Trading End-to-End Latency for Composability *

Slobodan Matic
UC Berkeley
matic@eecs.berkeley.edu

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

Abstract

The periodic resource model for hierarchical, compositional scheduling abstracts task groups by resource requirements. We study this model in the presence of dataflow constraints between the tasks within a group (intragroup dependencies), and between tasks in different groups (intergroup dependencies). We consider two natural semantics for dataflow constraints, namely, RTW (Real-Time Workshop) semantics and LET (logical execution time) semantics. We show that while RTW semantics offers better end-to-end latency on the task group level, LET semantics allows tighter resource bounds in the abstraction hierarchy and therefore provides better composability properties. This result holds both for intragroup and intergroup dependencies, as well as for shared and for distributed resources.

1. Introduction

As the number of applications that share the same resources increases, the integration of software components in real-time or embedded systems becomes more pertinent. A key challenge is to have real-time assurance and, at the same time, a high degree of flexibility in system design and maintenance. This problem is addressed within an *open* real-time system [5] that consists of mutually *independent* components with sets of tasks of different time criticality. Research in open systems concentrates on partitioning and scheduling schemes that make both the implementation and temporal behavior of a component independent of the presence of other components in the system. However, embedded real-time systems, e.g., automotive [11] or aircraft [19] systems, are often put together from several *interacting* software components corresponding to different control loops. The problem is even more demanding when components have to be implemented by different suppliers [8].

In an open system, schedulability analysis and the admission test for a task group cannot depend on the properties of any other task group in the system. In recent years work in the composition for open systems has shifted towards *hierarchical* scheduling frameworks [15, 17, 18], which extend resource partitioning over multiple levels. In such a framework a resource is often allocated by a higher to a lower scheduling level through a *scheduling interface*. The interface specifies the resource requirement from the lower level and the resource guarantee from the higher-level scheduler. A hierarchical scheduling framework should exhibit *separation* among levels, i.e., the interface should be minimal. Moreover, the main benefits of hierarchical scheduling arise if the framework is *compositional*, i.e., if properties established at the lower also hold at the higher level.

Abstraction of the internal complexity of a task group into a single requirement can be used to ensure the favorable properties and to reduce scheduling difficulties in the hierarchical scheduling framework. Early work in task group abstraction [20, 13] considers the *periodic resource* model (T, C), a resource abstraction under which a component is guaranteed to get C units of the resource every T units of time. This research showed how to abstract a group of independent periodic tasks with EDF or RM scheduling algorithms into a single periodic task. The compositionality of the framework was demonstrated by combining multiple scheduling interfaces into a single higher-level interface. Whereas these initial efforts with the periodic resource model addressed only independent periodic tasks, more recent efforts considered sets of tasks with blocking synchronization operations [1, 14].

In this paper we study, under the same periodic resource model, hierarchies of tasks with *data dependencies*. Namely, we assume that all applications that execute on the considered resources are specified in the conventional periodic task model with an underlying task precedence graph. We first (Sec. 2) give two different application interpretations, i.e., we present two semantics, RTW and LET, which differ in the propagation of data between tasks. While RTW follows the semantics of real-time code generated from a Simulink environment [6], LET has been used in some domain-specific languages for control applications [9]. The RTW scheme transfers the output of a task as soon as the

*This research was supported by the AFOSR MURI grant F49620-00-1-0327 and by the NSF grants CCR-0225610 and CCR-0208875.

Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)
0-7695-2490-7/05 $20.00 © 2005 IEEE

task completes execution. The LET scheme makes the output of a task available at the prespecified time, namely, at the relative deadline defined by the task period. Compared to the RTW semantics, this typically increases the latency.

The composition with abstracted components inevitably incurs higher resource utilization than the component utilization sum. Therefore, effectiveness of composition can be compromised. If component abstraction is too coarse, only a few components can be correctly composed, and the rest may be disallowed on the admission test, even when actual required resource utilization is low. Therefore, we focus on *tight* abstractions, i.e., abstractions that minimize lower level resource requirements. We show that the tightness of abstractions, and therefore composability, depends on the application semantics. So, although at the lower levels the end-to-end latency is less for the RTW semantics, at the higher levels, when task group abstraction is taken into account, the LET semantics permits tighter abstractions.

We compare the composability of the two data transfer semantics in several scenarios. Sec. 3 studies the abstraction of a task group that executes on a single resource and with precedence constraints among tasks within the group (*intragroup* task precedences). We show that the tightness difference in favor of the LET semantics can come from the underlying scheduling algorithm used to implement a particular semantics. Sec. 4 generalizes the result for the case of a task group distributed over several resources. We characterize how large the gap in the tightness of abstractions between the two schemes, RTW and LET, can be. Moreover, we show that with LET semantics both abstraction and scheduling is simpler. This is important for hierarchical open systems, since complicated interaction between scheduling levels increases unpredictability in task execution. Finally, Sec. 5 studies higher levels of the hierarchical scheduling framework. In this context task precedences among different task groups are allowed (*intergroup* task precedences). The LET semantics again results in tighter and simpler abstractions. In addition, and contrary to the RTW semantics, the LET semantics enables a compositional framework with separation between levels.

2. Multirate Task Programs

Let $\mathbb{Q}$ be a finite set of numbers that are all multiples of a certain sufficiently small unit rational number, and let $\mathbb{R}$ be the set of real numbers. A *task* $t = (p, e)$ consists of a period $p \in \mathbb{Q}$ and a worst-case execution time requirement $e \in \mathbb{R}$. A *task graph* $G = (V, E)$ is a directed graph with a set of tasks V and a set of task data dependencies $E \subseteq V^2$. In general, a program may exhibit multirate behavior because each task is characterized by its own period. Therefore, the program is fully specified only with the semantics of data transfer between tasks. In this paper we assume that all task graph edges comply with the same semantics. In particular, we focus on two dataflow semantics, RTW and LET.

Real-Time Workshop semantics. Real-Time Workshop (RTW) [6] is a tool for automatic code generation in the MATLAB/Simulink environment. For a given task graph the tool generates multithreaded code, one thread per each sample time of the graph. The code is supposed to run on an RTOS that offers a priority based preemption mechanism. Each task is assigned to a thread based on its period, and the schedule within a thread is constructed from the task dependencies. The rate monotonic (RM) scheduler invokes the generated code, enabling preemption between rates.

To make multirate models operate correctly in real time, the program is implicitly modified by placing *rate transition* blocks, hold or delay blocks, between tasks that have unequal periods. The rate transition blocks are assumed to execute in negligible time. Consider the data dependency shown in Fig. $1(a)$, where the period of the data consumer task t_2 is a multiple of the period of the data producer task t_1. A problem of data integrity exists when the input to task t_2 changes after its execution starts. Also, the output is nondeterministic and depends on how late t_2 starts. Adding a hold block h ensures that the second invocation of t_1 does not overwrite the data. The hold block executes with the slower period of t_2, but with the higher priority of the faster task t_1. In that way, it executes before task t_2 and its output value is held constant while t_2 executes. Beside the schedule for the tasks, Fig. $1(b)$ shows the input signal i_2 of t_2 over time, assuming incrementing functionality of t_1.

In the inverse case shown in Fig. 2, the period of the data producer task t_1 is a multiple of the period of the data consumer task t_2. The delay rate transition block d compensates for the varying execution time of t_1, i.e., it makes the time of data transfer deterministic no matter how early t_1 completes. The delay block executes with the period of t_1, but with the higher priority, so that its output value is written before required invocations of t_2.

The RTW rate transition mechanism limits the set of syntactically correct programs. First, the period of each task must be an integer multiple of a base period (e.g., the smallest period). Second, each cycle of the task graph G must contain a dependency resolved with a delay block.

