in sequence. In other words, the transactions of $\mathcal{D}_l$ can be considered to be sharing one resource. This results in a necessary condition on the schedulability of a transaction set shown in Theorem 3.4.1

Theorem 3.4.1 *A transaction set $\mathcal{T}$ is schedulable only if:*

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} = \sum_{\forall \tau_i \in \mathcal{D}_j} u_i \leq 1. \tag{3.4.1}$$

Motivating Applications for Acyclic Transaction Sets

Many real-time applications are in the form of data-flow processing applications [27, 47] where data may be processed through multiple consecutive stages. An example is the multipurpose status display application on an avionic system [27] which shows the status of all aircraft avionics devices. A task periodically gets data from I/O devices such as radars, then processes the data before sending information to a display task which is in charge of displaying useful information to the pilots. Another example is an application that processes transmission flows in 3G base stations [47]. In the down-link direction, the input flows are the data flow and the control flow. These flows are processed separately through several stages then merged back into one flow. The merged flow then goes through several additional processing stages before being sent to the output port. A popular programming model for this type of applications in SoC is the streaming model [40, 47] in which each processing stage is executed in one processing element. processing elements are in either a serial or parallel pipeline. Data is transfered between processing stages through the NoC.

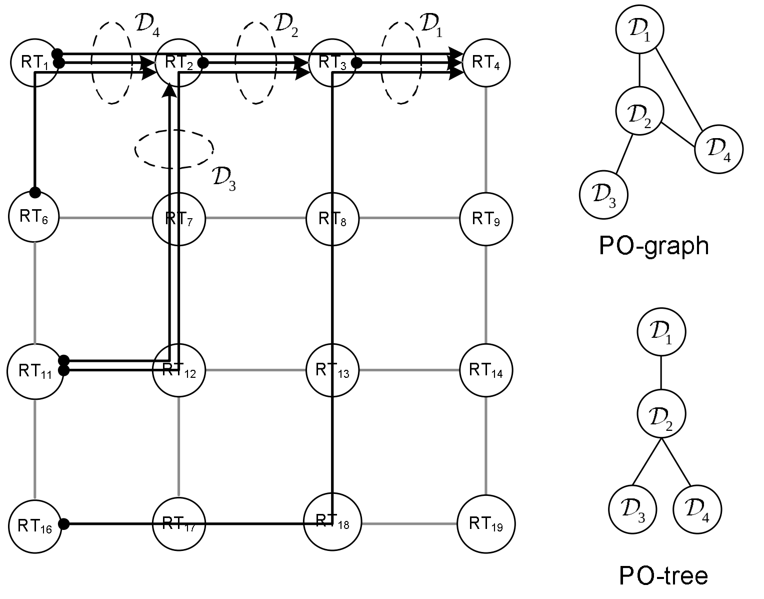


Figure 3.17: A data-stream processing application

If the streaming programming model is used, it is usually easy to come up with an allocation of stages to processing elements that result in acyclic transactions.

Figure 3.17 shows an acyclic transaction set created by a data-flow processing applications: input flows are processed in parallel by processing elements at $\{\mathrm{RT}_1, \mathrm{RT}_6\}$, $\{\mathrm{RT}_{11}, \mathrm{RT}_{16}\}$, then merged at RT_2 and processed by the processing element at this router; the merged flow continues going through two additional processing stages at RT_3 and RT_4. The induced PO-graph has the following PO-set nodes: $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, $\mathcal{D}_4$. In Figure 3.17, we represent each PO-set with a dashed circle where transactions that cut through a circle are transactions in the correspondent PO-set of the circle. The PO-graph and a PO-tree (rooted at $\mathcal{D}_1$) of this transaction set is also shown in Figure 3.17. The transaction set is acyclic because it satisfies Property 3.4.1 and 3.4.2. For example: the transaction which is in both $\mathcal{D}_1$ and $\mathcal{D}_4$ is also in $\mathcal{D}_2$ (Property 3.4.1); and $\mathcal{D}_3 \cap \mathcal{D}_4 = \emptyset$ (Property 3.4.2).

3.4.2 Real-time Scheduling for Acyclic Transaction Sets on General NoC

As aforementioned, we use a same scheduling framework POGen proposed in Section 3.3 for the problem at hand. However, the component algorithms of POGen which are POBase and GenerateLoad need to be designed anew to accommodate the generalized transaction model. Also note that, unlike the algorithms in Section 3.3, the new proposed algorithms are polynomial.

We proved in Section 3.3 that the schedule generated by executing POGen iteratively over all scheduling intervals is feasible if the following two conditions are true.

Condition 3.4.1 POBase *is optimal for a transaction set whose transactions' periods are the same, meaning that the schedule generated by* POBase *is feasible if the transaction set satisfies the necessary condition in Theorem 3.4.1.*

Condition 3.4.2 GenerateLoad *generates a feasible load set* $\{l_i^k : \forall \tau_i \in \mathcal{T}\}$ *for interval* int^k *if its inputs satisfy the following Inequalities:*

$$\forall \mathcal{D}_j \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor \leq |\mathsf{int}^k| \tag{3.4.2}$$

$$\sum_{x \in [0, t^k)} S(\tau_i, x) \geq \lfloor u_i * t^k \rfloor \tag{3.4.3}$$

In the following sections, we will propose algorithm POBase and GenerateLoad for acyclic transaction sets on NoC that satisfy Conditions 3.4.1 and 3.4.2. To differentiate with the algorithms in Section 3.3, we name the new algorithms POBaseNoC and GenerateLoadNoC. The proposed POBaseNoC satisfies Condition 3.4.1 because it feasibly schedules any same-period acyclic NoC transaction set that satisfies the necessary condition of Theorem 3.4.1. Furthermore, the proposed GenerateLoadNoC satisfies Condition 3.4.2 if all PO-sets satisfy the following utilization bound:

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}, \tag{3.4.4}$$

where L is defined as the greatest common divisor (GCD) of all transaction periods and is measured in terms of the basic slot size. Although the utilization bound is sufficient, it approximates 1 when L is large. The value of L in a real system is a function of both the slot size and transaction periods. As we discuss in more details in Section 3.5, we believe that in many modern NoCs L has practical values ranging from 10 to 100 slots. Since the proposed GenerateLoadNoC imposes a stricter requirement on the transaction set than that of POBaseNoC, Inequality 3.4.4 becomes a sufficient utilization bound of POGen.

The POBaseNoC **algorithm**

Algorithm 8 POBaseNoC

Input: $\mathcal{T}$ whose all transactions have same period p

```
1:  $\mathcal{L} \leftarrow$ list of $\forall \tau_i \in \mathcal{T}$ in ascending order of $\mathsf{minid}_i$
2:  add $\mathsf{d} = \big\{0, p, \{\text{null}\}\big\}$ to Dur
3:  for each $\tau_i \in \mathcal{L}$ do
4:      $r \leftarrow e_i$
5:      for each $\mathsf{d} \in$ Dur do
6:          $\mathcal{M} \leftarrow \{\tau_l \in \mathsf{d}.tlist : \mathsf{maxid}_l \geq \mathsf{minid}_i\}$
7:          $\tau_j \leftarrow \arg\min_{\tau_l \in \mathcal{M}} (\mathsf{maxid}_l)$
8:          //check if duration d is valid for $\tau_i$
9:          if $\tau_j$ =None or ($\tau_j \neq$None and $OV(\tau_j, \tau_i) = 0$) then
10:             if $r \geq \mathsf{d}.t_2 - \mathsf{d}.t_1$ then
11:                 $\mathsf{d}.tlist \leftarrow \mathsf{d}.tlist \cup \{\tau_i\}$
12:                 add $\{\mathsf{d}.t_1, \mathsf{d}.t_2\}$ to $\mathsf{Sched}(\tau_i)$
13:                 $r \leftarrow r - (\mathsf{d}.t_2 - \mathsf{d}.t_1)$
14:             else
15:                 replace $\mathsf{d} \in$ Dur with $\mathsf{d}_1 = \{\mathsf{d}.t_1, \mathsf{d}.t_1 + r, \mathsf{d}.tlist \cup \{\tau_i\}\}$ and $\mathsf{d}_2 = \{\mathsf{d}.t_1 + r, \mathsf{d}.t_2, \mathsf{d}.tlist\}$
16:                 add $\{\mathsf{d}_1.t_1, \mathsf{d}_1.t_2\}$ to $\mathsf{Sched}(\tau_i)$
17:                 $r \leftarrow 0$
18:                 break
```

The goal of POBaseNoC (Algorithm 8) is to find a feasible schedule for a transaction set which satisfies necessary condition in Theorem 3.4.1 and has all transactions with a same period p. The proposed POBaseNoC is executed as follows: each transaction, in ascending order of its minid, is assigned to the earliest *valid* slots (transactions with the same minid are ordered by their indexes). Slot t is valid for transaction τ_i if t has not been assigned to transactions overlapping with τ_i. In the next paragraphs, we will discuss an efficient implementation of POBaseNoC and prove its correctness and optimality.

In POBaseNoC, in order to find valid slots for a transaction, we scan through a list of durations (Line 5) (instead of scanning through all slots as in Algorithm 6). Each duration represents a set of consecutive slots to which some non-overlapping transactions have been assigned. A transaction τ_i can only be assigned to slots in duration d if transactions that have been assigned to d do not overlap with τ_i. We call these durations the *valid* durations of τ_i.

In POBaseNoC, Dur stores the list of durations. Each duration d is represented as a tuple $\{\mathsf{d}.t_1, \mathsf{d}.t_2, \mathsf{d}.tlist\}$ where all and only slots in $[\mathsf{d}.t_1, \mathsf{d}.t_2)$ are slots of d. The size of d is the number of slots of d. $\mathsf{d}.tlist$ is the list of transactions that have been assigned to d. Transaction τ_i is said to be *assigned* to d if and only if τ_i is assigned to all slots in d. The algorithm guarantees that a transaction is assigned to all slots of a duration or none of them. The first duration appended to Dur is $\{0, p, \{\text{null}\}\}$ where $\{\text{null}\}$ denotes an empty list. A duration will be updated or split into two durations after an assignment of a transaction completes. Sched(τ_i) stores the list of pairs $\{t_1, t_2\}$ where all slots in $[t_1, t_2)$ are assigned to τ_i, or equivalently $\forall t \in [t_1, t_2) : S(\tau_i, t) = 1$. Consider when transaction τ_i is being assigned. The value of variable r will be the remaining amount of transmission time of τ_i to be assigned. The algorithm checks if duration d is valid for τ_i using Line 6 to 9 (we will describe these steps in next paragraphs). If d is valid then: 1) if the remaining transmission time r of τ_i is larger than the size of d, τ_i is assigned to all slots of d (Line 12). In this case, d will be updated by adding τ_i to $\mathsf{d}.tlist$ (Line 11); 2) otherwise, d is split into two durations which are $\mathsf{d}_1 = \{\mathsf{d}.t_1, \mathsf{d}.t_1 + r, \mathsf{d}.tlist \cup \{\tau_i\}\}$ and $\mathsf{d}_2 = \{\mathsf{d}.t_1 + r, \mathsf{d}.t_2, \mathsf{d}.tlist\}$ (Line 15), and τ_i is assigned to d_1 (Line 16).

Lines 6 to 9 are used to check if d is valid for τ_i (i.e. if there is no transaction in $\mathsf{d}.tlist$ overlapping τ_i). To do this, it suffices to check τ_i with only one transaction in $\mathsf{d}.tlist$ which is $\tau_j = \arg\min_{\tau_l \in \mathcal{M}} (\mathsf{maxid}_l)$ where $\mathcal{M} = \{\tau_l \in \mathsf{d}.tlist : \mathsf{maxid}_l \geq \mathsf{minid}_i\}$ (see Lemma 3.4.3). This implementation results in a more time-efficient algorithm as will be shown later.

