Design of Safety-Critical Java Level 1 Applications Using Affine Abstract Clocks

Adnan Bouakaz
University of Rennes 1 / IRISA
adnan.bouakaz@irisa.fr

Jean-Pierre Talpin
INRIA / IRISA
jean-pierre.talpin@inria.fr

ABSTRACT

Safety-critical Java (SCJ) is designed to enable development of applications that are amenable to certification under safety-critical standards. However, its shared-memory concurrency model causes several problems such as data races, deadlocks, and priority inversion. We propose therefore a dataflow design model of SCJ applications in which periodic and aperiodic tasks communicate only through lock-free channels. We provide the necessary tools that compute scheduling parameters of tasks (i.e. periods, phases, priorities, etc) so that uniprocessor/multiprocessor preemptive fixed-priority schedulability is ensured and the throughput is maximized. Furthermore, the resulted schedule together with the computed channel sizes ensure underflow/overflow-free communications. The scheduling approach consists in constructing an abstract affine schedule of the dataflow graph and then concretizing it.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.4.7 [**Operating Systems**]: Organization and design–*real-time systems and embedded systems*

General Terms

Algorithms, Design, Languages

Keywords

Dataflow graphs, Symbolic schedulability analysis, Affine relation, Fixed-priority scheduling, Safety-critical Java

1. INTRODUCTION

In recent years, there has been a growing interest in using Java for safety-critical systems such as flight control systems, railway signaling systems, robotic surgery machines, etc. The high productivity of Java compared to low-level languages is one among multiple reasons that encourage efforts to develop Java environments for both high-end and low-end embedded devices. FijiVM, JamaicaVM, PERC, PERC Pico, KESO, and Muvium are examples of such environments. Much ongoing effort, mainly made by the JSR-302 expert group, is focused on developing the Safety-Critical Java (SCJ) specification [23]. SCJ is a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [22] and intended to develop applications that are amenable to certification under safety critical standards (such as DO-178B, Level A).

To better meet domain-specific safety requirements, the SCJ specification defines three levels of compliance (Levels 0, 1, and 2), each with a different model of concurrency, each aiming at applications of specific criticality. By the time of writing, there are few and incomplete implementations (only levels 0 and 1) of SCJ specification; for instance oSCJ [33] implemented on top of FijiVM [32], SCJ implemented on top of HVM [38], and the prototype implementation of SCJ on the Java processor JOP [35]. We have chosen Level 1 because it is less restricted than Level 0 and more analyzable than Level 2 not yet implemented. Indeed, SCJ/L0 supports only periodic handlers scheduled by a cyclic executive on a single processor; while SCJ/L2 has a much complicated concurrency model with nested missions, no-heap real-time threads, inter-processor job migration, self-suspension, etc.

In this paper, we will focus on the concurrency model of SCJ Level 1 rather than its memory model. SCJ/L1 relies entirely on periodic and aperiodic event handling. Each handler has its own server thread; this is however inconsistent with RTSJ in which a many-to-one relationship between handlers and servers is allowed [46]. Handlers communicate through shared memory; concurrency therefore becomes an issue as the programmer has to deal with data races, priority inversion, and deadlocks. In order to avoid data races, lock-based synchronization protocols are required though they are extremely complex and they complicate the schedulability analysis.

The goal of this paper is to propose a dataflow design model of SCJ/L1 applications, as an extension to the affine dataflow model [14], in which handlers communicate only through lock-free channels. Dataflow models of computation are commonly used in embedded system design to describe stream processing or control applications. Their simplicity allows waiving part of the difficult and error-prone tasks of programming real-time schedules for computations and communications from the engineering process by implementing automated code generation techniques. In the sequel, we will provide the necessary tools for computing channel sizes

SCOPES '13, June 19-21, 2013, St. Goar, Germany

and scheduling parameters that ensure schedulability and overflow/underflow-free communications.

The paper is organized as follows. The rest of this section gives an overview of SCJ/L1 and the dataflow design model. Section 2 discusses additional related work. Section 3 sketches the basic concepts of the affine dataflow model and how to construct abstract affine schedules. Sections 4 and 5 present the symbolic schedulability analysis. An empirical evaluation of the algorithms is presented in Section 6. Finally, Section 7 ends the paper with conclusions.