Logical Execution Time semantics. According to the logical execution time (LET) concurrency model defined in the scope of the Giotto programming language [9], each task has a *release* and a *termination* time: the release time specifies the exact time at which the task inputs are made available to the task; the termination time specifies when the task outputs become available to other tasks. The task must start running, may be preempted, and must complete execution during its LET, which is the time from release to termination. Thus the times when a task reads and writes

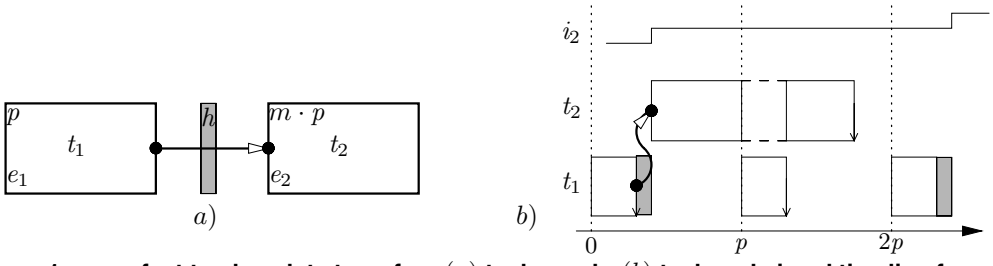

Figure 1. RTW**: fast to slow data transfer -** (a) **task graph;** (b) **task and signal timeline for** $m = 2$

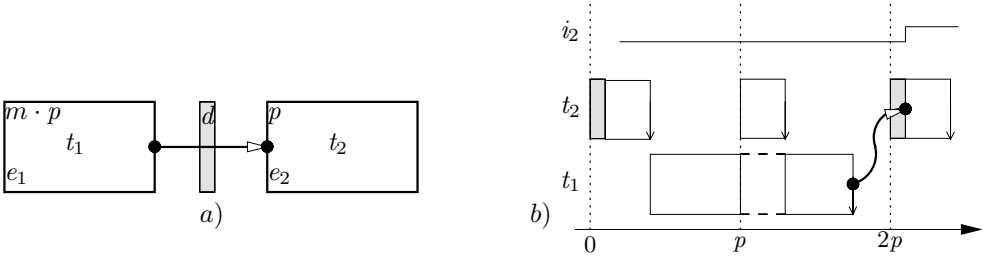

Figure 2. RTW**: slow to fast data transfer -** (a) **task graph;** (b) **task and signal timeline for** $m = 2$

data are decoupled from the task execution. For the periodic tasks that we consider in this paper, release and termination time instants of a task are equal to multiples of the task period. The LET model is an abstract programming model that does not prescribe any particular scheduling strategy. This is shown in Fig. $3(b)$ with dashed box throughout a period of a task. Of course, it must be ensured that the generated code satisfies the LET assumption.

For data precedences with LET semantics, a data transfer occurs at release/termination time instants, which abstracts away precedence constraints and makes tasks independently schedulable. Every LET precedence can be modeled as a sequence of a delay d and a hold h block as shown in Fig. $3(a)$. The delay block executes with the period of the data producer t_1 delaying its output until its termination time. The hold block executes with the period of the data consumer t_2 holding its input value during its LET. The LET semantics imposes no limitations on the program, i.e., task periods need not be harmonic and the task graph may be an arbitrary directed graph.

RTW versus LET. We end this section by comparing the two semantics with respect to latency and synchronization requirements. These properties will favor the RTW semantics. In the remaining sections we will compare the two semantics with respect to composability, which makes the LET semantics look better.

The end-to-end latency D of a sequence of n tasks t_j with task precedences (t_{j-1}, t_j) for each $j = 2, ..., n$, is the time between the release of task t_1 and the completion of task t_n. Unlike the RTW semantics, with the LET semantics each fast to slow precedence constraint increases the end-to-end latency by the period of the data producer task. In the worst case all task precedences in the sequence are such, i.e., let $p_j = m_j \cdot p_{j-1}$ for $j = 2, ..., n$ and $m_j \geq 1$. With the RTW semantics, the end-to-end latency D_{RTW} of the sequence is bounded by p_n. With the LET semantics, the latency D_{LET} is bounded by $p_1 + \cdots + p_n$. So, if all tasks have the same period, i.e., if $m_j = 1$, then the worst-case latency with the RTW semantics is n times smaller than the latency with the LET semantics.

Moreover, the latency of the sequence in the RTW case depends on the worst-case execution times of tasks and, as such, can be arbitrarily small. On the other hand, the latency in the LET case depends on the logical execution times of tasks, i.e., on the task periods, no matter how small actual execution times are. This difference is important for the case when a single task graph has a dedicated resource. However, in the case of a partitioned resource, as discussed in [20], the feasibility problem is more relevant than the latency minimization problem. Note that even the latency of the RTW can be reduced either by compromising determinism, or by more complicated synchronization or dataflow models [12, 7].

RTW and LET semantics differ also in their synchronization and memory requirements. Let task t_1 precede task t_2, and let $p_2 = m \cdot p_1$. In p_2 time units of RTW execution, the hold synchronization block is invoked only once. In the same time interval of LET execution, the hold block is invoked once and the delay block m times. However, both blocks represent data transfers and typically execute in negligible amounts of time.

The memory required for the execution of the program with n tasks and RTW semantics is bounded by n. The same program with LET semantics requires memory twice as large, since the output data must be stored even after a task completes. Moreover, this memory is used all the time during program execution, while in the RTW case memory needed for a task is used only during task execution. A more detailed study of memory requirements for a run-time system with the LET semantics is presented in [10].

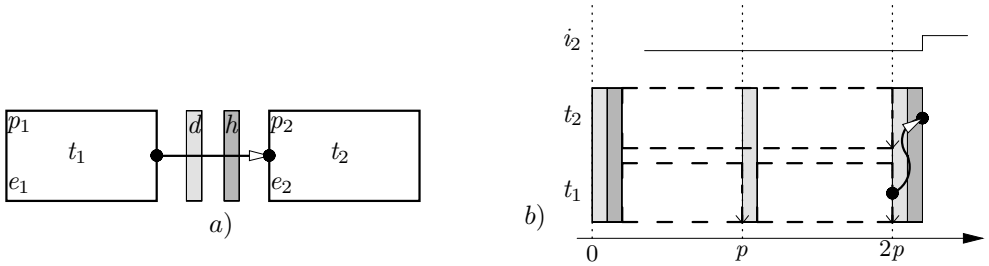

Figure 3. LET **data transfer -** (a) **task graph;** (b) **task and signal timeline for** $p_1 = p_2/2 = p$

3. Task Group Abstraction

3.1. Independent Task Set Abstraction

We first briefly present results from [20] for schedulability of a set of independent tasks under a periodic resource. A resource can be modeled as a *periodic resource* $R = (T, C)$ if it can guarantee allocations of at least C time units every T time units. The model does not specify how the guaranteed C time units are distributed over a time interval of size T. An *instance* of the periodic resource is any time trace of resource allocations that satisfies the guarantee (T, C). For a given periodic resource $R = (T, C)$, the resource *supply bound function* $\mathtt{sbf}_R : \mathbb{R} \rightarrow \mathbb{R}$ maps $\tau \in \mathbb{R}$ into the minimum supply of the resource R over all time intervals of size τ. For details on computing the supply bound function the reader is referred to [20]. As an example, Fig. 4 shows the supply bound function $\mathtt{sbf}_R$ for the periodic resource $R = (8, 7)$. Let V be the set of independent, periodic, and preemptive tasks. For a given set of tasks V, the resource *demand bound function* $\mathtt{dbf}_V : \mathbb{R} \rightarrow \mathbb{R}$ maps $\tau \in \mathbb{R}$ into the maximum resource demand over all time intervals of size τ. If the scheduling algorithm is earliest deadline first (EDF), we have $\mathtt{dbf}_V(\tau) = \sum_{t_i \in V} \lfloor \tau / p_i \rfloor \cdot e_i$, where $t_i = (p_i, e_i)$. For the rate monotonic scheduling algorithm (RM), the demand bound function is calculated for each task t_i as a cumulative resource demand of the task over an interval of time τ, i.e., $\mathtt{dbf}_V(\tau, t_i) = e_i + \sum_{t_k \in V(t_i)} \lceil \tau / p_k \rceil \cdot e_k$, where $V(t_i)$ is the set of tasks of higher priority than t_i.