Figure 3.18 shows a schedule generated by POBaseNoC for the transaction set shown in Figure 3.1 where the DFS travel is $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_6, \mathcal{D}_7$ and the tuple (e_i, p_i) of each transaction is given as follows: $\tau_1 : (2, 8)$; $\tau_2 : (2, 8)$; $\tau_3 : (3, 8)$; $\tau_4 : (1, 8)$; $\tau_5 : (6, 8)$; $\tau_6 : (7, 8)$; $\tau_7 : (7, 8)$; $\tau_8 : (1, 8)$; $\tau_9 : (4, 8)$; $\tau_{10} : (4, 8)$; $\tau_{11} : (4, 8)$. Given the transactions' parameters, all PO-sets have utilization 100%. The schedule of each PO-set is depicted by a set of continuous rectangles on the horizontal time line. Each rectangle represents an execution of a transaction. A valid transaction ordering in Line 3 is $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}, \tau_{11}$ (i.e. ascending order of transactions' minid). Consider when τ_3 is being allocated. At this point since τ_1 and τ_2 have been allocated, Dur has three durations $\mathsf{d}_1 = \{0, 2, \{\tau_1\}\}$, $\mathsf{d}_2 = \{2, 4, \{\tau_2\}\}$, $\mathsf{d}_3 = \{4, 8, \{null\}\}$. Since only d_3 is valid, POBaseNoC assigns τ_3 to slots in $[4, 7)$ and

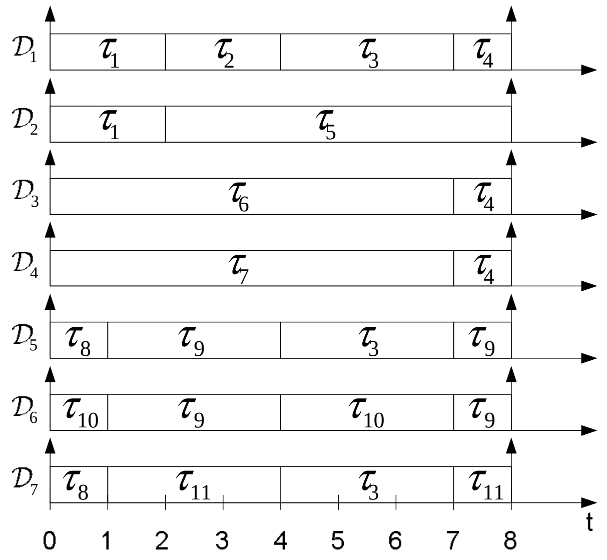


Figure 3.18: Schedule generated by POBaseNoC

splits d_3 into two durations which are $\{4, 7, \{\tau_3\}\}$ and $\{7, 8, \{null\}\}$.

The correctness of POBaseNOC is proved in Lemma 3.4.3 and 3.4.4 which show that Line 6 to 9 guarantees that every transaction is assigned to only valid durations which contains only valid slots. The proof of these lemmas requires Lemma 3.4.2. The optimality of the algorithm is, then, shown in Theorem 3.4.2.

Lemma 3.4.2 *Consider transaction τ_i and τ_j. If transaction τ_j has been allocated before τ_i and $OV(\tau_i, \tau_j) = 1$, then $\tau_j \in \mathsf{minPO}_i$.*

Proof.

Since $OV(\tau_i, \tau_j) = 1$, there exists $\mathcal{D}_l$ where $\tau_i, \tau_j \in \mathcal{D}_l$. Assume by contradiction that $\tau_j \notin \mathsf{minPO}_i$, i.e. $\mathcal{D}_l \neq \mathsf{minPO}_i$. Since τ_j has been allocated before τ_i, by the algorithm operation and the definition of minid_i, we have $\mathsf{minid}_j \leq \mathsf{minid}_i < l$. By Property 3.4.3 and 3.4.4, we have $\mathsf{minPO}_j \preceq \mathsf{minPO}_i \prec \mathcal{D}_l$. Then, by Property 3.4.1, we have $\tau_j \in \mathsf{minPO}_i$. □

Lemma 3.4.3 *If at Line 3, each duration d has $\mathsf{d}.tlist$ containing only non-overlapping transactions, then transaction τ_i will be allocated into only its valid durations.*

Proof.

Consider a duration d and let $\mathcal{M} = \{\tau_l \in \mathsf{d}.tlist : \mathsf{maxid}_l \geq \mathsf{minid}_i\}$ and $\tau_j = \arg\min_{\tau_l \in \mathcal{M}} (\mathsf{maxid}_l)$. We prove this lemma by showing that at Line 6, if there exists transaction τ_k in $\mathsf{d}.tlist$ that overlaps τ_i (thus d is invalid) then $\tau_j \equiv \tau_k$ (thus if-condition at Line 9 is false). Note that τ_k does not overlap τ_i when $\mathsf{maxid}_k < \mathsf{minid}_i$ because all PO-sets containing τ_k are different with that of τ_i. Hence, we can assume that $\tau_k \in \mathcal{M}$. Assume by contradiction that $\tau_j \not\equiv \tau_k$.

We will first prove that the propositions A, B, and C are true, where $A \equiv (\mathsf{minPO}_k \preceq \mathsf{minPO}_i)$, $B \equiv (\mathsf{minPO}_j \preceq \mathsf{minPO}_i)$, $C \equiv (\mathsf{minPO}_k \preceq \mathsf{maxPO}_j)$. For A, since $\tau_k \in \mathcal{M}$ and τ_k has been allocated before τ_i, we have $\mathsf{minid}_k \leq \mathsf{minid}_i \leq \mathsf{maxid}_k$. Then since $\mathsf{minPO}_k \preceq \mathsf{maxPO}_k$ (Property 3.4.4), by Property 3.4.3, we have A is true. We can prove B true in a similar way by replacing τ_k with τ_j. For C, since $\tau_j \in \mathcal{M}$ and τ_k is allocated before τ_i, we have $\mathsf{minid}_k \leq \mathsf{minid}_i \leq \mathsf{maxid}_j$. Together with the definition of τ_j, we have $\mathsf{minid}_k \leq \mathsf{maxid}_j \leq \mathsf{maxid}_k$. Then by Property 3.4.3, we have C is true.

Due to the tree structure of the PO-tree, from A and B, we have either proposition D or E is true, where $D \equiv (\mathsf{minPO}_j \prec \mathsf{minPO}_k)$ and $E \equiv (\mathsf{minPO}_k \preceq \mathsf{minPO}_j)$. If D is true, then together with C, we have F is true, where $F \equiv (\mathsf{minPO}_j \prec \mathsf{minPO}_k \preceq \mathsf{maxPO}_j)$. By Property 3.4.1, F implies that $\tau_j \in \mathsf{minPO}_k$. This, in turn, implies that τ_j overlaps τ_k which contradicts the lemma's assumption. If E is true, then together with B, we have G is true, where $G \equiv (\mathsf{minPO}_k \preceq \mathsf{minPO}_j \preceq \mathsf{minPO}_i)$. Since $\tau_k \in \mathsf{minPO}_k$ and $\tau_k \in \mathsf{minPO}_i$ (Lemma 3.4.2), by Property 3.4.1, G implies that $\tau_k \in \mathsf{minPO}_j$. This, in turn, implies that τ_j overlaps τ_k which, again, contradicts the lemma's assumption. Therefore $\tau_j \equiv \tau_k$ which means d is only invalid for τ_i when τ_i overlaps τ_j, otherwise d is valid and τ_i will be allocated into slots of d (Line 10 to 18).
□

Lemma 3.4.4 *Each transaction is allocated into only its valid durations.*

Proof.

Since, at the first iteration of Line 3, the assumption of Lemma 3.4.3 is true (because $\mathsf{d}.tlist = null$), by Lemma 3.4.3 and the operation from Line 10 to 18, we have τ_1 is allocated into valid durations and, at the end of this iteration, every duration d has $\mathsf{d}.tlist$ containing only non-overlapping transactions. Hence, the assumption of Lemma 3.4.3 is true again at the second iteration of Line 3. The proof, then, is complete by induction reasoning. □

Theorem 3.4.2 POBaseNoC *is optimal for same-period acyclic transaction sets.*

Proof.

We will show that if transaction set satisfies the necessary condition in Theorem 3.4.1, then at the end of the algorithm, the number of valid slots assigned to each transaction τ_i is e_i. Consider when transaction τ_i is being allocated. By Lemma 3.4.2, we have that all transactions whose schedule is allocated before τ_i and overlap τ_i must belong to minPO_i. Therefore invalid slots of τ_i must have all been assigned to transactions in minPO_i. Assume by contradiction that at the end of the algorithm, the number of valid slots assigned to τ_i is smaller than e_i. Since the algorithm guarantees that a transaction can only be assigned to all slots of a duration or none of them (Line 12 and 16), *all* and *only* valid slots of a transaction are contained in its valid durations. Therefore, the contradiction assumption is true only if at Line 4, the number of valid slots of τ_i is smaller than e_i. This implies that $p - \sum_{\tau_j \in \mathsf{minPO}_i \setminus \{\tau_i\}} e_j < e_i$. This contradicts with condition in Theorem 3.4.1 which implies that $\sum_{\tau_j \in \mathsf{minPO}_i} e_j \leq p$. In other words, there are always enough valid slots to allocate τ_i. Since this is also true for all other transaction, the proof completes.

□

POBaseNoC **Analysis**: Since a new duration is added only when a transaction assignment is complete i.e. $r = 0$ (Line 15), the maximum number of durations in Dur is N. Therefore, the for-loop between Line 3 and 18 is executed at most N^2 times. Since we can implement d.$tlist$ as a sorted queue based on maxid, the operation to look up τ_j at Line 6 and 7 will take $O(\log(N))$ time. And the operation to insert new item into the ordered list d.$tlist$ at Line 11 and 15 will also take $O(\log(N))$ time. Furthermore, since the rest of the code between Line 6 and 18 can be implemented with a constant number of operations, the time complexity of POBaseNoC is $O(N^2 \log(N))$. Finally, since there is at most N durations, we need at most $O(N)$ space to store the schedule of each transactions.

The GenerateLoadNoC **procedure**

As mentioned in Section 3.4.2, the second condition for POGen to work is that: procedure GenerateLoadNoC can generate a feasible load set $\{l_i^k : \forall \tau_i \in \mathcal{T}\}$ for interval int^k when its inputs satisfy Inequalities 3.4.2 and 3.4.3. A feasible load set is one that satisfies Inequalities 3.3.5 and 3.3.6. In this section, we will present a version of GenerateLoadNoC. There are two questions that have to be answered: (1) is there a feasible load set? (2) is there an efficient algorithm to find it? We will show that the problem of finding a feasible load set is equivalent to the problem *Circulations in Graphs with Demands and Lower bounds* [20] where the demand at every vertex is 0. This is the problem of finding a feasible circulation flow in a directed graph G

whose each edge has a capacity and a lower bound. Furthermore, we will prove that if the utilization of each PO-set is smaller than the utilization bound expressed by Inequalities 3.4.4, there always exists a feasible solution therefore answering Question 1. Then, since the Ford-Fulkerson algorithm [20] can be used to solve the problem, Question 2 is also answered.

In the following, we will describe the construction of the directed graph G from the input of $\mathsf{GenerateLoadNoC}$. Graph G is constructed such that: 1) the value of the flow on an edge represents the interval load of a transaction or the total interval load of a PO-set. Flow values are subject to the same upper bounds and lower bounds of the loads they represent (as in Inequalities 3.3.5 and 3.3.6); 2) the equality between the sum of the input-flow values and the sum of the output-flow values at a vertex represents the equality between the total interval load of some PO-sets and the sum of the interval loads of the transactions in these PO-sets. The construction guarantees that the value of a feasible flow on an edge is also a feasible interval load of the transaction or the PO-set it represents.

Graph construction: we define a tuple $G = (V, E)$ as follows:

Vertexes of G:

- For each PO-set $\mathcal{D}_l$, define a vertex v_l.
- The set of vertices of G is $V = \{v^*, v_l : l \in [1, N^{\mathcal{D}}]\}$ where v^* is an additional vertex.