1.1 Concurrency model of SCJ/L1

A SCJ/L1 application consists of a set of missions executed in sequence; nested missions are hence not allowed. The infrastructure `MissionSequencer` thread is responsible of creating and terminating missions, one after another. As depicted in Figure 1, a mission starts in an initialization phase during which a set of schedulable objects (periodic and aperiodic event handlers) are created. Thus, the number and scheduling parameters (i.e. periods, phases, deadlines, priorities, etc) of handlers are known at compile-time. Handlers are released during the mission execution phase such that periodic handler releases are time-triggered while aperiodic handler releases are event-triggered. SCJ/L1 specification does not support sporadic releases; therefore, aperiodic handlers can have only soft deadlines since the assumptions necessary to check their schedulability are not part of the profile. Each handler overrides the `handleAsyncEvent()` method to provide the computational logic executed when the handler is released. This is somehow similar to firing functions of actors in dataflow graphs [24] and hence justifies our choice of a dataflow design model.

Periodic and aperiodic handlers are bound asynchronous event handlers; i.e. each handler is permanently bound to a single implementation thread. Each handler has a fixed priority so that the set of handlers is executed in parallel with a full preemptive priority-based scheduler. Handlers cannot self suspend and access to shared data is controlled by `synchronized` methods and statement blocks to avoid race conditions. The scheduler must implement the priority ceiling emulation protocol.

Scheduling aperiodic tasks based on their priorities may cause some lower priority hard tasks to miss their deadlines. One simple solution to prevent soft aperiodic tasks from interfering with hard periodic tasks is to execute them as background tasks; i.e. they execute only when there are no ready periodic jobs. However, this solution generally leads to long response time of aperiodic tasks. In the past decades, many techniques based on aperiodic servers have been devised to improve the average response time of soft aperiodic tasks; examples of such techniques are: polling servers, deferrable servers [43], sporadic servers [42], priority exchange servers, etc. An aperiodic job executes as the capacity of its server is not exhausted, then it waits for the next replenishment of the server capacity according to its replenishment period and strategy. The SCJ specification is silent about using such approaches to execute aperiodic handlers although sporadic servers, for instance, are now supported by the POSIX standard P1003.1d.

SCJ/L1 also supports multiprocessor implementations which require full partitioned scheduling (through the notion of Affinity Sets). Each handler can execute on a fixed processor with no migration. The current state of the art of multiprocessor scheduling favors the partitioned scheduling over the global one for reasons that concern the run-time cost of inter-processor migration and the optimality of schedulability tests [18].

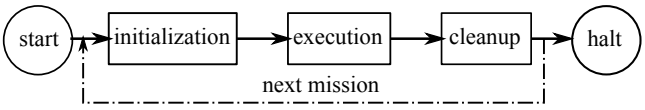

Figure 1: SCJ mission life cycle (taken from [23]).

1.2 Design model

Since missions are independent, each mission will be separately designed as a dataflow graph; while the mission sequencer can be modeled by a finite state machine. In a dataflow model of computation, an application is usually specified as a set of actors which communicate through one-to-one channels (i.e. streams). When an actor fires, it consumes tokens from its input channels and produces tokens on its output channels. Thus, an actor represents a SCJ handler. Among dataflow models of computation, synchronous dataflow (SDF) [25] and cyclo-static dataflow (CSDF) [10] are popular in the embedded system community. An actor has constant production and consumption rates in SDF and periodic rates in CSDF. In this paper, we will use a more succinct model which has ultimately periodic production and consumption rates.

The necessary user-provided information are illustrated in Figure 2. Solid rectangle nodes represent periodic actors such that a number inside a node denotes the worst-case execution time of the actor. Periodic actors communicate between each other through one-to-one FIFO channels (solid arrows). Those channels are flow-preserving; i.e. there are neither loss nor duplication of tokens. Channels are annotated by production and consumption rates. Sequence $1(2,0)$ on the output channel of actor p_1 means that the first firing (i.e. job or release) of p_1 produces one token; and then, alternatively, even jobs produce two tokens and odd jobs produce nothing. Those rates can be just a safe abstraction of the actual production and consumption rates that can be obtained by some static code analysis. Two non-communicating periodic actors can be linked together via an affine relation (described later). For instance, actors p_4 and p_3 are $(2,0,1)$-affine-related which means that actor p_3 is twice as fast as actor p_4. The set of periodic actors, their channels, and their affine relations must form a weakly connected graph.