We say that the scheduling model (V, T, C, A) is *schedulable* if under every instance of allocations of the periodic resource (T, C), there exists a feasible schedule for the task set V with the scheduling algorithm A. Theorems 1 and 2 in [20] give sufficient and necessary conditions for the schedulability of (V, T, C, A) with EDF or RM scheduling algorithms. Let $\mathtt{lcm}_V$ be the least common multiple of the periods of tasks in V. A scheduling model (V, T, C, EDF) is schedulable if and only if for all $0 < \tau \leq 2 \cdot \mathtt{lcm}_V$, the maximal resource demand is no greater than the minimum resource supply, i.e., $\mathtt{dbf}_V(\tau) \leq \mathtt{sbf}_{(T,C)}(\tau)$. For instance, if V is the set of three tasks $V = \{(24, 8), (8, 2), (16, 4)\}$, then Fig. 4 shows

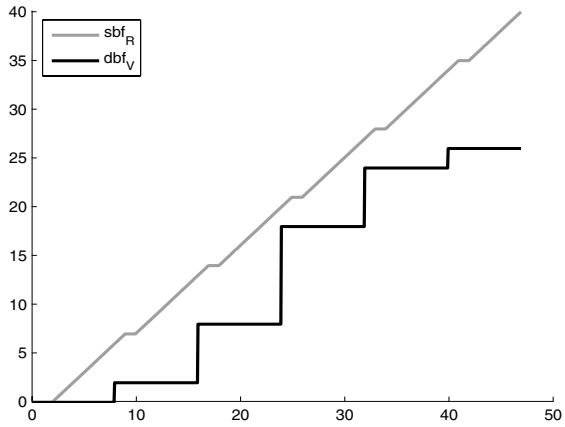

Figure 4. Supply and demand bound functions

the demand bound function $\mathtt{dbf}_V$, and also illustrates that (V, T, C, EDF) is schedulable if $(T, C) = (8, 7)$. A scheduling model (V, T, C, RM) is schedulable if and only if for all tasks $t_i \in V$, there exists $0 < \tau_i \leq p_i$ such that $\mathtt{dbf}_V(\tau_i, t_i) \leq \mathtt{sbf}_{(T,C)}(\tau_i)$.

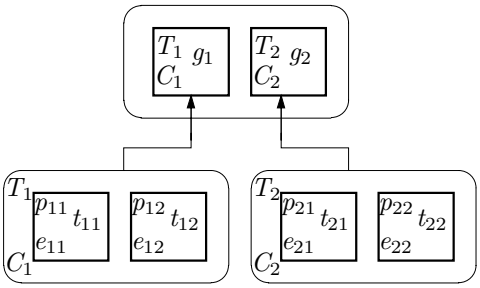

Figure 5. Hierarchical scheduling framework

In a hierarchical scheduling framework (Fig. 5), a separate scheduling problem is solved at each level of the hierarchy. If (V, T, C, A) is schedulable, then the set of independent periodic tasks V under resource (T, C) and algorithm A can be abstracted as a single periodic task (T, C). So, in a hierarchical scheduling framework, the higher-level scheduler allocates partitions for the set V as it was a periodic task (T, C). In figures we represent abstractions as rounded boxes, in this case characterized by (T, C) pairs.

3.2. Intragroup Task Precedence Abstraction

In this subsection we add precedences to a set of periodic tasks that execute on a single resource. Whereas here we consider a single group of tasks represented by a task graph, later we will discuss multiple, hierarchically structured task groups with precedences between them. We use the term "program" to capture tasks, constraints (timing and precedences), and the task graph dataflow semantics. Formally, a *program* (G, S) consists of a task graph $G = (V, E)$ and semantics $S \in \{\text{RTW}, \text{LET}\}$. For instance, the task graph shown in Fig. $6(a)$ is defined with $G_0 = (\{t_1, t_2, t_3\}, \{(t_1, t_2), (t_2, t_3)\})$.

Definition 1 (Program schedulability) *If under all instances of the periodic resource (T, C), there exists a schedule feasible for task graph G with semantics S the scheduling model (G, T, C, S) is* schedulable.

We assume that a child scheduler, when communicating resource requirements to its parent scheduler, provides not only a single pair (T, C), but a set of pairs, and, in particular, a function with the domain set $\mathbb{Q}$ that maps each period to an execution time requirement (capacity). Such a function enables a tighter hierarchy than a single pair, and also avoids computation of the optimal pair at the child scheduler (e.g., when the switching overhead is not known). We assume context switching time takes negligible time. This can be avoided by adding the appropriate overhead to task execution time.

Definition 2 (Program abstraction) *A function $c : \mathbb{Q} \rightarrow \mathbb{R}$* tightly abstracts *program (G, S) if c maps each period T into the smallest capacity C such that (G, T, C, S) is schedulable.*

If the function c tightly abstracts the program (G, S), then the *abstraction utilization* function $u : \mathbb{Q} \rightarrow \mathbb{R}$ of (G, S) maps each period T into $u(T) = c(T)/T$. In the rest of the paper, for two functions, f_1 and f_2, with arbitrary domain set A and range set $\mathbb{R}$, and for a relation $\sigma \in \{<, \leq, >, \geq\}$, we write $f_1 \ \sigma \ f_2$, if $f_1(a) \ \sigma \ f_2(a)$ for all $a \in A$. Note that the abstraction utilization function u satisfies $0 \leq u \leq 1$.

The results summarized in Sec. 3.1 cannot be used directly on task graphs because precedence constraints with semantics must be taken into account. As explained in Sec. 2, the RTW method uses fixed priority scheduling of tasks to maintain the order of task execution. The LET method is not restricted to any scheduling algorithm, and, in principle, its semantics can be implemented with the EDF algorithm where precedences are ignored. Thus, benefits of better schedulability, may in compositional scheduling frameworks be turned into tighter abstraction. However, if the EDF algorithm is not an option [3], some other, simpler scheduling algorithm might also give a tighter abstraction. For example, if the periods of all tasks have a common divisor d, then a simple round robin (RR) technique may be used. For each task (p_i, e_i), let $k_i = p_i/d$. A quantum of e_i/k_i time units is allocated to a task in each round. In this case the demand bound function for the RR scheduling algorithm is $\texttt{dbf}_V(\tau) = \sum_{t_i \in V} \lfloor \tau/p_i \cdot k_i \rfloor \cdot e_i/k_i$.

Example. Fig. 7 shows the functions that tightly abstract the program from Fig. $6(a)$ for different semantics, i.e., scheduling algorithms. As expected, the tightest abstraction is for the EDF scheduling algorithm, i.e., EDF for LET semantics. For this example, the RR algorithm for LET semantics gives the abstraction function that lies between EDF and RTW abstraction functions. In particular, for $T = 7.5$, the required capacities are $c_{\text{EDF}}(T) = 6.5$, $c_{\text{RR}}(T) = 6.83$, and $c_{\text{RTW}}(T) = 7.1$.

We performed simulations to evaluate the difference between the two semantics with respect to latency and composability properties. In all simulations we assumed a chain of tasks as the task graph. The size of the chain, i.e., the task workload size, was the parameter of the simulation. The periods of the tasks were randomly assigned from the range [1,20), and the task execution times were chosen such that total workload utilization was in the range [0.3,0.7]. For each pair of successive tasks in a task chain one period was the multiple of the other period, because of the limitation of the RTW dataflow model. For each workload size we ran simulations on 300 task graphs for both RTW and LET semantics (under EDF), and the relative difference, averaged over all task graphs, is shown in Fig. 8. The relative difference in end-to-end latency was calculated using the delay between the release of the first task and the worst-case completion time instant of the last task in the task chain. Under a shared periodic resource the delay for both semantics is determined by task periods. The relative difference in composability was calculated as the relative difference in the abstraction functions taken at the smallest period of chain tasks.

We now formalize the observed abstraction gap for a shared single resource, so that it can be compared with the distributed case in Sec. 4, and to use it as a base case for our hierarchical scheduling framework discussed in Sec. 5.

Lemma 1 *Let $G = (V, E)$ be a task graph and (T, C) a periodic resource. (1) If (G, T, C, S) is schedulable for $S = \{\text{RTW}, \text{LET}\}$, then (V, T, C, EDF) is schedulable. (2) If (V, T, C, EDF) is schedulable, then (G, T, C, LET) is schedulable.*

Proof. (1) If (G, T, C, S) is schedulable, then schedulability is preserved by removing all precedence constraints to obtain (V, T, C, S). Since for independent tasks EDF is the optimal scheduling algorithm even under a partitioned resource [15], it follows that (V, T, C, EDF) is schedulable.