Edges and flow values of G:

- For each PO-set $\mathcal{D}_l$, define a directed edge $g_l^{\mathcal{D}}$ which is called a *PO-set edge*. Furthermore, define for each edge $g_l^{\mathcal{D}}$ two integer constants $c_l^{\mathcal{D}}$ and $b_l^{\mathcal{D}}$ which are called the capacity and the lower bound of edge $g_l^{\mathcal{D}}$ where $c_l^{\mathcal{D}} = \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \rceil)$ and $b_l^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \rfloor$. Finally, define for each edge $g_l^{\mathcal{D}}$ a real variable $x_l^{\mathcal{D}}$ which is called the flow value of the edge. The flow value is subject to the constraints in Inequality 3.4.5 (which is the same as Inequality 3.3.6):

$$\forall \mathcal{D}_l \subset \mathcal{T} : b_l^{\mathcal{D}} \le x_l^{\mathcal{D}} \le c_l^{\mathcal{D}}. \tag{3.4.5}$$

- For each transaction τ_i, define a directed edge g_i which is called a *transaction edge*. Furthermore, define for each edge g_i two integer constants c_i and b_i which are called the capacity and the lower bound of edge g_i where $c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$ and $b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor$. Finally, define for each edge g_i a real variable x_i which is called the flow value of the edge. The flow value is subject to the constraints in Inequality 3.4.6 (which is the same as Inequality 3.3.5):

$$\forall \tau_i \in \mathcal{T} : b_i \leq x_i \leq c_i. \tag{3.4.6}$$

- The set of edges of G is: $E = \{g_i : i \in [1, N]\} \cup \{g_l^{\mathcal{D}} : j \in [1, N^{\mathcal{D}}]\}$. The total number of edges is $|E| = N + N^{\mathcal{D}}$.

Rules for directing edges:

- The set of edges that enter v^* is $\mathsf{Enter}^* = \{g_i : \tau_i \in \mathcal{D}_1\}$; the set of edges that exit v^* is $g_1^{\mathcal{D}}$.
- The set of edges that enter v_l is PO-set edge $g_l^{\mathcal{D}}$ and transaction edges representing transactions that are in children of $\mathcal{D}_l$ but not in $\mathcal{D}_l$, i.e. $\mathsf{Enter}_l = \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \{g_i : \tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l\}$.
- The set of edges that exit v_l are PO-set edges $\{g_m^{\mathcal{D}} : \mathcal{D}_m \in \mathsf{children}(\mathcal{D}_\mathsf{l})\}$ and transaction edges representing transactions that are in $\mathcal{D}_l$ but not in children of $\mathcal{D}_l$, i.e. $\mathsf{Exit}_l = \big\{g_i : \tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m\big\}$.

Flow conservation constraint:

- The flow values in graph G is subject to the flow conversation constraint [20] in which given a vertex, the sum of the flow values entering it minus the sum of the flow values exiting it is zero.

Figure 3.19 shows graph G constructed from the transaction set shown in Figure 3.15. Consider vertex v_1. Edges that enter v_1 are $g_1^{\mathcal{D}}$ and $\mathsf{Enter}_1 = \{g_5, g_6, g_8, g_9\}$. Edges in Enter_1 are edges that represent transactions which belong to $\mathsf{children}(\mathcal{D}_1) = \{\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_5\}$ but do not belong to $\mathcal{D}_1$. Furthermore, edges that exit v_1 are $\{g_2^{\mathcal{D}}, g_3^{\mathcal{D}}, g_5^{\mathcal{D}}\}$ and $\mathsf{Exit}_1 = \{g_2\}$. As we will prove later, this construction guarantees that the equality between the sum of the input-flow values and the sum of the output-flow values at v_1 (due to the flow conservation constraint) represents the equality between the sum of interval loads of transactions in every child of $\mathcal{D}_1$ and the total interval load of its children.

We will prove in Lemma 3.4.7 that graph G is indeed a directed graph in which every edge is directed and has two endpoints. The proof will use Lemma 3.4.5, and 3.4.6.

Lemma 3.4.5 *For every $\mathcal{D}_l, \mathcal{D}_m : \mathcal{D}_l \neq \mathcal{D}_m$, the following is true:*

$$\big(\mathcal{D}_l \setminus \mathsf{parent}(\mathcal{D}_l)\big) \cap \big(\mathcal{D}_m \setminus \mathsf{parent}(\mathcal{D}_m)\big) = \emptyset. \tag{3.4.7}$$

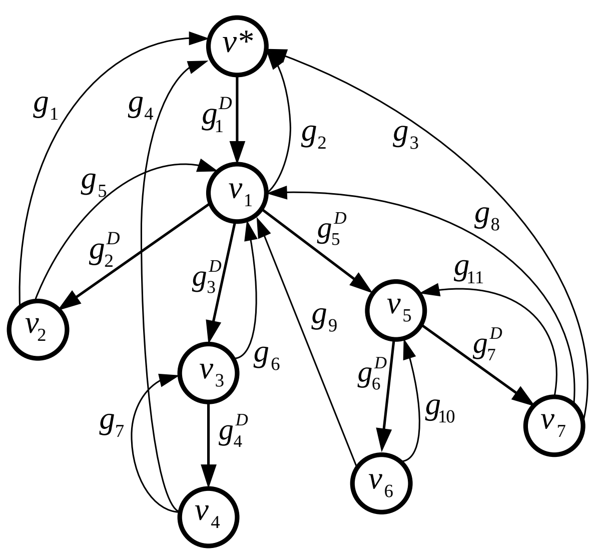


Figure 3.19: Graph G of $\mathcal{T}$ shown in Figure 3.15

Proof.

Since the lemma is trivial when $\mathcal{D}_l = \mathsf{parent}(\mathcal{D}_m)$ or $\mathcal{D}_m = \mathsf{parent}(\mathcal{D}_l)$, we assume that these conditions are false. If $\mathcal{D}_l \not\prec \mathcal{D}_m$ and $\mathcal{D}_m \not\prec \mathcal{D}_l$, then the lemma is true because of Property 3.4.2. If $\mathcal{D}_l \prec \mathcal{D}_m$, since $\mathcal{D}_l \neq \mathsf{parent}(\mathcal{D}_m)$, we have $\mathcal{D}_l \prec \mathsf{parent}(\mathcal{D}_m) \prec \mathcal{D}_m$. By Property 3.4.1, for every τ_i where $\tau_i \in \mathcal{D}_l$ and $\tau_i \in \mathcal{D}_m$ we have $\tau_i \in \mathsf{parent}(\mathcal{D}_m)$. Therefore $\mathcal{D}_l \cap \big(\mathcal{D}_m \setminus \mathsf{parent}(\mathcal{D}_m)\big) = \emptyset$, which proves the lemma. Finally, using similar technique and interchanging $\mathcal{D}_l$ with $\mathcal{D}_m$, we can also prove the lemma when $\mathcal{D}_m \prec \mathcal{D}_l$. □

Lemma 3.4.6 *For every $\mathcal{D}_l, \mathcal{D}_m : \mathcal{D}_l \neq \mathcal{D}_m$, the following is true:*

$$\begin{aligned}&\{\mathcal{D}_l \setminus \bigcup_{\mathcal{D}_n \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_n\} \bigcap \\ &\{\mathcal{D}_m \setminus \bigcup_{\mathcal{D}_n \in \mathsf{children}(\mathcal{D}_m)} \mathcal{D}_n\} = \emptyset.\end{aligned} \tag{3.4.8}$$

Proof.

Since the lemma is trivial when $\mathcal{D}_l = \mathsf{parent}(\mathcal{D}_m)$ or $\mathcal{D}_m = \mathsf{parent}(\mathcal{D}_l)$, we assume that these conditions are false. If $\mathcal{D}_l \not\prec \mathcal{D}_m$ and $\mathcal{D}_m \not\prec \mathcal{D}_l$, then the lemma is true because of Property 3.4.2. If $\mathcal{D}_l \prec \mathcal{D}_m$, since $\mathcal{D}_l \neq \mathsf{parent}(\mathcal{D}_m)$, we have there exists $\mathcal{D}_n \in \mathsf{children}(\mathcal{D}_l)$ where $\mathcal{D}_l \prec \mathcal{D}_n \prec \mathcal{D}_m$. By Property 3.4.1, for every τ_i where $\tau_i \in \mathcal{D}_l$ and $\tau_i \in \mathcal{D}_m$ we have $\tau_i \in \mathcal{D}_n$. Therefore $\big(\mathcal{D}_l \setminus \mathcal{D}_n\big) \cap \mathcal{D}_m = \emptyset$, which proves the lemma. Finally, using similar technique and interchanging $\mathcal{D}_l$ with $\mathcal{D}_m$, we can also prove the lemma when $\mathcal{D}_m \prec \mathcal{D}_l$. $\square$

Lemma 3.4.7 *G is a directed graph where $|E| \leq 2N$.*

Proof.

Since every edge of G is directed, it remains to show that each edge enters one and only one vertex and exits one and only one vertex. Note that there is one edge defined for each PO-set and one edge defined for each transaction.

Consider PO-set edges: By the rules of directing edges, PO-set edge g_1^D exits only vertex v^*. Furthermore, if $\mathcal{D}_l = \mathsf{parent}(\mathcal{D}_m)$, the PO-set edge g_m^D exits vertex v_l. Since each $\mathcal{D}_m$ where $\mathcal{D}_m \neq \mathcal{D}_1$ has one and only one parent, the PO-set edge g_m^D exits one and only one vertex. Therefore, each PO-set edge exits one and only one vertex. In addition, each PO-set edge g_l^D enters only vertex v_l. Therefore each PO-set edge enters one and only one vertex.

Consider transaction edges: By the rules of directing edges, the set of transaction edges that enter v^* is Enter^* and enter v_l is Enter_l. By Lemma 3.4.5, the collection of sets $\{\mathsf{Enter}^*, \mathsf{Enter}_l : \forall l\}$ is pairwise disjoint. Therefore each transaction edge g_i will appear in only one of the sets in the collection. Furthermore, since $\bigcup_{\mathcal{D}_m \in \mathcal{T}} \mathcal{D}_m \setminus \mathsf{parent}(\mathcal{D}_m) = \mathcal{T}$, each transaction edge g_i must appear in at least one of the sets in the collection $\{\mathsf{Enter}^*, \mathsf{Enter}_l : \forall l\}$. In other words, each transaction edge g_i enters one and only one vertex.

By the rules of directing edges, the set of transaction edges that exits v^* is empty and exits v_l is Exit_l. By Lemma 3.4.6, the collection of sets $\{\mathsf{Exit}_l : \forall l\}$ is pairwise disjoint. Therefore each transaction edge g_i will appear in only one of the sets in the collection. Furthermore, since $\bigcup_{\mathcal{D}_l \in \mathcal{T}} (\mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m) = \mathcal{T}$, each transaction edge g_i must appear in at least one of the sets in the collection $\{\mathsf{Exit}_l : \forall l\}$. In other words, each transaction edge g_i exits one and only one vertex.

Now we will prove that $|E| \leq 2N$, since each transaction edge exits one and only one vertex and each vertex except v^* has at least one transaction edge exists from it, we have $|V \setminus \{v^*\}| \leq N$. Since by definition $|V \setminus \{v^*\}| = N^{\mathcal{D}}$ and $|E| = N + N^{\mathcal{D}}$, we have $|E| \leq 2N$ $\square$

It remains to show that $\mathsf{GenerateLoadNoC}$ honors Condition 3.4.2. For simplicity of exposition, we split the proof in multiple lemmas. First, Lemma 3.4.8 proves an important property of graph G regarding the flow values. Then, this property will be used to prove in Lemma 3.4.9 that graph G has a feasible flow if Inequalities 3.4.2, 3.4.3 are satisfied for interval int^k and furthermore all PO-sets satisfy the utilization bound shown in Inequality 3.4.4. Note that we know from [20] that if graph G has a feasible flow, then it has an integral feasible flow which can be found by the Ford-Fulkerson algorithm [20]. Therefore, to complete the proof, we will have to prove that a feasible load set can be derived from an integral feasible flow of G (Lemma 3.4.10).

Lemma 3.4.8 *A flow in graph G honors the flow conservation constraint at every vertex v_l if and only if the following equalities hold for every PO-set $\mathcal{D}_l$:*

$$\sum_{\tau_i \in \mathcal{D}_l} x_i = x_l^{\mathcal{D}}. \tag{3.4.9}$$

Proof.