A FIFO channel `e` will be simply implemented as a cyclic array `E` with a fixed size `s` such that the instruction `e.set(v)` is implemented as `{E[i]=v;i=(i+1)%s;}` such that `i` is

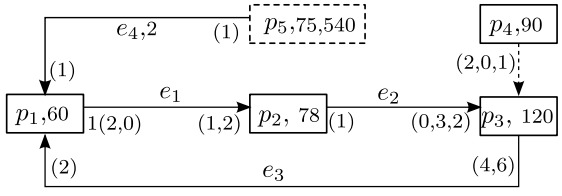

Figure 2: Example of dataflow graph with periodic and aperiodic handlers.

a local index in the producer. Calls for the `get()` method are implemented in a similar way. The analyses, described in Section 3, will ensure that there is no need for synchronization protocols to access the arrays.

Dashed rectangle nodes represent aperiodic handlers. In Figure 2, actor p_5 is an aperiodic handler bound to a sporadic server thread which has a capacity equal to 75 and a replenishment period equal to 540. Parameters of servers that minimize the average response time are generally obtained by simulation. Some selection criteria have been proposed in [8]. Since there can be multiple servers in the system, a capacity sharing protocol like the one described in [9] may increase the responsiveness of aperiodic tasks. Sporadic servers are often considered the best because they achieve a higher processor utilization and can be considered in the schedulability analysis just like sporadic tasks [42]. Bernat and Burns have shown that there is no big difference between the performance of a deferrable server and a sporadic one [8]. They have also shown that a deferrable server can be also considered in the schedulability analysis as a sporadic task with a jitter. In the sequel, we will consider only sporadic servers but the results can be easily extended to consider deferrable or polling servers.

Besides having unknown (or extremely large) worst-case execution times, aperiodic tasks have unknown inter arrival time of requests. Therefore, an aperiodic task cannot communicate with other tasks via simple FIFO channels. Indeed, these communications may be unbounded or empty and hence block the other tasks for an indeterminate time. In our design model, an aperiodic task communicates with other (periodic or aperiodic) tasks through Cyclical Asynchronous Buffers (CAB) [17]. CABs offer bounded non-blocking communications. Tokens in a CAB are maintained until they are over-written by the producer. Hence, some produced tokens are lost if the producer is faster than the consumer; and the consumer may read the same tokens several times if it is faster than the producer. This is not a problem in many control applications, where tasks require fresh data rather than the complete stream. CABs were used in HARTIK system [15] and SimpleRTK. Sample-and-hold communication mechanisms are used in many design models; for instance in the Architecture Analysis and Design Language (AADL) [34] and the Prelude compiler [30]. Based on the parameters of the aperiodic servers bound to the aperiodic tasks, it is possible to compute the average sizes of CABs. In the sequel, the word *channel* refers to the flow-preserving FIFO channels that link periodic tasks.

The user may provide further optional information: (1) Lower bounds and upper bounds on periods. For some real-time systems, frequencies of tasks must be higher than some minimal frequencies under which safety is not ensured or the service quality is poor. The frequencies also may be lower than some maximal frequencies over which a device, for example, may get damage. Chosen frequencies have to be as close as possible to the maximal ones to get better service quality (i.e. throughput). (2) Deadlines of tasks as fractions (less than one) of periods; otherwise deadlines are assumed to be equal to periods. (3) The number of identical processors for multiprocessor implementations; otherwise uniprocessor scheduling is considered. (4) Buffer sizes and number of initial tokens in channels.

In the rest of this paper, we will present the necessary algorithms that compute the scheduling parameters of periodic actors (i.e. periods, phases, deadlines, and priorities) and the channel sizes and their number of initial tokens so that: (1) The execution is free from overflow and underflow exceptions over communication channels. (2) The total sum of buffer sizes is minimized. (3) Periodic tasks will always meet their deadlines and achieve a high throughput.

2. RELATED WORK

Dataflow models of computation have been used in many real-time Java programming models such as Eventrons [39], Reflexes [40], Exotasks [4], StreamFlex [41], and Flexotasks [3]. The main goal of those restricted subsets of Java is to achieve lower latencies than what can be achieved by the real-time garbage collector. They attest the software engineering benefits of the dataflow model by offering some model-driven development capabilities such as automated code generation. Some of them also provide type systems to ensure memory isolation; i.e. tasks can communicate only through specified mechanisms such as buffers and transactional memory. The above mentioned issues have been already partially solved by the SCJ memory model or the SCJ annotation checker [45]. Those dataflow programming models lack however the necessary tools for computing buffer sizes and priority-driven scheduling parameters.