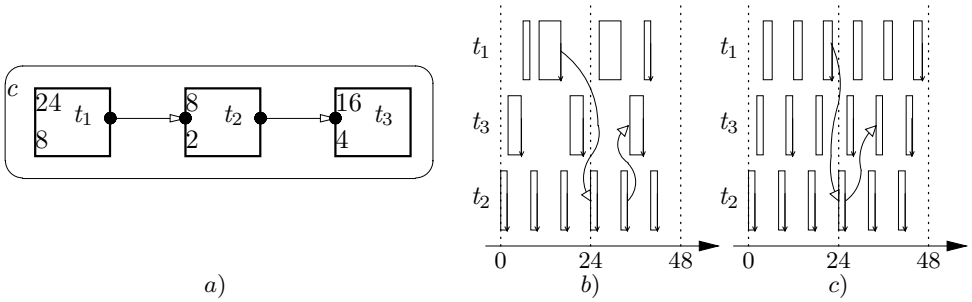

Figure 6. (a) **Task graph;** (b) RTW **schedule;** (c) LET **schedule (**RR **schedule)**

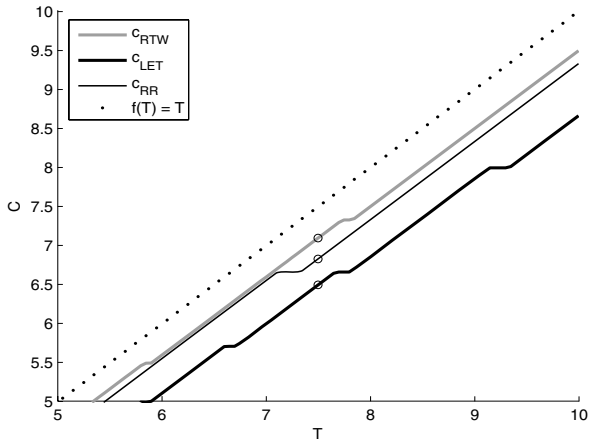

Figure 7. Abstraction functions for Fig. 6(a)

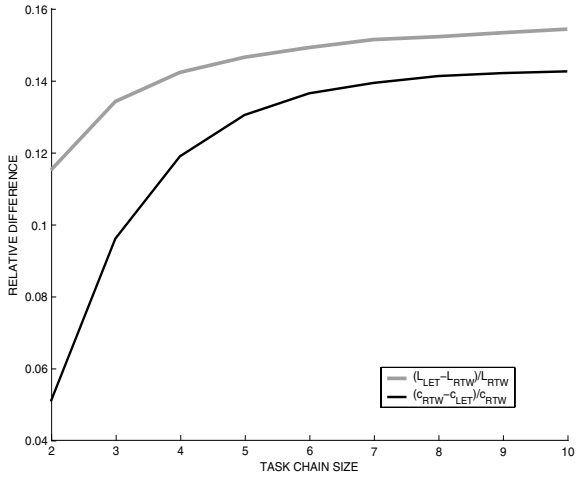

Figure 8. Relative difference between RTW **and** LET **semantics w.r.t. latency and composability**

(2) If the independent task set (V, T, C, EDF) is schedulable, then G is schedulable with the LET semantics even with task precedence constraints, since the concurrent task instances are independent. □

Theorem 1 (Tightness) *Let G be a task graph. If there exists a function c_{RTW} that tightly abstracts (G, RTW), then there exists a function c_{LET} that tightly abstracts (G, LET) and $u_{\text{RTW}} - u_{\text{LET}} \geq 0$.*

Proof. Let $G = (V, E)$ and suppose that c_{RTW} tightly abstracts (G, RTW). For all $T \in \mathbb{Q}$, we have that $(G, T, c_{\text{RTW}}(T), \text{RTW})$ is schedulable. From Lemma 1 it follows that $(V, T, c_{\text{RTW}}(T), \text{EDF})$ is schedulable, and consequently, that $(G, T, c_{\text{RTW}}(T), \text{LET})$ is schedulable. So, $c_{\text{LET}}(T)$ is defined and can only be smaller than $c_{\text{RTW}}(T)$. □

Proposition 1 *There exists a task graph G such that in Thm. 1 strict inequality holds, i.e., $u_{\text{RTW}} - u_{\text{LET}} > 0$.*

Similar to the case with a dedicated resource (e.g., [3]), a task graph G with a pair of tasks t_1 and t_2 whose periods p_1 and p_2 are not in a harmonic relation (i.e., there exists no $m \geq 1$ such that $p_2 = m \cdot p_1$ or $p_1 = m \cdot p_2$) can satisfy the proposition. An example is the task graph in Fig. $6(a)$, with c_{RTW} and c_{LET} shown in Fig. 7.

4. Distributed Task Precedence Abstraction

In this section tasks are distributed over a set of resources. We still consider abstractions of a task graph, i.e. a single task group with task precedence constraints. We again show that the LET approach provides tighter abstraction, and therefore, better composability. In this case the benefits do not only come from the fixed-priority scheduling of the RTW approach. To motivate the problem, consider a teleconferencing application with video and audio streams studied in [4]. The task graph is shown in Fig. 9 and the task parameters in Tab. 1. The application is distributed over five resources, and the goal is to find its tight abstraction.

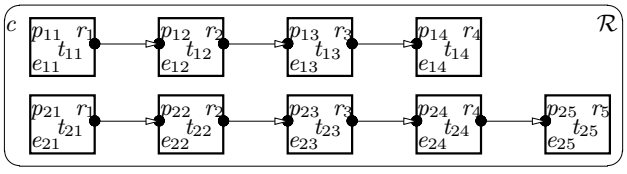

Figure 9. Teleconferencing application task graph

Let $\mathcal{R} = \{r_1, ..., r_m\}$ be a set of computational resources on which tasks from the task graph G execute, and let m be the size of $\mathcal{R}$. A task is preallocated to a resource and

Video tasks	Get frame	IO Route	IO Route	Display	Audio tasks	Get sample	LP filter	IO Route	IO Route	DA converter
Resource	Disk	Sparc	FDDI	PC	Resource	Disk	Sparc	FDDI	PC	DSP
Period	2	2	1	720	Period	384	384	768	3	3
Exec. time	0.66	1.11	0.44	401.38	Exec. time	0.73	18.43	0.49	0.49	0.60

Table 1. Example teleconferencing application data

there is no task migration. So, each task is defined with a triple (p, e, r), where $r \in \mathcal{R}$. If a task graph G consists of tasks defined with such triples and S is a semantics, we refer to (G, S) as a *distributed* program. We assume that communication between tasks can either be modeled as a task or it takes a negligible amount of time.

In the multi-resource case each resource has its own independent periodic model, so the period $T \in \mathbb{Q}^m$ and the capacity $C \in \mathbb{R}^m$ are m-tuples. We assume that the scheduler allocates different resources independently, i.e., it only ensures that for each resource the periodic requirement is individually satisfied. First note that the program schedulability definition remains exactly the same as Def. 1 for a single resource. To define tight abstraction in this case, we have to consider minimal capacity with respect to some metric, and here we use a simple multi-resource utilization metric. Given a tuple $T = (T_1, ..., T_m) \in \mathbb{Q}^m$ and a tuple $C = (C_1, ..., C_m) \in \mathbb{R}^m$, let $\mu(T, C) = \sum_{j=1}^{m} \frac{C_j}{T_j}$.

Definition 3 (Multi-resource program abstraction) *A function $c : \mathbb{Q}^m \to \mathbb{R}^m$* tightly abstracts *distributed program (G, S) if c maps each period $T \in \mathbb{Q}^m$ into the capacity $C \in \mathbb{R}^m$ such that*

1. *(G, T, C, S) is schedulable;*
2. *for each $C' \in \mathbb{R}^m$ such that (G, T, C', S) is also schedulable, we have $\mu(T, C') \geq \mu(T, C)$.*

If the function c tightly abstracts the program (G, S), then the *multi-resource abstraction utilization* function $u : \mathbb{Q}^m \to \mathbb{R}$ of (G, S) is defined by $u(T) = \mu(T, c(T))$. Note that, for a given program (G, S), while there may be several functions c that tightly abstract (G, S), the multi-resource abstraction utilization function u of (G, S) is unique. Also, the function u satisfies $0 \leq u \leq m$.

Theorem 2 (Tightness) *Let G be a task graph. If there exists a function c_{RTW} that tightly abstracts (G, RTW), then there exists a function c_{LET} that tightly abstracts (G, LET) and $u_{\text{RTW}} - u_{\text{LET}} \geq 0$.*

Proof. With an argument similar to the proof of Thm. 1, it can be shown that for each $T \in \mathbb{Q}^m$, if $(G, T, c_{\text{RTW}}(T), \text{RTW})$ is schedulable, then $(G, T, c_{\text{RTW}}(T), \text{LET})$ is also schedulable. Consequently, for each $T \in \mathbb{Q}^m$, $u_{\text{LET}}(T)$ is smaller or equal to $\mu(T, c_{\text{RTW}}(T)) = u_{\text{RTW}}(T)$. □

Consider the task graph G in Fig. 10(a), distributed over $m = 2$ resources, and notice that the tasks have equal period p. Computing the abstraction utilization function u_{RTW} is more difficult than u_{LET}, because the capacity required for resource r_2 depends on the capacity for resource r_1. The worst case is when resource r_1 is allocated at the end of a period p, and resource r_2 is allocated at the beginning (see Fig. 10(b)). If $x \in [e_2, p - e_1]$ and the capacity for r_1 is $e_1 + x$, then the capacity for r_2 has to be at least $p - x + e_2$, so that t_2 completes on time. Based on this task graph, we show in the following proposition that the difference between the abstraction utilization functions for the two semantics can be as large as $m - 1$.