We prove the lemma by induction.

Basis case: Consider v_l where $\mathsf{height}(\mathcal{D}_l) = 0$ or equivalently $\mathsf{children}(\mathcal{D}_l) = \emptyset$. By the rules of directing edges, we have the set of edges that enter v_l is $\{g_l^{\mathcal{D}}\}$ and the set of edges that exit v_l is $\{g_i : \tau_i \in \mathcal{D}_l\}$. Therefore, the flow conservation constraint holds at vertex v_l if and only if $\sum_{\tau_i \in \mathcal{D}_l} x_j = x_l^{\mathcal{D}}$.

Induction case: Assume that the hypothesis is true $\forall \mathcal{D}_l$ where $\mathsf{height}(\mathcal{D}_l) \leq H - 1$. We will prove that the hypothesis is also true $\forall \mathcal{D}_l$ where $\mathsf{height}(\mathcal{D}_l) = H$. By definition of the tree node height, the induction hypothesis implies that $\forall \mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)$, v_m honors the flow conservation constraint if and only if:

$$\sum_{\tau_i \in \mathcal{D}_m} x_i = x_m^{\mathcal{D}}. \tag{3.4.10}$$

By the rules of directing edges and Lemma 3.4.5, the total value of flows that enters vertex v_l is:

$$x_j^{\mathsf{enter}} = \sum_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l} x_i + x_l^{\mathcal{D}},$$

and the total value of flows that exits vertex v_l is:

$$x_j^{\mathsf{exit}} = \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i + \sum_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} x_m^{\mathcal{D}}.$$

To complete the induction step, it remains to show that $x_j^{\mathsf{enter}} = x_j^{\mathsf{exit}}$ if and only if Equation 3.4.9 holds.

Note that by Equation 3.4.10, $x_j^{\mathsf{enter}} = x_j^{\mathsf{exit}}$ is equivalent to:

$$\begin{aligned} x_l^{\mathcal{D}} &= -\sum_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l} x_i \\ &+ \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i + \sum_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m} x_i \\ &= \sum_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \cap \mathcal{D}_l} x_i + \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i \end{aligned} \tag{3.4.11}$$

Finally, since by Property 3.4.2, PO-sets in $\mathsf{children}(\mathcal{D}_l)$ are pairwise disjointed sets, Equation 3.4.11 is equivalent to:

$$\begin{aligned} x_l^{\mathcal{D}} &= \sum_{\tau_i \in \mathcal{D}_l \bigcap \left(\bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m \right)} x_i \\ &+ \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \mathsf{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i = \sum_{\tau_i \in \mathcal{D}_l} x_i \end{aligned}$$

This completes the proof. □

Lemma 3.4.9 *There exists a feasible flow in graph G if Inequalities 3.4.2, 3.4.3 are satisfied for interval* int^k *and all PO-sets satisfy the utilization bound in Inequality 3.4.4.*

Proof.

First note that Inequalities 3.4.2 are necessary for the edge constraints on each PO-set edge (Inequality 3.4.5) to be satisfied. Let us construct a flow as follows.

$$\begin{aligned} &\forall \tau_i \in \mathcal{T} : x_i = \mathsf{lag}(\tau_i, \mathsf{int}^k) \\ &\forall \mathcal{D}_l \subset \mathcal{T} : x_l^{\mathcal{D}} = \mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \end{aligned}$$

We will have to prove that the constructed flow satisfies the edge constraints and the flow conservation constraints. Given the constructed flow, it is easy to verify that the edge constraints of each transaction edge (Inequality 3.4.6) and the left-side edge constrains of each PO-set edge (Inequality 3.4.5) are satisfied. The right-side edge constraints of each PO-set edge are satisfied because by the definition of the lag function and by Inequalities 3.4.3, before the execution of $\mathsf{GenerateLoadNoC}$ for interval int^k we have the following:

$$\begin{aligned}
&\mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) = u_l^{\mathcal{P}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_l} \sum_{x \in [0, t^k)} S(\tau_i, x) \\
&\leq u_l^{\mathcal{P}} * t^{k+1} - \lfloor u_l^{\mathcal{P}} * t^k \rfloor \\
&< u_l^{\mathcal{P}} * t^{k+1} - u_l^{\mathcal{P}} * t^k + 1. \qquad (3.4.12)
\end{aligned}$$

Since $\mathsf{L} \leq \min_k(\mathsf{int}^k)$, we have $\frac{\mathsf{L}-1}{\mathsf{L}} \leq \frac{|\mathsf{int}^k|-1}{|\mathsf{int}^k|}$. Hence, by Inequality 3.4.4, we have $u_l^{\mathcal{P}} \leq \frac{|\mathsf{int}^k|-1}{|\mathsf{int}^k|}$. Apply this inequality to Inequality 3.4.12, we have:

$$\mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) < u_l^{\mathcal{P}} * t^{k+1} - u_l^{\mathcal{P}} * t^k + 1 \leq |\mathsf{int}^k|.$$

Since furthermore $\mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \leq \lceil \mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \rceil$, it follows that $\mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \leq \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}_l, \mathsf{int}^k) \rceil)$.

It remains to verify that the flow conservation constraint is honored at every vertex. Since the constructed flow satisfied Equation 3.4.9, the sufficient condition of Lemma 3.4.8 proves this statement. □

Lemma 3.4.10 *If there is an integral feasible flow in graph G, then there is a feasible load set where*

$$\forall \tau_i \in \mathcal{T} : l_i^k = x_i \qquad (3.4.13)$$

Proof.

We have to prove that the load set where $\forall \tau_i \in \mathcal{T} : l_i^k = x_i$ satisfies Inequalities 3.3.5 and 3.3.6. By the transaction edge constraints in Inequality 3.4.6, the following inequality holds.

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$$

Thus the interval loads satisfy Inequality 3.3.5.

Furthermore, by the necessary condition of Lemma 3.4.8, we have:

$$\forall \mathcal{D}_l \in \mathcal{T} : \sum_{\tau_i \in \mathcal{D}_l} x_i = x_l^{\mathcal{P}}. \qquad (3.4.14)$$

Then since $\sum_{\tau_i \in \mathcal{D}_l} l_i^k = \sum_{\tau_i \in \mathcal{D}_l} x_i$ and $x_l^{\mathcal{P}}$ is subject to PO-set edge constraints in Inequality 3.4.5, Inequality 3.3.6 is satisfied. □

Algorithm analysis: Since $\forall g_i \in E : c_i - b_i \leq 1$, $\forall g_l^{\mathcal{D}} \in E : c_l^{\mathcal{D}} - b_l^{\mathcal{D}} \leq 1$ and $N^{\mathcal{D}} \leq N$, we have $\Delta = \sum_{g_i \in E} (c_i - b_i) + \sum_{g_l^{\mathcal{D}} \in E} (c_l^{\mathcal{D}} - b_l^{\mathcal{D}}) \leq 2N$. The time complexity of the Ford-Fulkerson algorithm in finding a feasible circulation in graph G is $O(|E| f^{max})$ where f^{max} is the maximum flow value of a graph derived from G and $f^{max} \leq \Delta$ (see [20] for details). Since $\Delta \leq 2N$ and $|E| \leq 2N$, the time complexity of GenerateLoadNoC is $O(N^2)$. Finally, since the time complexity of POBaseNoC is $O(N^2 \log(N))$, the time complexity of POGen to generate the schedule for each scheduling interval is $O(N^2 \log(N))$.

The sufficient utilization bound of POGen: As shown in Lemma 3.4.9, GenerateLoadNoC induces a sufficient utilization bound shown in Inequality 3.4.4. Since this bound is stricter than that of POBaseNoC, it becomes a sufficient utilization bound of POGen.

3.5 Implementation

In this section, we discuss some practical factors in the implementation of POGen in real systems and show the experimental measurement of the algorithm execution overheads.

An important parameter in the operation of POGen is the size of L in number of slots because it dictates the POGen's sufficient utilization bound (shown in Inequalities 3.4.4). Note that the real-time value of L is determined by the applications. As shown in [27], typical real-time applications have periods that are multiple of milliseconds. Therefore, we believe it can be assumed that the greatest common divisor of all transaction periods is at least 1ms. Meanwhile a typical modern SoC bus [2, 10, 12] has bus clock frequency no smaller than 100MHz. Therefore, it is reasonable to select the slot size to be no bigger than 10us (equivalent to 1000 bus cycles), which results in the value of L in number of slots to be no smaller than 100. The utilization bound, therefore, will be 0.99. It is worth noting that the size of a slot does *not* affect the algorithm overhead because the time complexity POGen only depends on the number of transactions N.

Implementing our scheduling framework requires two components: 1) POGen is executed on each processing element at the beginning of each interval to generate the interval's schedule and 2) a *slot scheduler* transmits a given transaction in its assigned slots according to the generated interval schedule. The implementation of the slot scheduler depends on the NoC architecture. In a software controlled NoC such as in the CellBE processor [2], the slot scheduler can be implemented in software on each processing element by using an interrupt handler to trigger the scheduler at the beginning of each duration; since there are at most N durations, on each processing element there are at most N interrupts within each scheduling intervals. In

N	10	20	30	40
Average (us)	4.5	6.4	9.5	12.1
Max (us)	6.9	7.7	11.2	13.8

Table 3.1: POGen execution overhead

a custom NoC, the slot scheduler could be implemented in the router connected to the processing element.

In order to measure POGen overhead in the real SoC systems, we implemented POGen on a IBM CellBE processor platform [7]. We selected Cell processor because it represents a typical modern SoC whose components are interconnected by a software controllable NoC. We generated random transaction sets using the same method described in Section 3.6. Table 3.1 shows the average and maximum execution time of POGen given various transaction sets size N. Given the smallest scheduling interval to be 1ms, the overhead is less than 1.5% of the scheduling interval size.

3.6 Evaluation

Most of the previous related works [43, 44, 3, 24, 28] have focused on the Fixed-priority Scheduling Algorithm (FPA). These works deal with the methods for schedulability analysis and priority assignment. More specifically, Shi et al. have recently proposed in [43] a branch-and-bound algorithm that searches for a feasible priority set for a transaction set. If a feasible priority set exists, then the transactions set is guaranteed to be schedulable under the worst-case transaction latency (WTL) analysis proposed in [44]. To the best of our knowledge, the works in [44, 43] are the state of the art.

In this section, we are interested in comparing the performance of POGen with the solution proposed in [44, 43] assuming *acyclic* transaction sets[4]. The concerned performance metric is the acceptance rate of POGen and the FPA where the acceptance rate is calculated as the number of schedulable transaction sets over the total number of generated transaction sets. A transaction set is schedulable under POGen if it passes the utilization bound test shown in Inequality 3.4.4, whereas it is schedulable under the FPA if it has a feasible priority set generated by the algorithm in [43].

To generate random transaction sets for the experiments, we used similar parameters and methods used in [43] except that we are only concerned with acyclic transaction sets. The transactions' sources and destinations are randomly generated on square 2D mesh and transactions are routed using the dimension-

[4]Note that algorithms in [44, 43] can also be used for cyclic transaction sets.