Periodic scheduling of dataflow graphs can be either static or dynamic. Static-periodic scheduling aims at creating a cyclic executive that executes actors in sequence; thus it is useful to design SCJ/L0 applications. static-periodic scheduling of (C)SDF graphs has been extensively addressed in the past decades with respect to many goals: buffer minimization under throughput constraints, computation of the trade-offs between the throughput and the buffering requirements, latency minimization, code size minimization, etc. Dynamic (or strictly) periodic scheduling of dataflow graphs aims at constructing periodic task systems for which real-time scheduling theory is applicable. There are few works that have addressed this problem. *Non-preemptive* rate-monotonic (RM) scheduling of SDF graphs has been addressed in [31]. Bamakhrama and Stefanov have addressed the preemptive earliest-deadline first (EDF) and RM scheduling of *acyclic* CSDF graphs that maximizes the throughput [5] or minimizes the latency [6].

A periodic scheduling technique of dataflow graphs with ultimately periodic rates was presented in [14]. The proposed technique computes the timing parameters of actors assuming that deadlines are equal to periods (i.e. an implicit-deadline task model). It ensures either EDF or RM feasibilities on uniprocessor systems using simple (and possibly inexact) processor utilization schedulability tests [27]. In this paper, as an extension to that approach, we propose a more general model (multiprocessor scheduling, constrained deadlines, aperiodic servers, etc) with an exact schedulability analysis: the response time analysis (RTA) [1]. We will also present a priority assignment policy that aims at reducing the buffering requirements.

Abstract clocks (discrete sets of logical instants denoting when events are observed in the system) are the main entities of synchronous languages [7] such as Signal, Esterel, and Lustre. Abstract clocks are used in our approach to describe an abstract schedule of the dataflow graph; i.e. a set of timeless scheduling constraints that relates the activation rates of actors. We are mainly interested in affine transformation of abstract clocks [37] which enjoy a canonical form and

some useful mathematical properties. A *subclass* of affine relations between abstract clocks was used in [21] to address time requirements of streaming applications on multiprocessor systems on chip. Recently, affine relations are used in [13] to construct EDF schedules of dataflow graphs. The objective was to reduce the buffer sizes by adjusting deadlines of actors.

The process of synthesizing timing parameters of task systems so as to ensure schedulability and maximize a cost function (e.g. throughput) is called symbolic schedulability analysis. Few works had addressed the symbolic schedulability problem. Cimatti et al. [16] used parametric timed automata to symbolically compute the schedulability region (i.e. the region of the parameter space that corresponds to feasible designs) of a task system scheduled using *fixed* priorities. In [20], the Inverse method for parametric timed automata is used to synthesize zones of the timing parameter space where the system is schedulable.

An analytic method that explores the f-space of a task system in order to maximize a cost function was presented in [11]. In this technique, periods are the only free variable; and the f-space is hence all the possible periods for which the system is fixed-priority schedulable. In our approach, priorities are also considered as free variables. Furthermore, since we consider only connected graphs, the optimization problem is simpler and space exploration can be avoided.

3. ABSTRACT AFFINE SCHEDULES

Our fixed-priority scheduling of dataflow graphs consists of two steps: the construction of an abstract schedule of the graph (i.e. a set of scheduling constraints) and then the concretization of the schedule (i.e. computing the concrete timing parameters). This section presents the first step which concerns only periodic actors while sporadic servers are considered in the second step. Thus, in this section, the dataflow graph refers to the subgraph composed of periodic actors, channels, and affine relations.