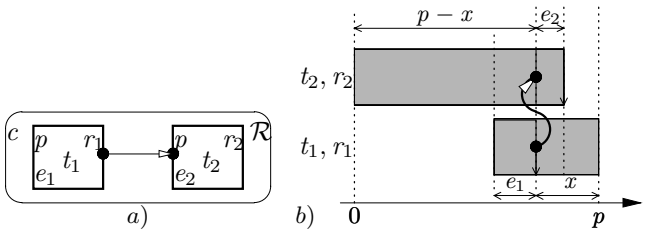

Figure 10. Example for $m = 2$ resources: (a) task graph; (b) resource partition

Proposition 2 *For all $\epsilon > 0$, there exists a task graph G and a period $T \in \mathbb{Q}^m$ such that $u_{\text{RTW}}(T) - u_{\text{LET}}(T)$ is within ϵ of $m - 1$.*

Proof. Let $G = (V, E)$ be a chain of m tasks with the same period p assigned to m different resources, i.e., a generalization of the task graph from Fig. 10(a). Let $\overline{V}$ be the task set obtained from V by modifying, for each $j = 1, ..., m$, task $(p, e_j, r_j) \in V$ into task $(p, \overline{e}_j, r_j) \in \overline{V}$, where $\overline{e}_j = e_j + x_j$ if $j = 1$, and $\overline{e}_j = p - x_{j-1} + e_j + x_j$ if $1 < j < m$, and $\overline{e}_j = p - x_{j-1} + e_j$ if $j = m$ (see Fig. 11). Let $T = (p, ..., p)$. Then (G, T, C, RTW) is schedulable if and only if $(\overline{V}, T, C, \text{EDF})$ is schedulable and $x_j \in [e_{j+1}, p - e_j]$ for each $j = 1, ..., m-1$. If e_1 is close to p, and e_j is close to 0 for each $j = 2, ..., m$, then $\overline{e}_j$ is close to p for each $j = 1, ..., m$. Consequently, $u_{\text{RTW}}(T)$ can be arbitrarily close to m. We also have that (G, T, C, LET) is schedulable if and only if (V, T, C, EDF) is schedulable.

Since e_1 can be arbitrarily close to p, and e_j arbitrarily close to 0 for each $j = 2, ..., m$, $u_{\text{LET}}(T)$ can be arbitrarily close to 1. □

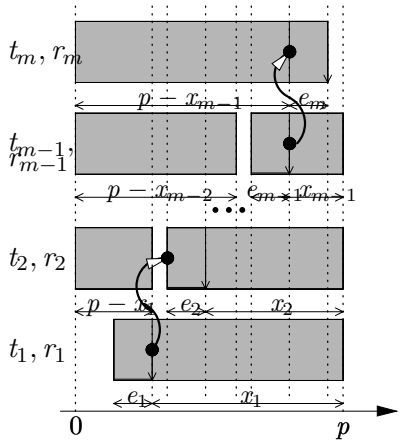

Figure 11. Resource partition for Prop. 2

Remark 1 If we assume that for all tasks (p_j, e_j, r_j) of the task graph G in Prop. 2 the execution time e_j is less than p/m, then we can consider a pipelined execution, i.e., the task graph G' which differs from G in that each task period p_j (for $j = 1, ..., m$) is equal to p/m. Such a task graph with LET semantics has the same latency as the original task graph with RTW semantics. Moreover, abstraction is still tighter for LET: $u_{\text{RTW}}(T) - u'_{\text{LET}}(T)$ can be arbitrarily close to $((m-1)p + \sum e_j - m \sum e_j)/p$, and thus can be greater than 0, since $p - \sum e_j \geq 0$. □

Remark 2 Although Prop. 2 holds for $m > 1$, its form for $m = 1$ is similar to Prop.1. However, Prop. 1 holds globally with strict inequality, which is not the case for Prop. 2 since both u_{RTW} and u_{LET} approach m as the argument T becomes large. □

We next argue that abstraction and scheduling for tasks with precedences and multiple shared resources are simpler with the LET semantics. Let $G = (V, E)$ be a task graph, and let $V_j \subseteq V$ (for $j = 1, ..., m$) be the set of all tasks allocated to the resource r_j. Let the task graph $G_j = (V_j, E \cap V_j^2)$ be the *r_j-task graph*, and let $c_j : \mathbb{Q} \to \mathbb{R}$ be a function that tightly abstracts (G_j, S).

Theorem 3 (Abstraction) *(1) The function c_{LET} that maps each period $T = (T_1, ..., T_m)$ to $c_{\text{LET}}(T) = (c_1(T_1), ..., c_m(T_m))$, tightly abstracts (G, LET). (2) There exist two task graphs G and G' with the same r_j-task graph for each $j = 1, ..., m$, such that the functions that tightly abstract (G, RTW) and (G', RTW) are not equal.*

Consider, for instance, Fig. 12. The function c_{LET} that tightly abstracts the teleconferencing program from Fig. 9 is defined by the $m = 5$ single-variable functions $c_{j,\text{LET}}$ computed independently for each resource r_j as discussed in Sec. 3.2. According to Thm. 3(2), knowing all functions $c_{j,\text{RTW}}$ is not sufficient to construct the function c_{RTW}. An example for the task graph G in Thm. 3(2) is the graph from Fig. 10(a), and for the task graph G', the same graph without the edge between t_1 and t_2.

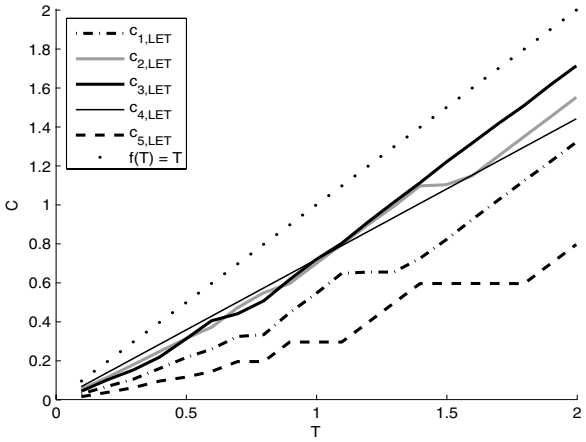

Figure 12. LET **abstraction functions for Fig. 9**

Note that the complexity of a scheduling problem on distributed task graphs, asking whether an end-to-end latency requirement is satisfied, depends on the choice of semantics. For the RTW semantics the problem is similar to the job-shop scheduling problem, and is NP-hard even in simple variants [2]. If the LET semantics is acceptable, then the scheduling problem can be decomposed into a set of simple single-resource scheduling problems. This simplicity is favorable for program admission tests.

5. Hierarchical Intergroup Abstraction

In this section we allow for the existence of precedences between different task groups, i.e., different task graphs. We discuss tightness and the construction of abstractions for such task graphs. To simplify the presentation we restrict the discussion to the single-resource case. We define a hierarchical scheduling framework that takes into account precedence constraints. In this framework the separation property between parent and children levels is not satisfied, i.e., the parent scheduler has to know the details of the entire hierarchy below it. However, we formally show that the separation property holds with LET, albeit not with RTW semantics.

We use the term "program" for the first level of a hierarchy, and "hierarchical program" for higher levels. So, programs are composed into hierarchical programs, and these are composed into higher-level hierarchical programs. Let $G = (V, E)$ be a *flat* task graph, a graph defined with the set V of all tasks and the set E of all precedences. We first

inductively define a *task hierarchy*. (1) Every set of tasks $\mathcal{H} \subseteq V$ is a task hierarchy on G. In this case, the set of vertices $\mathcal{V}$ of the task hierarchy $\mathcal{H}$ is equal to $\mathcal{H}$. (2) If $\mathcal{H}_j$ is a task hierarchy on G with a set of vertices $\mathcal{V}_j$ for each $j = 1, ..., k$, and all sets of vertices are mutually disjoint, then the collection $\mathcal{H} = \{\mathcal{H}_1, ..., \mathcal{H}_k\}$ is a task hierarchy on G. In this case, the set of vertices $\mathcal{V}$ of a task hierarchy $\mathcal{H}$ is the union of the sets of vertices of its elements, i.e., $\mathcal{V} = \cup_{j=1}^{k} \mathcal{V}_j$.