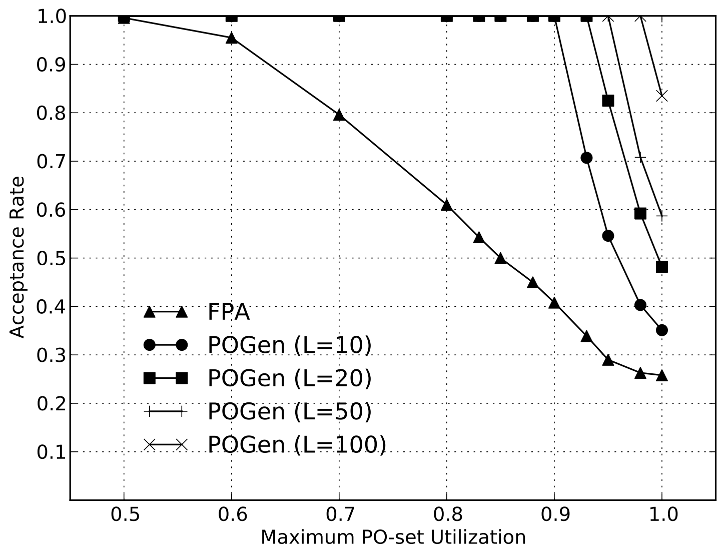


Figure 3.20: Acceptance rate versus PO-set utilizations

order X-Y routing. Like [43], we use the maximum PO-set utilization of a transaction set (which is ”the maximum link utilization” in [43]) as a controlled variable. Given a maximum PO-set utilization u^{max}, the utilization of transactions is generated according to the uniform distribution algorithm in [5] such that the utilizations of all PO-sets are no larger than u^{max}. The transmission time e_i of transaction τ_i is a uniformly-distributed random number in the range from 1 to 1024 slots. The period p_i is then determined as a multiple of L using the following formula: $\lceil e_i/(u_i * \mathsf{L}) \rceil * \mathsf{L}$. Given a pair of $\{e_i, p_i\}$, we recalculate u_i to be e_i/p_i. Following the discussion in Section 3.3.4, we generated transaction sets with L to be 10, 20, 50, and 100 slots. In the following experiments, there are 1000 transaction sets generated at each measurement point.

Figure 3.20 shows the acceptance rates of FPA and POGen with various maximum PO-set utilization. The size of NoC is 10x10 and each transaction set has 20 transactions. For POGen, we report the results with L equals to 10, 20, 50, 100. The better performance of POGen comes from the fact that the WTL analysis in [44] does not take advantage of the parallelism between non-overlapping transactions. For example, consider the transaction set shown in Figure 3.1. Assume τ_6 and τ_7 have higher priority than τ_4. According to the WTL analysis in [44], the interference of transactions τ_6 and τ_7 on the execution of τ_4 is calculated as if all transactions were using a single-shared resource. However, POGen allows τ_6 and τ_7 to be executed in parallel as shown in Figure 3.18.

Figure 3.21 shows the acceptance rate of FPA and POGen with various transaction set sizes and NoC sizes. We report the result where L equals to 10 slots and the maximum PO-set utilization is 0.95. In most

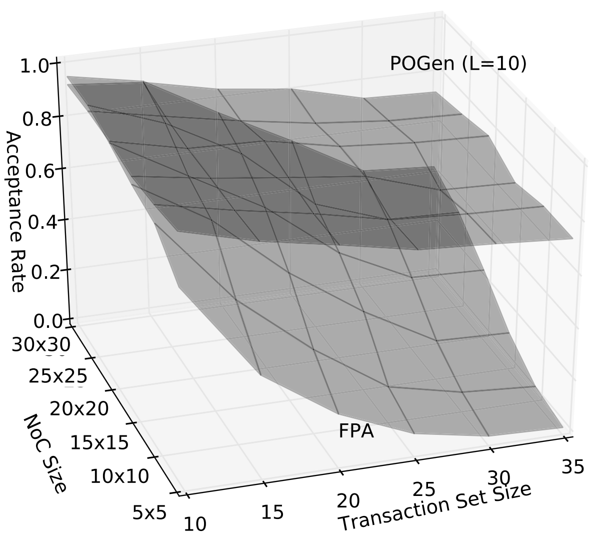


Figure 3.21: Acceptance rate versus NoC sizes and N

cases, the acceptance rate of POGen is higher than that of FPA especially when transaction set size is higher and the NoC size is smaller. The reason is that in these situations, there are more transaction overlaps. Therefore, FPA suffers more from the effect described in the previous paragraph.

3.7 Conclusion

In this research, we investigated on the problem of real-time communication scheduling on multi-core processor buses. This scheduling problem has main assumptions about resource sharing pattern that are different with that of traditional real-time problems. In particular, transactions on multi-core processor buses may depend on each other in an intransitive manner. We proposed a novel scheduling framework that are able to dynamically schedule transactions by taking into account the intransitive dependence between them. Compared to related works, our algorithms achieves much higher bus utilization. Our research has also opened up a new scheduling paradigm that worths further investigations.

Chapter 4

System Design for Predictable Memory Access

As described in Chapter 1, real-time systems design is facing significant challenges to handle in a predictable manner the interaction between the CPU cache and peripherals while they are accessing the shared main memory. In modern computer architectures, peripherals are connected to the system through a peripheral interconnection with master capabilities (DMA), such as the PCI bus, can directly initiate read/write transactions toward either other peripherals or the main memory. Bus master mode is essential to avoid overloading the processor, especially in the case of fast I/O interfaces that could otherwise produce millions of interrupts per second. However, since the memory is a shared resource in the system, peripheral transactions can interfere with cache line fetches produced by the CPU memory controller whenever a task experiences a cache miss. This interaction can slow down task execution tremendously: our experiments in Section 4.3 show that task execution time in the presence of heavy I/O load is increased up to 44%.

In general, two types of solutions can be feasibly applied to this problem. The first is to account for the effect of all peripheral traffic in the worst case computation time (WCET) of each task. In this case, we need an analysis that can compute the increase in WCET given design-time bounds on peripheral traffic. The problem of this solution is that, as already mentioned, the WCET increment can be very large, up to 44%. The approach that we propose in this research is to *coschedule* CPU tasks and I/O transactions: in fact, assuming we find a way to control when peripherals are allowed to transmit, we can create a bus transmission schedule and synchronize it with the CPU task schedule. We can then formulate our coscheduling objective as follows: maximize the traffic transmitted by each peripheral, while guaranteeing that each task meets its

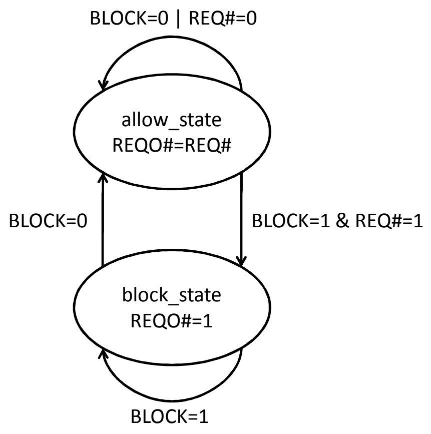


Figure 4.1: Peripheral Gate State Machine

real-time deadline.

This research has two main contributions. First, in Section 4.1 we introduce the design of a device for the PCI/PCI-X bus, called a *peripheral gate* (or *p-gate* for short), that allows us to control peripheral access to the bus. The implemented p-gate is compatible with existing computer architectures: no modification to either the peripheral or the motherboard is required. Second, we propose in Section 4.2 a novel coscheduling method to coordinate tasks' execution and peripheral operations. More specifically, we divide each task into a series of *superblocks*; each superblock can include branches and loops, but superblocks must be executed in sequence. By running the task, the CPU can collect information on the number of cache misses in each superblock. We can then compute a safe WCET bound by determining a worst case arrival pattern of cache misses in each superblock. The coscheduling heuristic uses the WCET analysis and the run-time information provided by the OS to compute available task slack and it dynamically opens the p-gates when it is safe to do so. In Section 4.3, we show that our heuristic performs well compared to the best possible run-time, adaptive and predictive algorithm.

4.1 Peripheral Gate

In this section, we first provide a brief overview of the Peripheral Component Interconnect (PCI) standard and then describe our p-gate implementation. PCI is the current standard family of architectures for motherboard - peripheral interconnection in the personal computer market; it is also widely popular in the embedded domain [34]. The standard can be divided in two parts: a *logical* specification, which details how

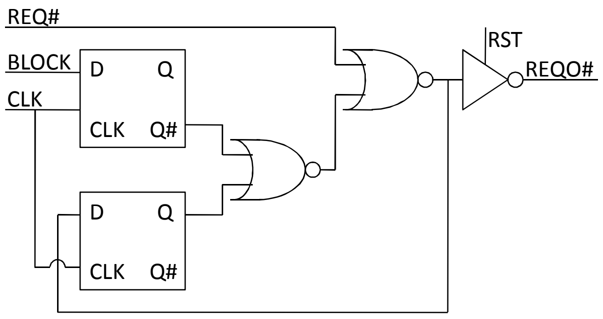


Figure 4.2: Peripheral Gate Schematic

the CPU configures and accesses peripherals through the system controller, and a *physical* specification, which details how peripherals are connected to and communicate with the motherboard. Several widely different physical specifications have been published; here we focus on the PCI/PCI-X physical specification, which uses a shared bus architecture with support for multiple bus segments connected by bridges. To gain access to the shared bus, each peripheral must first obtain permission from the bus segment arbiter using a standard handshake with two point-to-point, active-low wires, `REQ#` and `GNT#`. The peripheral first lowers `REQ#` to signal a request for the bus, and the arbiter grants permission by lowering `GNT#`. The peripheral then waits for the bus to become free and starts a data transfer (also called a bus *transaction*). The handshake finishes after both the peripheral and the arbiter raise `REQ#` and `GNT#` in succession; if the peripheral wants to initiate another transaction, it must reacquire the grant.

We implemented the p-gate based on a PCI extender card, a debug card that is interposed between the peripheral card and the motherboard and provides easy access to all signals. We modified the card to intercept the `REQ#` signal and to control it based on an input `block` signal coming from the reservation controller. The main idea is to force `REQ#` to remain high whenever `block` is active; in this way, the peripheral is not able to get the grant from the arbiter and thus can not transmit. The actual implementation is more complex: if `block` is raised while `REQ#` is active low, we could violate the PCI specification by immediately deactivating `REQ#`. Instead, we must allow the current request to finish and then we can block all further requests. A corresponding synchronous state machine is shown in Figure 4.1 and an optimized schematic in Figure 4.1, where `REQ#` is the input from the peripheral and `REQO#` is the controlled output to the arbiter. Our implementation uses discreet components: two positive-edge-triggered D flip-flops, two nor gates and an inverting tri-state buffer. The output buffer is required by the specification to set the output to high impedance whenever the bus is reset. We measured a worst case propagation delay for the circuit of

7ns, which allowed us to run the bus at a frequency up to 66Mhz.

The reservation controller outputs a `block` signal for each p-gate in the system. We implemented a prototype reservation controller based on a Xilinx ML505 board. The board is connected to the system using a PCI-E motherboard slot, and uses a Virtex-5 FPGA to implement a custom peripheral. All registers used by the peripheral are memory mapped; a PCI driver is used to allocate the registers in the CPU virtual memory space, hence tasks running in user mode can communicate with the peripheral performing memory reads/writes. The reservation controller can run in two different modes: in *data acquisition mode* (see Section 4.2.2) it simply collects statistics about the task execution while keeping all p-gates closed. In *execution mode* (see Section 4.2.4) it runs the coscheduling algorithm and dynamically controls the p-gates. This solution moves as much computation as possible in hardware on the FPGA, thus minimizing the overall CPU overhead.

4.2 I/O traffic and Tasks Coscheduling

As aforementioned, there are two types of approaches for handling the I/O traffic interference on task execution. The first one is to estimate the worst-case I/O interference. This estimation requires as the inputs the cache access function and the I/O load function. If these functions are coarse the estimation will be very pessimistic. However, obtaining a precise cache access function is very hard. Both running the task on real hardware and using static analysis [39] only provides imprecise information, i.e. number of cache misses in an interval. In this research we propose an alternative solution: we use a combination of the worst-case interference analysis with the regulation of I/O traffic. The end result is a coscheduling algorithm that can efficiently guarantee tasks' deadlines while maximizing the I/O throughput. In our solution, we assume that at compile time, a control flow graph for the task can be derived comprised of a series of S *superblocks* $\{s_1, \ldots, s_S\}$. Each superblock can include branches and loops, but superblocks must be executed in sequence. For each s_i, we measure the worst case execution time $wcet_i$ without peripheral interference and the worst case number of cache misses CM_i, either through static analysis or making use of CPU self-measures. We can then obtain a safe bound on task delay in the following way: for each superblock s_i, we consider the worst case pattern of CM_i cache misses in an interval of length $wcet_i$, i.e. the pattern that results in the highest possible delay. We will then use $wcet_i$ and CM_i to derive adaptive algorithms which decide when to open the p-gate to maximize the I/O throughput. In the followings, we first detail how to obtain the described measurement in a concrete setting and then we provide our algorithms.