3.1 Affine dataflow graphs

A dataflow graph is a directed graph $G = (P, E, R)$ which consists of a finite set of periodic actors P, a set of FIFO channels E, and a set of affine relations R. Each actor p_i has a priority $\omega_i \in \mathbb{N}^*$ ($\mathbb{N}^*$ is the set of strictly positive integers) with 1 being the highest priority. In this paper, we will assume that the SCJ virtual machine has a sufficient number of priorities so that each actor can have a distinguished priority. Each channel e has exactly one producer and one consumer, and it is associated with two integer functions $x, y : \mathbb{N}^* \to \mathbb{N}$. The function x denotes the production rate; i.e. the producer writes $x(j)$ tokens on channel e during its j^{th} firing. Similarly, the function y denotes the consumption rate; the consumer reads $y(j)$ tokens from channel e during its j^{th} firing. Rate functions must be bounded since an actor cannot produce (or consume) an infinite number of tokens during a finite number of firings. Rate functions are constant in SDF, periodic in CSDF; and they are ultimately periodic in this model. The number of initial tokens on channel $e \in E$ is denoted by $\theta(e)$ s.t. $\theta : E \to \mathbb{N}$, while its size is denoted by $\delta(e)$ s.t. $\delta : E \to \mathbb{N}^*$. Since channels are implemented as separated arrays, the buffering requirements of the graph is equal to $\sum_{e \in E} \delta(e) \digamma(e)$ where $\digamma(e)$ is the size of a token in channel e.

The cumulative function of a rate function x is a monotone function $X : \mathbb{N} \to \mathbb{N}$ such that $X(j) = \sum_{k=1}^{j} x(k)$. Hence, $X(j)$ denotes the total number of produced (or consumed) tokens until and including the j^{th} firing. The j^{th} job of actor p is denoted by $p[j]$ for $j \in \mathbb{N}^*$. We will use an ILP formalism to construct the abstract schedule; let us therefore take functions $X^l, X^u : \mathbb{N} \to Q$ to be the linear lower bound and the linear upper bound of the cumulative function X, respectively. In the dataflow model, we may allow any kind of rate functions as long as their cumulative functions are linearly bounded.

Ultimately periodic rates

A rate function x is ultimately periodic if and only if $\exists j_0, \pi \in \mathbb{N}^* : \forall j \geq j_0 : x(j + \pi) = x(j)$. The notation $x = u(v)$, for two finite integer sequences u and v, means that $x(j) = u[j]$ if $j \leq |u|$, and $x(j) = v[(j - |u| - 1) \mod |v| + 1]$ otherwise; such that $|u|$ denotes the length of a finite sequence u, $\|u\|$ denotes the sum of its elements, and $u[j]$ denotes its j^{th} element. It is easy to compute the linear bounds of the cumulative function X which increases by $\|v\|$ every $|v|$ steps. So, we have that $\exists \lambda_x^l, \lambda_x^u \in \mathbb{Q} : \forall j \in \mathbb{N} : \frac{\|v\|}{|v|} j + \lambda_x^l \leq X(j) \leq \frac{\|v\|}{|v|} j + \lambda_x^u$.

In the abstract schedule, each actor p has an abstract activation clock $\hat{p}$ (i.e. an infinite ordered set of ticks). An actor instance, or job, is released at each tick of $\hat{p}$ and its execution must complete before the next tick (i.e. auto-concurrency is disabled) and cannot self-suspend. Two activation clocks can be affine-related as defined below.

Affine relations

A (n, φ, d)-affine relation between two clocks $\hat{p}_i$ and $\hat{p}_k$ has three parameters $n, d \in \mathbb{N}^*$ and $\varphi \in \mathbb{Z}$. In case φ is positive (resp. negative), clock $\hat{p}_i$ is obtained by counting each n^{th} instant on a referential abstract clock $\hat{c}$ starting from the first (resp. $(-\varphi + 1)^{th}$) instant; while clock $\hat{p}_k$ is obtained by counting each d^{th} instant on $\hat{c}$ starting from the $(\varphi + 1)^{th}$ (resp. first) instant.

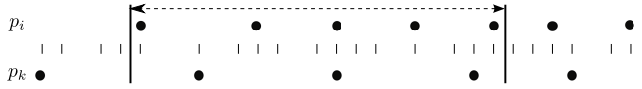

Figure 3: A $(3, -4, 5)-$affine relation. There is a positioning pattern which consists of 5 activations of p_i and 3 activations of p_k.

Figure 3 depicts a $(3, -4, 5)-$affine relation between two actors. If actors p_i and p_k are (n, φ, d)-affine-related, then there is a positioning pattern of ticks that will repeat infinitely so that for every $\frac{d}{\gcd(n,d)}$ activations of actor p_i, there are $\frac{n}{\gcd(n,d)}$ activations of actor p_k. The affine relation describes therefore the relation between the rates of activations of actors. Indeed, a $(1, \varphi, 2)$-affine relation means that the first actor is twice as fast as the second actor; while the difference between their phases is expressed by the parameter φ.