A *hierarchical task graph* $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ on the flat task graph G consists of a task hierarchy $\mathcal{H}$ on G with a set of vertices $\mathcal{V}$ and the set of precedences $\mathcal{E} = E \cap \mathcal{V}^2$. A *hierarchical program* $(\mathcal{G}, S)$ on the flat task graph G consists of a hierarchical task graph $\mathcal{G}$ on G and a semantics $S \in \{\text{RTW}, \text{LET}\}$.

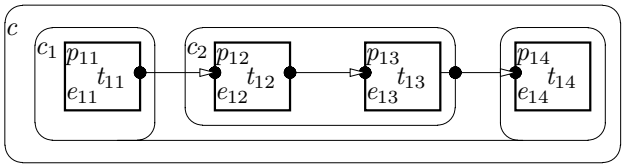

Figure 13. Video stream hierarchical abstraction

Example. Assume that the video stream from the teleconferencing application (Sec. 4) executes entirely on the same resource, and that all execution times are scaled down 4 times. Fig. 13 gives an example of task groups with intergroup precedences. There are $k = 2$ task groups at the leaf level of the hierarchy, and note that there is a precedence between the groups in each direction. The flat task graph $G = (V, E)$ is defined with $V = \{t_{11}, t_{12}, t_{13}, t_{14}\}$ and $E = \{(t_{11}, t_{12}), (t_{12}, t_{13}), (t_{13}, t_{14})\}$. There are three task hierarchies $\mathcal{H}_1 = \{t_{11}, t_{14}\}$, $\mathcal{H}_2 = \{t_{12}, t_{13}\}$, and $\mathcal{H} = \{\mathcal{H}_1, \mathcal{H}_2\} = \{\{t_{11}, t_{14}\}, \{t_{12}, t_{13}\}\}$. The corresponding sets of vertices are $\mathcal{V}_1 = \{t_{11}, t_{14}\}$, $\mathcal{V}_2 = \{t_{12}, t_{13}\}$, $\mathcal{V} = V$, and the sets of precedences are $\mathcal{E}_1 = \emptyset$, $\mathcal{E}_2 = \{(t_{12}, t_{13})\}$, and $\mathcal{E} = E$. Finally, the three hierarchical task graphs defined by the hierarchy are $\mathcal{G}_1 = (\mathcal{H}_1, \mathcal{E}_1)$, $\mathcal{G}_2 = (\mathcal{H}_2, \mathcal{E}_2)$, and $\mathcal{G} = (\mathcal{H}, \mathcal{E})$. □

Note that if in a hierarchical task graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the task hierarchy $\mathcal{H}$ is equal to a subset of tasks V, then $\mathcal{G}$ reduces to a subgraph of G, and the hierarchical program $(\mathcal{G}, S)$ reduces to a program as defined in 3.2. An example is the hierarchical task graph $\mathcal{G}_2$. For such a hierarchical program, Def. 1 defines schedulability and Def. 2 defines abstraction functions. We use these definitions as a base case for Def. 4 and Def. 5, which respectively define the same properties for higher-level hierarchical programs. The following convention holds for all remaining propositions in this section, and the proofs of the propositions are presented in the Appendix.

Convention. Let $(\mathcal{G}, S)$ be a hierarchical program on a flat task graph G with $\mathcal{G} = (\mathcal{H}, \mathcal{E})$ and $\mathcal{H} = \{\mathcal{H}_1, ..., \mathcal{H}_k\}$. Let $\mathcal{V}$ be the set of vertices of $\mathcal{H}$, and let $\mathcal{E}$ (resp. $\mathcal{E}_j$) be the set of precedences of $\mathcal{H}$ (resp. $\mathcal{H}_j$). Let $\mathcal{G}_j = (\mathcal{H}_j, \mathcal{E}_j)$ for $j = 1, ..., k$; we refer to $\{\mathcal{G}_1, ..., \mathcal{G}_k\}$ as the set of *component graphs* of $\mathcal{G}$. Let $\mathcal{C} = (c_1, ..., c_k)$ be a tuple of functions such that for each $j = 1, ..., k$ the function $c_j : \mathbb{Q} \to \mathbb{R}$ tightly abstracts the hierarchical program $(\mathcal{G}_j, S)$. Given a tuple $P = (P_1, ..., P_k) \in \mathbb{Q}^k$ and a tuple $\mathcal{C} = (c_1, ..., c_k)$ of functions $c_j : \mathbb{Q} \to \mathbb{R}$, let $V_{P,\mathcal{C}}$ be the set of independent tasks defined with $V_{P,\mathcal{C}} = \{(P_j, c_j(P_j)) \mid j = 1, ..., k\}$. □

In a hierarchical scheduling framework, a separate scheduling problem is solved at each level of the hierarchy. We assume that the scheduler at the level of a hierarchical program $((\mathcal{H}, \mathcal{E}), S)$, knows only about the precedences in $\mathcal{E}$, but not about the entire set E. The scheduler has to determine a schedule that satisfies both the requirements of the components, i.e., the requirements of the task set $V_{P,\mathcal{C}}$ for some tuple P, and all precedences introduced up to this level, i.e., the requirements of the program $((\mathcal{V}, \mathcal{E}), S)$. In the example from Fig. 13 there are two levels of scheduling. Let c_1 and c_2 be the functions that respectively abstract programs $\mathcal{G}_1$ and $\mathcal{G}_2$. The higher-level scheduler has to satisfy the requirements of the entire program (G, S), but also the requirements of the task set $\{(P_1, c_1(P_1)), (P_2, c_2(P_2))\}$ for some rationals P_1 and P_1. Since components do not specify resource requirements as a single (T, C) pair, the following definition contains an additional existential quantifier.

Definition 4 (Hierarchical program schedulability) *If under all instances of a given periodic resource (T, C), there exist a tuple $P \in \mathbb{Q}^k$ and a schedule feasible both for the set of independent tasks $V_{P,\mathcal{C}}$ and the program $((\mathcal{V}, \mathcal{E}), S)$, we say that $(\mathcal{G}, T, C, S)$ is* schedulable.

Definition 5 (Hierarchical program abstraction) *A function $c : \mathbb{Q} \to \mathbb{R}$* tightly abstracts *a hierarchical program $(\mathcal{G}, S)$ if c maps each period T into the smallest capacity C such that $(\mathcal{G}, T, C, S)$ is schedulable.*

The following theorem shows that the hierarchical scheduling framework is *compositional* for the LET semantics, but not for the RTW semantics, because in the LET case, the abstraction function (i.e., timing properties) for the composition can be established from independent abstraction functions for the components. In the case of LET semantics, we show how to construct a function that tightly abstracts a hierarchical program from the tuple $\mathcal{C}$ of functions that tightly abstract hierarchical programs at the next lower level of the hierarchy. Given $T \in \mathbb{Q}$, let

$$c_{min}(T) = \min_{P \in \mathbb{Q}^k} \{c'(T) \mid c' : \mathbb{Q} \to \mathbb{R} \text{ tightly abstracts } V_{P,\mathcal{C}}\}.$$

Consider the example from Fig. 13, and Fig. 14. The two functions drawn with dash lines, $c_{1,\text{LET}}$ and $c_{2,\text{LET}}$, are

tight abstractions of the leaf-level hierarchical programs $(\mathcal{G}_1, \text{LET})$ and $(\mathcal{G}_2, \text{LET})$, respectively. These are computed as explained in Sec. 3.2. The third function, c_{LET}, which tightly abstracts $(\mathcal{G}, \text{LET})$, is the function c_{min} from the above expression computed using $c_{1,\text{LET}}$ and $c_{2,\text{LET}}$. Beside this, the following theorem also shows that, in general, knowing the tuple of functions $c_{j,\text{RTW}}$ is not sufficient to construct the function c_{RTW} that tightly abstracts $(\mathcal{G}, \text{RTW})$.