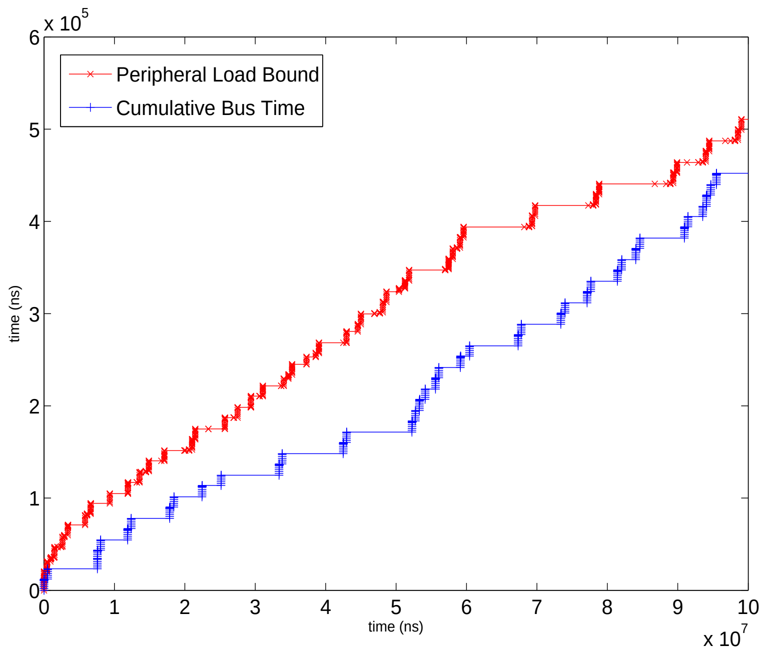


Figure 4.3: Measured Peripheral Load.

4.2.1 Peripheral Load Evaluation

The peripheral load function can be obtained in two ways. If the peripheral is an I/O interface and the node is part of a distributed system using a real-time communication protocol, then a bound on the peripheral activity can be derived analytically. Otherwise, we propose a following testing methodology for systems where analytical analysis is not available[1]. A trace of activity for a PCI/PCI-X peripheral can be gathered monitoring the bus with a logic analyzer. For example, Figure 4.3 shows the first 100ms of a measured trace for a 100Mb/s Ethernet network card in term of cumulative time taken by peripheral transactions on a 32bit, 33Mhz PCI bus segment; the whole recorded trace consisted of 1000 transactions. We developed a simple algorithm that computes the peripheral load function $E(t)$ from a trace in quadratic time in the number of bus transactions. The algorithm performs a double iteration over all transactions, computing at each step the amount of peripheral traffic in an interval between the beginning of any transaction and the end of any other successive transaction. The computed values are inserted into a list ordered by interval length, and all non maximal values are culled. Figure 4.3 shows the resulting $E(t)$ function in the interval

[1]While testing can fail to reveal the real worst case, we argue that it is nevertheless an accepted and commonly used methodology in the industry.

$[0, 100ms]$. If multiple traces are recorded, then an upper bound can be computed by merging the computed load functions for each trace and again removing all non maximal values. Finally, note that the computed $E(t)$ expresses a load bound for the bus segment on which the peripheral is located. In the case of the PCI/PCI-X architecture, the segment is connected to the memory controller through one or multiple bridges, each of which has buffering capabilities. In this case, a safe bound on the generated memory controller load can be obtained by summing a factor B/C to the computed $E(t)$ function, where B is the sum of the sizes of all traversed buffers and C is the speed of the memory controller.

4.2.2 Cache Miss Measurement

We devised a testing methodology to experimentally obtain the worst case execution time (WCET) and worst case number of cache misses for each superblock. Our implementation uses the Intel Core Microarchitecture architectural performance counters [17], but other CPU architectures such as IBM PowerPC provide similar support for CPU self-measures. Support was added by modifying the Linux/RK kernel [33]. The Core Microarchitecture specifies support for three architectural performance counters, each of which can be configured to count a variety of internal events. In particular, we used Counter 0 to count the number of elapsed CPU clock cycles and Counter 1 to count the number of level-2 cache misses. To accurately measure task execution without the effects of OS overhead, we configured both counters to be active only when the CPU is executing in user mode. Finally, we allowed reading the counter values from user mode with the `rdpmc` instruction (the counters can still be written and configured only in kernel mode) to reduce measurement overhead.

Counters are read inside each task by adding the checkpoint code in Figure 4.4 at the end of each superblock. The `cpuid` instruction inserts a synchronization barrier, i.e. it makes sure that all instructions fetched before `cpuid` are completed before the counters are read; this is required to cope with out-of-order execution. The counter values are then sent to the reservation controller running in data collection mode; this ensures that no write to system memory is performed at the checkpoint, which could cause an additional cache miss. The reservation controller determines the execution time and number of cache misses in each superblock computing the difference between the values obtained in successive checkpoints. After the task has finished, the computed values are read back from the reservation controller and $wcet_i$ and CM_i can be determined as the worst case over several task runs. Note that performance counters are not task-specific, so we had to modify the kernel to support reading the counters in a multitasking environment. We added

```
cpuid; //synchronization barrier
mov ECX, 0000 0000H;
rdpmc; //read Counter 0
//move value from DL:EAX to reservation controller
mov [RESCON_COUNTER0_H], DL;
mov [RESCON_COUNTER0_L], EAX;
mov ECX, 0000 0001H;
rdpmc; //read Counter 1
mov [RESCON_COUNTER1_H], DL;
mov [RESCON_COUNTER1_L], EAX;
```

Figure 4.4: Checkpoint Assembler Code.

two new fields to the task descriptor, `counter_extime` and `counter_cmisses`, to store the counter information. When a task is created, the fields are set to zero. When a task is preempted, the kernel first reads the counter values and saves them in the preempted task's descriptor. Then, it writes the values in the preempting task's descriptor back in the counters. Finally, the kernel writes the id of the preempting task in a register of the reservation controller, so that the controller can correctly associate the received information with the running task.

We implemented a compiler pass using the LLVM compiler infrastructure [21] to automatically add checkpoint code to the task. In the current implementation, the designer must manually identify the superblock boundaries selecting an initial and final basic block for each superblock. The choice involves a tradeoff, as smaller superblocks provide better information and tighter wcet bounds but at the price of increased measurement overhead.

4.2.3 I/O-inflicted Delay Analysis

Given load function $E(t)$ and values $wcet_i, CM_i$ for superblock s_i, we use the analysis provided in [35] to determine the worst case delay $D(s_i)$ suffered by all cache misses in s_i. The key idea is that a worst case pattern can be produced by "spreading out" the cache misses over the length $wcet_i$ of the superblock.

4.2.4 Coscheduling Algorithm

It is important to note that even when the analysis in [35] is tight, it is rare that at run-time a task will suffer a delay equal to the bound: a particular pattern for both cache misses and peripheral transactions is required to produce the worst case. As such, accounting for the worst case delay inflicted by all peripherals in the computation time budget of each task can lead to a large waste of resources. Motivated by this observation, we propose an alternative solution based on a run-time adaptive algorithm which leverages on our described architecture, composed of software checkpointing, peripheral gates and reservation controller. The idea is to assign to a task the minimal time budget of $\sum_{1 \leq i \leq S} wcet_i$, and then to monitor the actual execution of the task and open the p-gates whenever possible. At run-time, information on the execution time consumed by the current job is sent to the reservation controller at each checkpoint. The controller uses this information to determine the actual execution time e_i of superblock s_i for the current job. The *accumulated slack* time after superblock s_i can then be computed as $\sum_{1 \leq j \leq i}(wcet_j - e_j)$; the slack time is the maximum delay that the task can suffer while still meeting its computation time budget. We can then design a coscheduling algorithm that strives to maximize the amount of time that the p-gates are opened under the constraint that the slack can never become negative. Our proposal is to integrate both the analysis and coscheduling techniques in a mixed-criticality system. Inspired by the avionic domain, we consider two types of guaranteed real-time tasks: *safety critical* tasks like flying control, that have stringent delay and verification requirements, and *mission critical* tasks that are still hard real-time but have lower criticality. We propose to schedule safety critical tasks blocking all I/O traffic except the one from peripherals used by the task, and to account the delay in the time budget. For *mission critical* tasks we instead use coscheduling. Finally, we also assume that the system runs best effort or soft real-time tasks for which all p-gates are opened.

Algorithm 9 is our main coscheduling heuristic. For simplicity, we describe the algorithm for a single controlled task and a single peripheral, but it can be easily extended to a multitasking environment with multiple p-gates. Communication from the task to the reservation controller triggers the algorithm at the beginning of each job and at each checkpoint. The algorithm maintains two variables: `i` is the index of the last executed superblock, and `slack` represents the accumulated slack. At the end of each superblock s_i, the algorithm first recomputes the slack and then performs a check: if the slack is at least equal to the maximum delay $D(s_{i+1})$, then the p-gate is opened because we are sure that the slack will be non negative after the next superblock s_{i+1} is executed. Otherwise, the p-gate is kept closed.

The limitation of Algorithm 9 is that it greedily "allocates" all slack to the next superblock by immedi-

Algorithm 9 Adaptive Algorithm

```
JobStart() {
$slack := 0$
$i := 0$
CloseGate()
}
Checkpoint($e$) {
$i := i + 1$
$slack := slack + wcet_i - e$
if $D(s_{i+1}) \leq slack$ then
    OpenGate()
else
    CloseGate()
}
```

ately opening the p-gate. This can lead to a suboptimal allocation, as superblock s_{i+1} could be short and have a lot of cache misses while superblock s_{i+2} could be longer with very few cache misses. If we have additional information on the task, we can potentially do better using a predictive heuristic. In particular, Algorithm 10 assumes that the average case computation time avg_i and average case delay $D^{avg}(s_i)$ for each superblock s_i is known. The algorithm keeps track of the *predicted slack*, i.e. the total slack assuming that all future superblocks will execute for $e_j = avg_j$. We can then compute a strategy that maximizes the amount of time that the p-gate is opened by allocating the predicted slack among all future superblocks: if we decide to open the p-gate during s_j, we consume an amount of slack equal to $D^{avg}(s_j)$ and the p-gate is opened for $avg_j + D^{avg}(s_j)$ time units. It is easy to see that this allocation problem is equivalent to the KNAPSACK problem [18], which is well known to be NP-hard. We therefore use a sub-optimal polynomial time greedy solver: off-line, we order all superblocks by non-increasing values of $\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)}$. At run-time, we perform the allocation by iterating through the list ignoring all superblocks already executed. When the iteration arrives to the next superblock s_{i+1}, the p-gate is opened if the remaining predicted slack is greater or equal than $D^{avg}(s_{i+1})$.

Note that Algorithm 10 is not the only possible predictive algorithm; in fact, no on-line algorithm can be optimal, since any optimal algorithm must known exactly the computation time of future superblocks, i.e.

Algorithm 10 Predictive Algorithm

```
JobStart() {
$slack := 0$
$pslack := \sum_{k=1}^{S}(wcet_k - avg_k)$
$i := 0$
CloseGate()
}
Checkpoint($e$) {
$i := i + 1$
$slack := slack + wcet_i - e$
$tmp := pslack := pslack + avg_i - e$
**for all** $k$ in ORDERED_LIST $(\frac{avg_j + D^{avg}(s_j)}{D^{avg}(s_j)})$ **do**
  **if** $k > i + 1 \wedge D^{avg}(s_k) \leq tmp$ **then**
    $tmp := tmp - D^{avg}(s_k)$
  **if** $k == i + 1$ **then**
    **if** $D(s_{i+1}) \leq slack \wedge D^{avg}(s_{i+1}) \leq tmp$ **then**
      OpenGate()
    **else**
      CloseGate()
    **return**
}
```

it must be clairvoyant. However, for the sake of comparison it is interesting to compute an upper bound on the best possible performance of any on-line predictive algorithm. Assume that for a specific run, e_i is the execution time of s_i assuming that the p-gate is closed, and $\bar{e}_i$ is the execution time assuming that the p-gate is opened. An upper bound can be computed by solving the following integer linear programming problem:

$$\max \sum_{i=1}^{S} x_i \bar{e}_i \tag{4.2.1}$$

$$\forall i, 1 \leq i \leq S : x_i D(s_i) \leq \sum_{j=1}^{i-1} (wcet_j - (1 - x_j)e_j - x_j \bar{e}_j) \tag{4.2.2}$$

$$\forall i, 1 \leq i \leq S : x_i \in \{0, 1\}, \tag{4.2.3}$$

where $\{x_1, \ldots, x_S\}$ are indicator variables (i.e., $x_i = 1$ if the p-gate is opened during s_i). Equation 4.2.1 maximizes the open time, while Equation 4.2.2 expresses the slack constraint.