An affine relation can be described by the two monotone functions $\Delta_{i,k}, \Delta_{k,i} : \mathbb{N}^* \to \mathbb{N}$ such that $\forall j \in \mathbb{N}^* : \Delta_{i,k}(j) = \max\{0, j' \in \mathbb{N}^* |$ job $p_k[j']$ is released strictly before job $p_i[j]\}$. The *relative positioning* of ticks of clocks $\hat{p}_i$ and $\hat{p}_k$ can be described entirely by these two functions. They do not however describe the physical duration between

clock ticks; hence the notion of *abstract* clocks. We have that $\forall j \in \mathbb{N}^*$

$$\Delta_{i,k}(j) = \max\{0, \left\lceil \frac{n(j-1)-\varphi}{d} \right\rceil\}$$

$$\Delta_{k,i}(j) = \max\{0, \left\lceil \frac{d(j-1)+\varphi}{n} \right\rceil\}$$

The sign $\lceil x \rceil$ refers to the smallest integer not less than x. From these two equations, if p_i and p_k are (n, φ, d)-affine-related, then equivalently p_k and p_i are $(d, -\varphi, n)$-affine-related.

3.2 Affine relation synthesis

The abstract affine schedule will consist of the set of affine relations R in addition to an affine relation between every two communicating actors. In the sequel, we will show how to use integer linear programming to compute the appropriate affine relations so that: (1) Overflow and underflow exceptions are *statically* excluded to ensure functional determinism. (2) The abstract affine schedule is consistent. (3) The buffering requirements are minimized.

An underflow exception occurs when an actor attempts to read from an empty channel; while an overflow exception occurs when an actor attempts to write into a full channel. Excluding overflow and underflow exceptions implies that the number of accumulated tokens on every channel and at each step of the execution is greater or equal to zero (i.e. no underflows) and less or equal to the buffer size (i.e. no overflows).

Let p_i be the producer and p_k be the consumer of channel $e = (p_i, p_k, x = u_1(v_1), y = u_2(v_2))$ such that actors p_i and p_k are $(n, \varphi, d)-$affine-related. When jobs $p_i[j]$ and $p_k[j']$ complete, the number of accumulated tokens on channel e is given by $\theta(e) + X(j) - Y(j')$. Of course, indices j and j' are affine-related as shown in details later.

- No overflow exception over channel e means that:

$$\forall (j, j'), \theta(e) + X(j) - Y(j') \leq \delta(e) \quad (2)$$

- No underflow exception over channel e means that:

$$\forall (j', j), \theta(e) + X(j) - Y(j') \geq 0 \quad (3)$$

3.2.1 Overflow analysis

Let us suppose that actors p_i and p_k are allocated to the same processor and that $\omega_i < \omega_k$ (i.e. p_i has a higher priority than p_k). The linear upper bound of accumulated tokens $\theta(e) + X^u(j) - Y^l(j')$ always being smaller or equal to $\delta(e)$ is a sufficient condition to exclude overflow exceptions. We have that $X^u(j) = \frac{\|v_1\|}{|v_1|} j + \lambda_x^u$ and $Y^l(j') = \frac{\|v_2\|}{|v_2|} j' + \lambda_y^l$. It remains to compute the linear lower bound of j' in terms of j according to the affine relation and the scheduling policy.

Figure 4 (cases (a) and (b)) shows what job of p_k precedes immediately job $p_i[j]$. In case (b), jobs $p_i[j]$ and $p_k[j'+1]$ are released simultaneously which implies that job $p_k[j']$ completes before job $p_i[j]$ writes any token on the channel. In case (a), job $p_k[j']$ is released before job $p_i[j]$; but since that latter has a higher priority, it can preempt job $p_k[j']$ before it reads some tokens (we do not assume any knowledge about the implementation code of the firing functions). Thus, we are only certain that job $p_k[j'-1]$ reads all its needed tokens before job $p_i[j]$ starts writing some results.