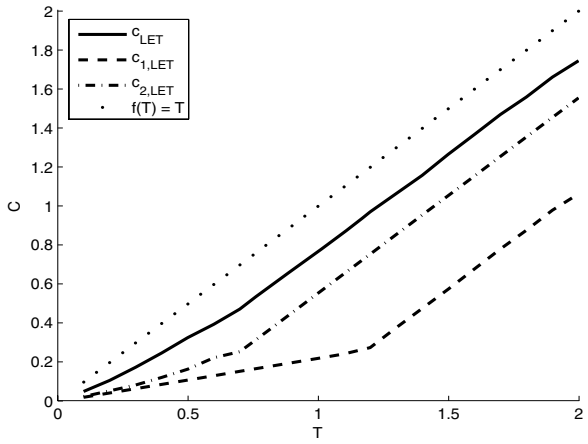

Figure 14. LET **abstraction functions for Fig. 13**

Theorem 4 (Abstraction) *(1) The function c_{LET} that maps each period T to $c_{\text{LET}}(T) = c_{min}(T)$, tightly abstracts $(\mathcal{G}, \text{LET})$. (2) There exist two hierarchical task graphs $\mathcal{G}$ and $\mathcal{G}'$ with the same set of component graphs, such that the functions that tightly abstract $(\mathcal{G}, \text{RTW})$ and $(\mathcal{G}', \text{RTW})$ are not equal.*

The next theorem shows that the hierarchical scheduling framework exhibits *separation* for the LET semantics, because for scheduling only component abstractions, and not component internals, are sufficient. For the RTW semantics, in general, knowing the tuple of functions $c_{j,\text{RTW}}$ is not sufficient to construct a feasible schedule. We explain how to construct the composite schedule in the case of LET semantics. Given $T \in \mathbb{Q}$, let $P_{min}(T)$ be $P \in \mathbb{Q}^k$ such that for a function c' that tightly abstracts $V_{P,\mathcal{C}}$ we have $c'(T) = c_{min}(T)$, i.e., let it be equal to the P that minimizes the expression defining the function c_{min}.

Theorem 5 (Scheduling) *Let $(\mathcal{G}, S)$ be a hierarchical program and (T, C) a periodic resource such that $(\mathcal{G}, T, C, S)$ is schedulable. (1) If $S =$ LET, then the* EDF *algorithm for $V_{P_{min}(T),\mathcal{C}}$ constructs a feasible schedule for $(\mathcal{G}, T, C, \text{LET})$. (2) If $S =$ RTW, then there exists a hierarchical task graph $\mathcal{G}'$ with the same set of component graphs as $\mathcal{G}$, such that no schedule is feasible both for $(\mathcal{G}, T, C, \text{RTW})$ and $(\mathcal{G}', T, C, \text{RTW})$.*

Consider again the example from Fig. 13. According to Fig. 14, if $T = 1$ then the required capacity for the hierarchical program $(\mathcal{G}, \text{LET})$ is $c_{\text{LET}}(T) = 0.776$. Fig. $15(a)$ shows an instance of the resource model $R = (1, 0.776)$ for the first three periods T, in which the resource is respectively allocated at the beginning, at the end, and in the middle of the period T. The rest of Fig. 15 shows the hierarchical generation of the schedule. If $T = 1$, then $P_{min}(T) = (1, 0.7)$, and from Fig. 14 we have $c_{1,\text{LET}}(1) = 0.22$ and $c_{2,\text{LET}}(0.7) = 0.25$. Fig. $15(b)$ shows the higher-level schedule: the EDF schedule for the two periodic tasks $(1, 0.22)$ and $(0.7, 0.25)$ in the partitions from Fig. $15(a)$, i.e., it shows the schedule for $(\mathcal{G}_1, \text{LET})$ and $(\mathcal{G}_2, \text{LET})$. Fig. $15(c)$ shows the lower-level schedule for $\mathcal{G}_1$, i.e., it shows the schedule for tasks $t_{11} = (2, 0.16)$ and $t_{14} = (720, 100.34)$. Finally, Fig. $15(d)$ shows the same for tasks $t_{12} = (2, 0.27)$ and $t_{13} = (1, 0.11)$.

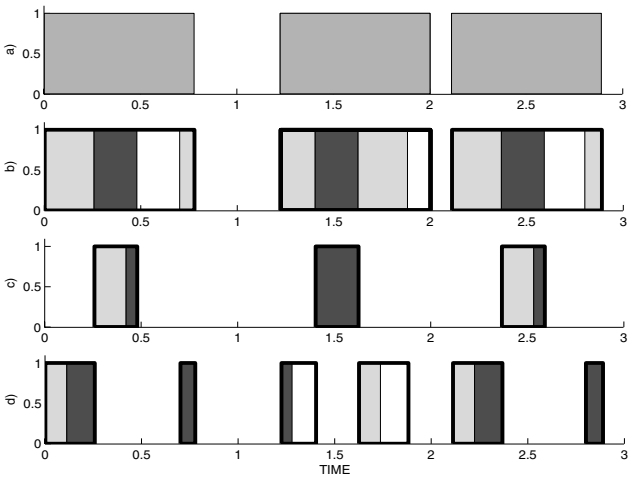

Figure 15. (a) **Instance of** $R = (1, 0.776)$**;** (b) **L:** $\mathcal{G}_2$**, D:** $\mathcal{G}_1$**;** (c) **L:** t_{11}**, D:** t_{14}**;** (d) **L:** t_{13}**, D:** t_{12} **(L=light, D=dark)**

The following two statements generalize Thm. 1 and Prop. 1.

Theorem 6 (Tightness) *If there exists a function c_{RTW} that tightly abstracts $(\mathcal{G}, \text{RTW})$, then there exists a function c_{LET} that tightly abstracts $(\mathcal{G}, \text{LET})$ and $u_{\text{RTW}} \geq u_{\text{LET}}$.*

Thm. 6 follows from Lemma 2 similar to the proof of Thm. 1.

Proposition 3 *There exists a hierarchical task graph $\mathcal{G}$ such that in Thm. 6 strict inequality holds, i.e., $u_{\text{RTW}} - u_{\text{LET}} > 0$.*

Prop. 3 follows from the proof of Thm. 4(2). An example is the hierarchical task graph from Fig. $16(a)$, whose c_{LET} function is shown in Fig. $17(a), (b)$.

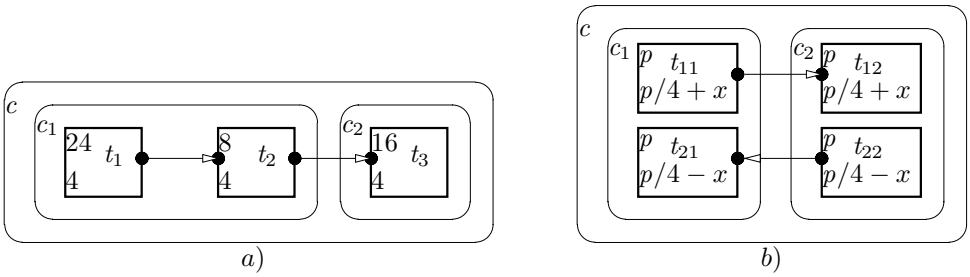

Figure 16. Intergroup precedence abstraction examples for (a) Thm. 4(2); (b) Thm. 5(2)

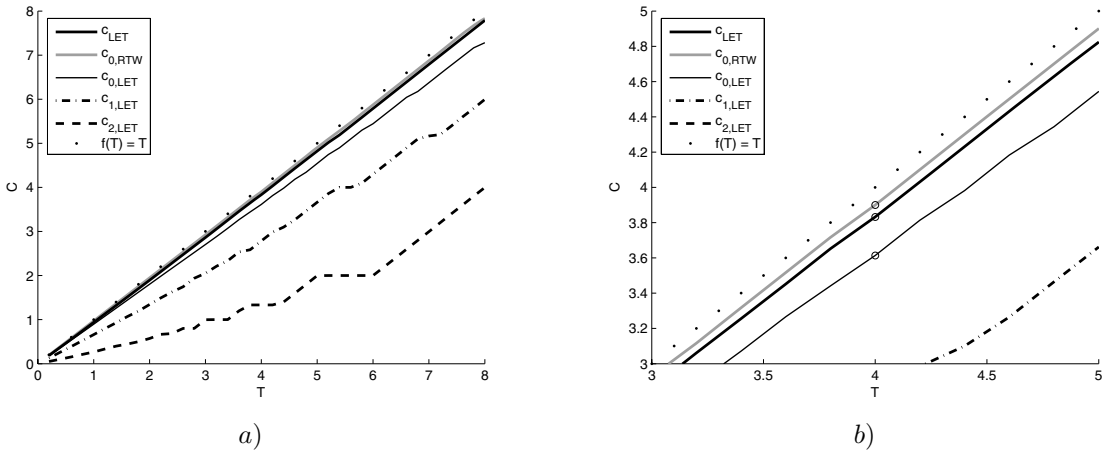

Figure 17. (a) Component abstraction function for the hierarchical program in Fig.16(a); (b) Detailed view

6. Conclusion

We addressed the problem of abstracting interacting periodic real-time components in the scope of hierarchical scheduling. We compared two semantics, RTW and LET, for task precedences, within and between components, on single or distributed resources. The results of the last two sections can be generalized for applications with both intergroup and distributed task precedences. We recognized the latency vs. composability trade-off between the two semantics. We showed that advantageous properties of a hierarchical framework, separation and compositionality, can be achieved with the LET semantics. A natural way to extend the framework would be combining the two semantics, i.e., defining a framework in which a particular semantics would be specified for each precedence constraint. The LET semantics would typically be selected for less time-critical paths in the application task graph. A potential solution for both low latency and tight abstractions is a more complicated scheduling interface. Another approach may use a different resource model. There are related efforts in this direction [16, 21]; how they can be used in the the context of interacting components is a topic for future research.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *EMSOFT*, pages 95–103. ACM Press, 2004.