4.3 Experimental Results

To validate our architecture, we performed experiments on a platform comprised of an Intel Core2 CPU and an Intel 975X system controller. Using a PC platform allowed us easy access to all PC slots; however, to derive meaningful measures we changed the memory controller clock frequency obtaining a speed of 900Mhz for the CPU and a theoretical bandwidth of 2.4Gbyte/s for the memory controller, which is in line with typical values for embedded platforms.

We first performed an experiment to evaluate the maximum delay incurred by a task due to peripheral interference. To obtain repeatable measures, we implemented a custom traffic generator peripheral for the PCI-X bus based on a Xilinx ML455 board. The peripheral periodically initiates write transactions to main memory, and both the period and transactions length can be configured to produce a load up to the maximum of 1 Gbyte/s supported by PCI-X. We then designed a task to maximize cache stall time. The task allocates a memory buffer of double the size of the CPU level 2 cache, and then cyclically reads from the buffer, one word for each 128-byte cache line; a cache miss is thus generated for each memory read. We first ran the task without using the traffic generator and measured an execution time of 48.73ms and 580,227 cache misses. Using a memory benchmark, we evaluated a main memory throughput of $C = 1.8$Gbyte/s, which is slightly lower than the theoretical memory controller speed. Since 128 bytes must be transferred for each cache miss, the task is actually stalled for $\frac{580{,}227 \cdot 128}{1.8e9} = 41.26$ms, resulting in the desired high cache stall

SLACKONLY	ADAPTIVE	PREDICTIVE	BOUND
4.89%	31.21%	36.65%	40.85%

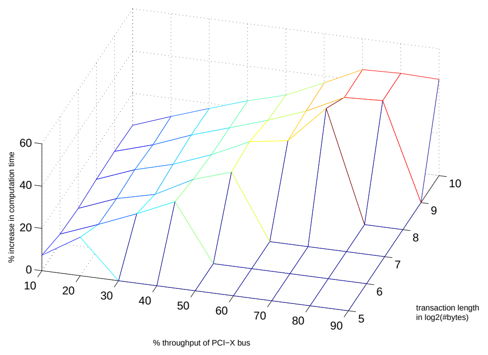
Table 4.1: Benchmark Results.

time of 84.67%. We then ran the task again, enabling the traffic generator with maximum load, and we measured an average increase in computation time of 43.85%.

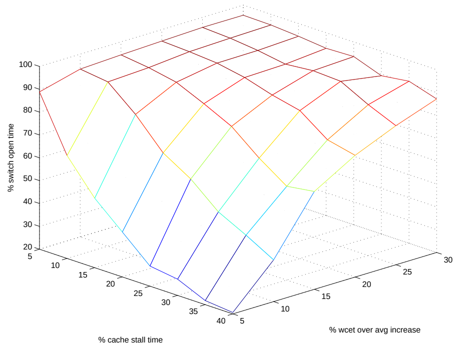
It is important to note that the measured value is an average case delay, since obtaining the worst case pattern over more than 500,000 fetches is improbable. Our analysis is able to compute the worst case delay, but we need additional information on the system controller, in particular the type of arbitration used by the memory controller and the maximum length values L and L'. These data are typically available for components used in embedded systems, see [29] for example. However, in the PC market manufacturers are often wary of revealing precise details for fear of losing competitive edge. We therefore used the following experimental methodology to obtain the required information: we first guessed values for L, L' and the arbitration type, and we built a simulator to predict average case delay for a variety of settings. We then performed extensive experiments and confronted the measured values with the predictions to determine if our guesses were acceptable.

Experimental results are shown in Figure 4.5, where each point is an average over 5 runs. We measured the percentage increase in computation time for the aforementioned task varying the offered peripheral load and length of peripheral transactions. Note that for small lengths we are not able to significantly load the bus, as the period of the traffic generator is constrained by PCI-related overhead; hence, some points in Figure 4.5(a) can not be generated on the bus and we show them as zero values. All results are within 5% of our predictions, assuming round-robin arbitration and $L = L' = 32\text{bytes}/C = 17.8$ns, which means that each fetch is broken down into 4 data transfers on the memory controller. Unexpectedly, we found that delay is constant over 70% load. Investigation of the PCI-X bus using a logic analyzer revealed that it is an issue of the PCI bridge, which is not fast enough to buffer all peripheral data. We also performed additional experiments varying the cache stall time of the task; this can be achieved by inserting a variable number of instructions between each successive cache line read. The obtained wcet increases also matched our simulation results within a small deviation.

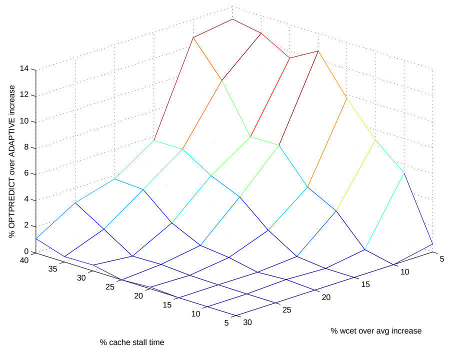
We then evaluated the performance of the described coscheduling algorithms on our platform. We chose a MPEG decoder [11] as our benchmark for two reasons: it is both a memory and I/O intensive application,



(a) ***Measured Delay.***



(b) ***Synthetic Tasks, ADAPTIVE.***



(c) ***Synthetic Tasks, algorithm ratio.***

Figure 4.5: Experimental Results.

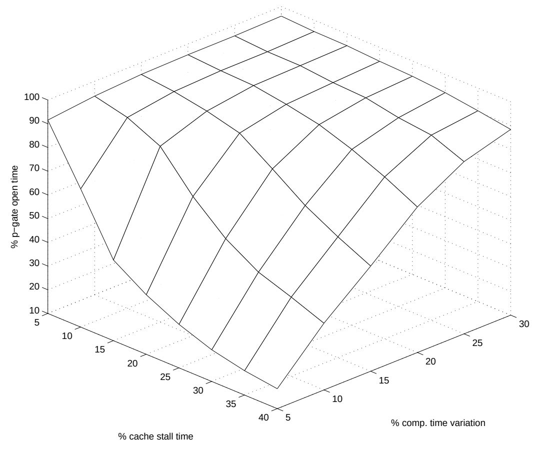


Figure 4.6: Synthetic Tasks, ADAPTIVE, $\sigma = 0.1$

and it is representative of the type of video computation that is becoming increasingly important for mission control in avionic systems. We collected average and worst case statistics on a test video clip after placing multiple checkpoints for each frame; the MPEG decoder is run as a periodic task, with 20 superblocks in each period. To mirror the behavior of a real application and increase the number of cache misses, we also ran a higher priority task that preempts the MPEG decoder every 1ms and replaces its cache content. Results averaged over 50 runs are shown in Table 4.1 in term of the percentage of time that the p-gate is opened in the task period. In the table, SLACKONLY represents a baseline solution where the p-gate is kept closed while the task is executing and is opened after the task has finished for its remaining time budget $\sum_{1 \leq i \leq S}(wcet_i - e_i)$; ADAPTIVE is Algorithm 9; PREDICTIVE is Algorithm 10; BOUND is the bound computed by solving the ILP problem of Equations 4.2.1-4.2.3. Note that since BOUND is not implementable at run-time, we computed the bound offline using measured values of computation times and number of cache misses. We can see that SLACKONLY tends to perform very poorly; ADAPTIVE is within 30% of BOUND and PREDICTIVE is roughly in between the two, which seems to suggest that prediction offers limited improvement.

To check whether the obtained results hold for more general settings, we also performed extensive simulations on synthetic tasks, each composed of 20 superblocks, varying three parameters σ, α, β. For each task and each superblock s_i, we first generated the average computation time avg_i from a uniform distribution with constant mean and coefficient of variation σ, and the average cache stall time $stall_i$ from a

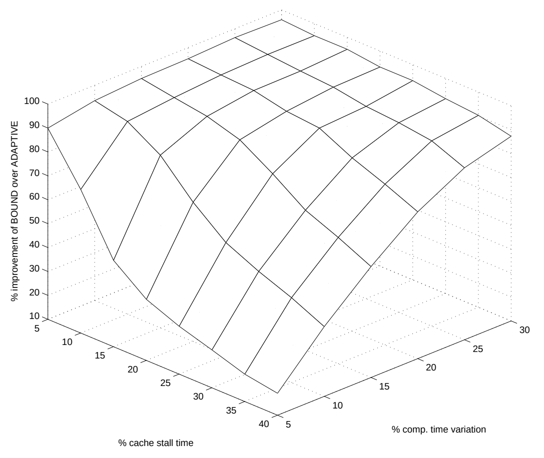


Figure 4.7: Synthetic Tasks, ADAPTIVE, $\sigma = 0.2$

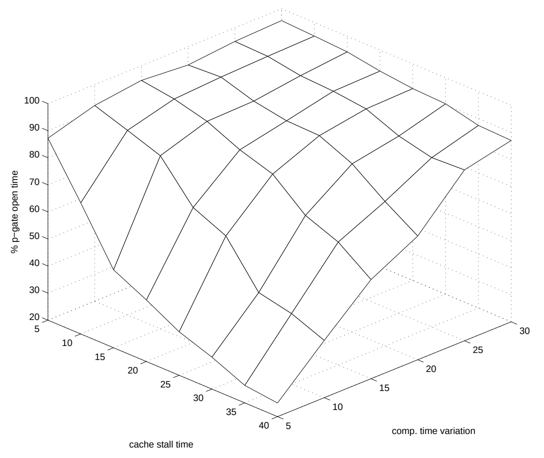


Figure 4.8: Synthetic Tasks, ADAPTIVE, $\sigma = 0.4$

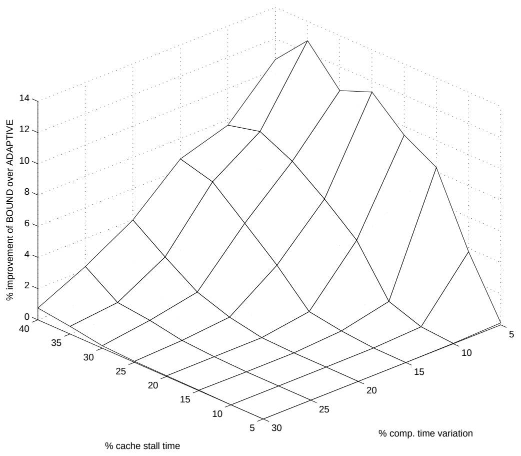


Figure 4.9: Synthetic Tasks, algorithm ratio, $\sigma = 0.1$

uniform distribution with mean β and coefficient of variation σ. We then simulated 10 task runs by extracting for each run and each superblock a computation time $e_i = avg_i(1 + \bar{\alpha})$ and a number of cache misses equal to $stall_i(1 + \bar{\alpha})$, where $\bar{\alpha}$ is extracted from a uniform distribution with mean 0 and maximum value α. Note that this implies $wcet_i = avg_i(1 + \alpha)$, i.e. α is the increase in computation time between the average and the worst case.