So, if d divides $(n(j-1)-\varphi)$, then $j' = \Delta_{i,k}(j)$; otherwise $j' = \Delta_{i,k}(j) - 1$. Hence, the linear lower bound of j' is given by $\frac{n}{d} j - \frac{\varphi+n+d-1}{d}$. By substituting all the linear bounds in Equation 2, we obtain the following linear constraint. $\forall j \in \mathbb{N}^*$:

$$\theta(e) - \delta(e) + \frac{\|v_2\|}{|v_2|d}\varphi + \xi j \leq \lambda_y^l - \lambda_x^u - \frac{\|v_2\|}{|v_2|}(\frac{n+d-1}{d}) \quad (4)$$

such that $\xi = \frac{\|v_1\|}{|v_1|} - \frac{\|v_2\| n}{|v_2| d}$.

Since j tends to infinity, it is a requirement for an execution free of overflows and underflows that ξ equals zero. Consequently, the boundedness criterion is:

$$\frac{n}{d} = \frac{\|v_1\|}{|v_1|} \frac{|v_2|}{\|v_2\|} \quad (5)$$

This boundedness criterion is equivalent to the balance equation in static-periodic scheduling of (C)SDF graphs.

If p_k has a higher priority than p_i (i.e. $\omega_k < \omega_i$), then the value of j' is computed as depicted in cases (c) and (d) of Figure 4. In case actors p_i and p_k are allocated to different processors, the value of j' is computed like in cases (a) and (b) since priorities will not ensure, for instance, that job $p_k[j']$ of case (c) will precede job $p_i[j]$. Indeed, they can execute in parallel.

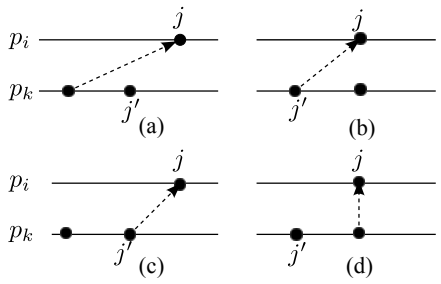

Figure 4: Precedences in the overflow analysis.

3.2.2 Underflow analysis

The underflow analysis is roughly dual to the overflow analysis. The linear lower bound of accumulated tokens $\theta(e)+X^l(j)-Y^u(j')$ always being greater or equal to zero is a sufficient condition to exclude underflow exceptions. Again, we need to compute the linear lower bound of j in terms of j'. Let us suppose that $\omega_i < \omega_j$ and that p_i and p_k are allocated to the same processor. If n divides $(d(j'-1)+\varphi)$, then $j = \Delta_{k,i}(j') + 1$; otherwise $j = \Delta_{k,i}(j')$. Hence, the linear lower bound of j is given by $\frac{d}{n} j' + \frac{\varphi-d+1}{n}$. By substituting all the linear bounds in Equation 3, we obtain the following linear constraint.

$$\forall j' \in \mathbb{N}^* : \theta(e) + \frac{\|v_2\|}{|v_2|d}\varphi \geq \lambda_y^u - \lambda_x^l + \frac{\|v_2\|}{|v_2|}(\frac{d-1}{d}) \quad (6)$$

3.2.3 Consistency

If the dataflow graph is acyclic then each affine relation can be computed independently. However, if there are some cycles in the graph, then some constraints must be satisfied in order to construct a consistent schedule. If there is no consistent schedule, then the graph cannot be executed (as a periodic task system) in bounded memory and/or without underflow exceptions. It is worth mentioning that some cyclic graphs have deadlock-free bounded static-periodic schedules but not strictly periodic ones.

The graph of affine relations is an undirected graph where nodes represent actors and edges represent affine relations (recall that an affine relation can be reversed). Affine relations are those user-provided (i.e. the set R) or those relating communicating actors. Consistency of the affine abstract schedule is given by the following proposition.

PROPOSITION 1 (**from [14]**). *The schedule is consistent if for every fundamental cycle* $p_1 \xrightarrow{(n_1,\varphi_1,d_1)} p_2 \rightarrow \cdots \rightarrow p_m \xrightarrow{(n_m,\varphi_m,d_m)} p_1$ *in the graph of affine relations, we have that*

$$\prod_{i=1}^{m} n_i = \prod_{i=1}^{m} d_i \tag{7a}$$

$$\sum_{i=1}^{m} (\prod_{j=1}^{i-1} d_j)(\prod_{j=i+1}^{m} n_j)\varphi_i = 0 \tag{7b}$$

Fundamental cycles can be obtained easily after constructing a spanning tree of the graph of affine relations. They form a cycle basis of the cycle space; i.e. any cycle of the graph can be obtained by disjoint union of some fundamental cycles. Therefore, if Proposition 1 is satisfied for fundamental cycles, then it is also satisfied for any cycle in the graph.