[2] P. Brucker, S. Kravchenko, and Y. Sotskov. Preemptive job-shop scheduling problems with a fixed number of jobs. *Mathematical Methods of Operations Research*, 49:41–76, 1999.

[3] G. C. Buttazzo. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29:5–26, 2005.

[4] S. Chatterjee and J. K. Strosnider. Distributed pipeline scheduling: A framework for distributed, heterogeneous real-time system design. *The Computer Journal*, 38:271–285, 1995.

[5] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, pages 308–319. IEEE Computer Society, 1997.

[6] Models with multiple sample rates. In *Real-Time Workshop User Guide*, pages 1–34. The MathWorks Inc, 2005.

[7] S. Goddard and K. Jeffay. Managing latency and buffer requirements in processing graph chains. *The Computer Journal*, 44:486–503, 2001.

[8] B. Hardung, T. Koelzow, and A. Krueger. Reuse of software in distributed embedded automotive systems. In *EMSOFT*, pages 203–210. ACM Press, 2004.

[9] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.
[10] C. Kirsch, M. A. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. In *VEE*, pages 35–45. ACM Press, 2005.
[11] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *ISORC*, pages 51–60. IEEE Computer Society, 2003.
[12] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computing*, 36:24–35, 1987.
[13] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, pages 151–158. IEEE Computer Society, 2003.
[14] G. Lipari, P. Gai, M. Trimarchi, G. Guidi, and P. Ancilotti. A hierarchical framework for component-based real-time systems. In *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 209–216. Springer, 2004.
[15] A. K. Mok and A. X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *RTSS*, pages 129–138. IEEE Computer Society, 2001.
[16] A. K. Mok and A. X. Feng. Real-time virtual resource: A timely abstraction for embedded systems. In *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2002.
[17] A. K. Mok, A. X. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS*, pages 75–84. IEEE Computer Society, 2001.
[18] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *RTSS*, pages 3–14. IEEE Computer Society, 2001.
[19] J. Rushby. *Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance*. NASA Report CR-1999-209347, NASA Langley Research Center, 1999.
[20] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, pages 2–13. IEEE Computer Society, 2003.
[21] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS*, pages 57–67. IEEE Computer Society, 2004.

Appendix

Lemma 2 generalizes Lemma 1 to the hierarchical framework.

Lemma 2 *Let (T, C) be a periodic resource. (1) If $(\mathcal{G}, T, C, S)$ is schedulable for $S = \{\text{RTW}, \text{LET}\}$, then there exists a tuple $P \in \mathbb{Q}^k$ such that $(V_{P,\mathcal{C}}, T, C, \text{EDF})$ is schedulable. (2) If there exists a tuple $P \in \mathbb{Q}^k$ such that $(V_{P,\mathcal{C}}, T, C, \text{EDF})$ is schedulable, then $(\mathcal{G}, T, C, \text{LET})$ is schedulable.*

Proof. (1) Follows directly from Def. 4. (2) We prove that the schedule for the task set $V_{P,\mathcal{C}}$ is also a feasible schedule for $((\mathcal{V}, \mathcal{E}), \text{LET})$, which by Def. 4 makes $(\mathcal{G}, T, C, \text{LET})$ schedulable. A schedule for $(\mathcal{V}, \mathcal{E})$ is feasible with LET semantics if all tasks in $\mathcal{V}$ individually satisfy their timing requirements. Note that each task in $\mathcal{V}$ is an element of the set from the task hierarchy. We use induction on the structure of $\mathcal{G}$. At each level of the hierarchy a feasible schedule for $V_{P,\mathcal{C}}$ makes $(\mathcal{G}_j, P_j, c_j(P_j), \text{LET})$ schedulable for each $j = 1, ..., k$. At the leaf level this condition guarantees schedulability of each task group of the hierarchy. □

Proof of Thm. 4. (1) Given $T \in \mathbb{Q}^k$, we first prove that $(\mathcal{G}, T, c_{min}(T), \text{LET})$ is schedulable. From the definition of the function c_{min}, let $P \in \mathbb{Q}^k$ and a function c' be such that c' tightly abstracts $V_{P,\mathcal{C}}$ and $c'(T) = c_{min}(T)$. From Def. 2, it follows that $V_{P,\mathcal{C}}$ is schedulable under each instance of the periodic resource $(T, c'(T)) = (T, c_{min}(T))$. From Lemma 2 it follows that $(\mathcal{G}, T, c_{min}(T), \text{LET})$ is schedulable. Assume that there exists $C < c_{min}(T)$ such that $(\mathcal{G}, T, C, \text{LET})$ is schedulable. By Def. 4, there exists $P \in \mathbb{Q}^k$ such that $(V_{P,\mathcal{C}}, T, C, \text{LET})$ is schedulable. Let c' be the function that tightly abstracts $V_{P,\mathcal{C}}$. From Def. 2, we have $c'(T) \leq C$ and from definition of the function c_{min}, we have $c_{min}(T) \leq c'(T)$. This is a contradiction.

(2) Consider the example shown in Fig. 16(a). Let $\mathcal{G}_0 = (\{t_1, t_2, t_3\}, \{(t_1, t_2), (t_2, t_3)\})$ and $\mathcal{G} = (\{\{t_1, t_2\}, \{t_3\}\}, \{(t_1, t_2), (t_2, t_3)\})$. Let $c_{0,S}$ (resp., c_S) be the function that tightly abstracts hierarchical program $(\mathcal{G}_0, S)$ (resp., $(\mathcal{G}, S)$). Fig. 17(b) shows that $c_{0,\text{LET}} < c_{\text{LET}} < c_{0,\text{RTW}}$. On the other hand, $c_{0,\text{RTW}} \leq c_{\text{RTW}}$, since $\mathcal{G}_0$ is a leaf-level task graph containing the same tasks as $\mathcal{G}$. Let $\mathcal{G}' = (\{\{t_1, t_2\}, \{t_3\}\}, \{(t_1, t_2), (t_3, t_2)\})$ be the hierarchical task graph equal to $\mathcal{G}$, except that the edge (t_2, t_3) is of opposite direction. The function that tightly abstracts $(\mathcal{G}', \text{RTW})$ is equal to $c_{\text{LET}} < c_{\text{RTW}}$, since both edges in the hierarchical task graph $\mathcal{G}'$ introduce delays. This means that knowing only the functions that tightly abstract the lower-level hierarchical programs, and not the direction of all edges, is not sufficient to compute c_{RTW}. □

Proof of Thm. 5. (1) The schedule generated with $P = P_{min}(T)$ is feasible even if the provided capacity C is $c_{min}(T)$. From Lemma 2, it follows that the schedule is feasible for $(\mathcal{G}, T, C, \text{LET})$. (2) Consider the hierarchical task graph shown in Fig. 16(b). Let $x \in [-p/4, p/4]$ be a variable parameter of the task execution requirements not known to the scheduler at the higher-level. Let, for instance, the hierarchical graphs $\mathcal{G}$ and $\mathcal{G}'$ from the statement of Thm. 5 be as in Fig. 16(b) with $x = -p/4$ and $x = p/4$ respectively. With RTW semantics, depending on x, one or the other data precedence becomes more critical. The total requirement per hierarchical program is independent of the value for x, i.e., the functions that tightly abstract the hierarchical programs do not depend on x. However, to construct a schedule that satisfies all data precedences, the scheduler would have to know the value of x. □