We focus on results for the ADAPTIVE and BOUND cases: Figure 4.6, 4.7, 4.8 show the value of ADAPTIVE for values of σ equal to $0.1, 0.2, 0.4$ respectively. Figure 4.9, 4.7 4.11 shows the competitive ratio of $\frac{\text{BOUND}-\text{ADAPTIVE}}{\text{ADAPTIVE}}$, again for $\sigma = 0.1, 0.2, 0.4$. All points are averages over 10 tasks (100 runs total of the simulator). We varied the average cache stall time β between $[0.05, 0.4]$ and the computation time variation α between $[0.05, 0.3]$; axis direction is inverted between the two sets of figures for easier visualization. Note that the definition of α implies SLACKONLY=$\frac{\alpha}{1+\alpha}$ for all cases. First note that the obtained results are very close for the different values of σ, which seems to indicate that none of the tested algorithm is very sensitive to variations in superblock size. The second main observation is that the performance of the algorithms depends on the difference between α and β. For $\alpha > \beta$, both algorithms can open the p-gate almost all the time because the high wcet variability forces us to over-provision the computation time budget of the task; however, note that a coscheduling algorithm is still needed to guarantee safety, as there are times where the p-gate must be closed to ensure that the task meets its deadline. In the case $\alpha < \beta$, which is representative of more predictable, but memory intensive real-time tasks, the fraction of time the p-gate can

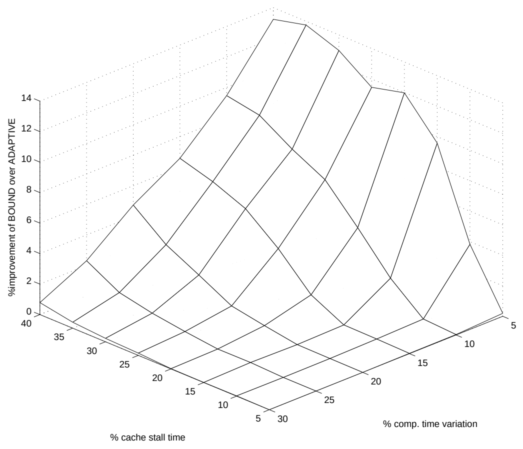


Figure 4.10: Synthetic Tasks, algorithm ratio, $\sigma = 0.2$

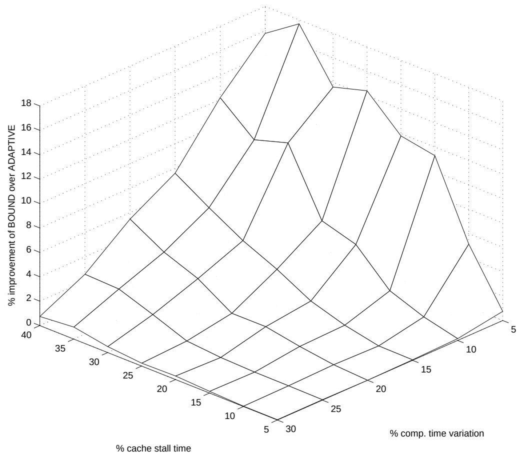


Figure 4.11: Synthetic Tasks, algorithm ratio, $\sigma = 0.4$

be opened decreases as the delay $D(s_i)$ becomes significant compared to $wcet_i - avg_i$. The performance of ADAPTIVE degrades more rapidly than BOUND, but it remains with a competitive ratio of 18%, which compares even more favorably than the MPEG case.

4.4 Related Work

Apart from [36], to the best of our knowledge the only works that study the impact of I/O load on real-time scheduling are [42] and [16]. [42] uses a PCI-based testbed similar to ours, but its empirical approach can not derive safe wcet bounds. [16] uses an analytical approach but it assumes highly predictable cycle-stealing bus arbitration, which is not true of commodity systems.

Two other research areas are related to our work. First of all, peripheral activities impose an additional overhead on the CPU: device driver execution. Techniques to account for such overhead have been described in [23, 45] for network cards and hard disks based on experimentally-derived bounds. Second, there is a second potential source of interference at either the cache or the memory controller level: other CPUs. A methodology to compute cache access delay in multiprocessor systems has been proposed in [41] based on static analysis. However, we argue that this problem domain is essentially different from ours because synchronizing task schedule across multiple processors is easier than synchronizing CPU and peripheral execution.

4.5 Conclusions

The effects of the conflicts in accessing memory between I/O devices and processors can not be ignored: our experiments using typical settings for an embedded system reveal that interference at the memory controller can increase the computation time of a task by almost half. In this research, we proposed a hardware and software system to cope with this effect. The system is used to control peripheral activities using a peripheral gate and a coscheduling algorithm that dynamically allows/disallows peripherals to transmit making sure that the task computation time does not exceed its WCET without traffic. Our experiments show that even a simple adaptive coscheduling heuristic can greatly improve the amount of allowed traffic compared to the baseline approach of blocking all peripherals while the task is executing. More complex predictive heuristics can do even better, but our experiments revealed that the improvement space is somehow limited.

Chapter 5

Conclusion

In this research, we have investigated on the issue of guaranteeing task predictable execution in the current computer architecture with focus on the issue involving cache, memory controller and Network-on-Chip. These are supportive components that may inflict significant unpredictability in tasks' execution time but have not received adequate attention from previous researches. Each of these components has its own characteristics and affect task execution in different ways. For this reason, we have analyzed each component separately and proposed suitable solutions that we believe to be both practical and sound in theory.

As discussed in Chapter 2, uncontrolled use of cache may cause significant increase in tasks' worst-case execution time. In practice, however, this increase is unnecessary and can be avoided if cache sharing is properly managed. Our proposed cache-partitioning optimization technique strives to eliminate inter-task cache interference when possible. As the result, it significantly improves system utilization. Although, our technique is designed for single-core professors, it can also be used on multi-core processors whose cores share a cache memory because the technique partitions cache on task basis. However, on multi-core processors which has no inter-core shared cache, a cache partition algorithm should be designed together with a task allocation algorithm. Since in these processors, cache partition and task allocation are inter-dependent. This problem is left open for future works.

The new resource sharing paradigm appearing in NoC has opened up a challenging scheduling problem. In particular, the non-transitive dependence between transactions in NoC can cause the existing scheduling algorithms inefficient. By taking into account this new dependence pattern, we proposed a novel scheduling framework that, under practical conditions, is able to guarantee transaction deadlines with very high NoC utilization. The novelty of our solutions comes from our use of graph theory tools to take advantage of the topological relationship between transactions. Although our current proposal can only work on some popular transaction topologies, I believe it has provided deep insights on the problem at hand. We recognize, however, that more works need to be done in order to make the framework applicable to all cases. The general solution may be in the form of an approximation algorithm that borrows the insights from this research. Another interesting future direction is to apply our framework to the scheduling of real-time tasks on parallel-computing systems. A task running on these systems may has the same resource usage pattern as a transaction on NoC, i.e a task may use multiple processors simultaneously.

Like with cache, main memory sharing can also be the source of unpredictability in task execution. Accounting for this unpredictability by worst-case analysis alone results in system that is significantly underutilized. To tackle this issue, we proposed a software and hardware system and a co-scheduling technique which are used to regulate the peripheral traffic to the main memory such that tasks are guaranteed to not execute longer than their run-alone WCET.

I believe that the research in this dissertation has addressed some of the key issues in guaranteeing real-time task execution predictability and has made important contribution to real-time scheduling theory. With the continuing advance in computer architecture technology, building safe and sound real-time systems will continue to demand significant research efforts. I hope this dissertation will be considered as an important building block of these ongoing efforts.

Bibliography

[1] Aeronautical Radio Inc. *ARINC 653 Specification*. http://www.arinc.com/.

[2] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[3] Shobana Balakrishnan and Füsun Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(7):664–678, 1998.

[4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 345–354, New York, NY, USA, 1993. ACM.

[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2), 2005.

[6] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 101–110, Washington, DC, USA, 2008. IEEE Computer Society.

[7] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation: A performance view. IBM Research, 2005.

[8] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:101–110, 2006.

[9] S. Dropsho. Comparing caching techniques for multitasking real-time systems. Technical Report UM-CS-1997-065, Amherst, MA, USA, 1997.

[10] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *IEEE International Solid-State Circuits Conference*, 2010.

[11] FFMPEG project. *libavcodec multimedia library*. `http://ffmpeg.mplayerhq.hu/`.

[12] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, 2005.

[13] S. Gopalakrishnan, L. Sha, and M. Caccamo. Hard real-time communication in bus-based networks. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 405–414, Washington, DC, USA, 2004. IEEE Computer Society.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization. WWC-4. IEEE International Workshop*, 2001.

[15] P. Holman and J. H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4):543–564, 2005.

[16] Tay-Yi Huang, Jane W. S. Liu, and Jen-Yao Chung. Allowing cycle-stealing direct memory access I/O concurrent with hard-real-time programs. In *Int. Conf. on Parallel and Distributed Systems*, Tokyo, 1996.

[17] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, February 2008.

[18] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[19] D. B. Kirk and J. K. Strosnider. Smart (strategic memory allocation for real-time) cache design using the mips r3000. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990.

[20] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[22] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *SIGMETRICS Perform. Eval. Rev.*, 14(1):44–53, 1986.

[23] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability: Study of a network driver. In *Proceedings of the* 13^{th} *IEEE Real Time Application Symposium*, Apr 2007.

[24] J. Li and M. W. Mutka. Real-time virtual channel flow control. *J. Parallel Distrib. Comput.*, 32(1):49–65, 1996.

[25] J. Liedtke, H. Hartig, and M. Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[26] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[27] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, 1990.

[28] Z. Lu, A. Jantsch, and I. Sander. Feasibility analysis of messages for on-chip networks using wormhole routing. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 960–964, New York, NY, USA, 2005. ACM.

[29] Marvell. *Discovery II PowerPC System Controller MV64360 Specifications.* `http://www.marvell.com/`.

[30] F. Mueller. Compiler support for software-based cache partitioning. In *Proceedings of the ACM Workshop on Languages, Compilers, and Tools for Real-Time Systems*, 1995.

[31] M. D. Natale and A. Meschi. Scheduling messages with earliest deadline techniques. *Real-Time Syst.*, 20(3):255–285, 2001.

[32] S. Oikawa and R. Rajkumar. Linux/rk: A portable resource kernel in linux. In *Proceedings of the 19th IEEE Real-Time Systems Sumposium*, 1998.

[33] S. Oikawa and R. Rajkumar. Linux/RK: a portable resource kernel in linux. In *Proceedings of the* 19^{th} *IEEE Real-Time System Symposium*, Madrid, Spain, December 1998.

[34] PCI SIG. *Conventional PCI 3.0, PCI-X 2.0 and PCI-E 2.0 Specifications*. `http://www.pcisig.com`.

[35] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008.

[36] R. Pellizzoni and M. Caccamo. Towards the predictable integration of real-time COTS based systems. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, Dec 2007.

[37] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, 1998.

[38] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, 2005.

[39] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[40] IBM Research. Cell BE programming tutorial. IBM, 2007.

[41] J. Rosen, P. Eles A. Andrei, and Z. Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of the* 28^{th} *IEEE Real-Time System Symposium*, December 2007.

[42] S. Schönberg. Impact of pci-bus load on applications in a pc architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec 2003.

[43] Z. Shi and A. Burns. Priority assignment for real-time wormhole communication in on-chip networks. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 421–430, Washington, DC, USA, 2008. IEEE Computer Society.

[44] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society.

[45] M. Stanovich, T. Baker, and A. Wang. Throttling on-disk schedulers to meet soft-real-time requirements. In *Proc. of the* 14^{th} *IEEE RTAS*, St. Louis, Missouri, April 2008.

[46] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9:147–171, 1995.

[47] D. Wiklund and D. Liu. Design mapping, and simulations of a 3G WCDMA/FDD basestation using Network-on-Chip. *System-on-Chip for Real-Time Applications, International Workshop on*, 0:252–256, 2005.

[48] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution time problem - overview of methods and survey of tools. Technical report, 2007.

[49] J. L. Wolf, H. S. Stone, and D. Thiébaut. Synthetic traces for trace-driven simulation of cache memories. *IEEE Trans. Comput.*, 41(4):388–410, 1992.

[50] A. Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.

[51] M. Zbigniew and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, December 2004.

[52] D. Zhu, D. Mosse, and R. Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary. In *24th IEEE International Real-Time Systems Symposium*, 2003.