Equation 7a concerns the boundedness of the schedule. If it is not satisfied, then the execution requires unbounded memory. It follows directly from the fact that for each d_i activations of p_i, there are n_i activations of p_k such that $k = (i+1) \mod m+1$. Equation 7b concerns consistency of parameter φ of affine relations. Since the difference between phases of two successive actors in the cycle is encoded with the parameter φ, Equation 7b ensures that the difference between the phases of each actor and itself is null.

3.2.4 Algorithm

Let us suppose that priorities are computed and actors are allocated to processors (as described later). Firstly, we use the boundedness criterion (Equation 5) to compute the parameters n and d of all the affine relations. Then, we use Equation 7a to check the consistency of the graph of affine relations.

To compute the parameters φ, we construct an integer linear program by applying Equations 4, and 6 on channels; and Equation 7b on fundamental cycles. The objective function of the linear program is to minimize the buffering requirements. Sizes obtained by the solution of the linear program are a safe approximation of the actual sizes. Therefore, they may be recomputed after obtaining the affine relations. For a channel $e = (p_i, p_k, x, y)$, the minimum number of initial tokens is given by $\theta(e) = |\min\{0, \min_{j'}\{X(\Gamma(j')) - Y(j')\}\}|$ such that $p_i[\Gamma(j')]$ is the last job of p_i *certainly* finished before $p_k[j']$ starts reading tokens. The minimum size of the channel is given by $\delta(e) = \max_{j}\{\theta(e) + X(j) - Y(\Gamma(j))\}$ such that $p_k[\Gamma(j)]$ is the last job of p_k *certainly* finished before job $p_i[j]$ starts writing on the channel. Since an affine relation have a positioning pattern and rates are ultimately periodic, these computations are limited to a given number of instances of the positioning pattern.

4. UNIPROCESSOR SCHEDULABILITY ANALYSIS

The task system consists of a set of periodic actors P and a set of aperiodic actors S. An aperiodic actor p_s is handled by a sporadic server with user-provided capacity C_s and replenishment period π_s. A periodic actor is mapped to a periodic task $(C_i, \pi_i, r_i, d_i, \omega_i)$ such that C_i is the worst-case execution time, π_i is the period, d_i is the deadline, and r_i is the phase. Aperiodic servers are assigned higher priorities than periodic tasks. The processor utilization of a task p_i is $U_i = \frac{C_i}{\pi_i}$. So, the utilization of periodic tasks is $U_p = \sum_{p_i \in P} U_i$; while the utilization of sporadic servers is $U_s = \sum_{p_i \in S} U_i$. The total utilization U is hence equal to $U_p + U_s$.

Symbolic schedulability analysis is the process of computing the scheduling parameters $(\pi_i, r_i, d_i, \omega_i)$ of each periodic actor p_i so that: (1) They respect the affine abstract schedule, the bounds on periods, and other constraints if any. (2) They ensure fixed-priority schedule feasibility. (3) They maximize the processor utilization factor U. Other cost functions (e.g. power consumption) can be also considered.

The first requirement implies that each $(n, \varphi, d)-$affine relation between actors $p_i, p_k \in P$ is concretized as follows.

- $n\pi_k = d\pi_i$.
- $r_k - r_i = \frac{\varphi}{n}\pi_i$.

In words, the concretization imposes constant time intervals between ticks of every activation clock. Since the graph of affine relations is connected and from the first equation, the periods and deadlines of all actors can be expressed in terms of the period of an arbitrary actor. Therefore, $\forall p_i \in P : \pi_i = \alpha_i T$ and $d_i = \beta_i T$ such that $\alpha_i, \beta_i \in \mathbb{Q}$ and $T^l \leq T \leq T^u$. We can also put $U_p = \frac{\sigma}{T}$. The bounds on T, if any, are deduced from the bounds on periods imposed by the user and from the constraint $C_i \leq d_i$. Since timing parameters are integers, T should be multiple of some integer B so that the previous equations may have a solution.

A necessary condition for a task system to be schedulable on $m \geq 1$ processors is that $U \leq m$. So, $T^l = \max\{T^l, \left\lceil \frac{\sigma}{m-U_s} \right\rceil\}$. Hence, the values of T to be tested are in $\{T | T^l \leq T = kB \leq T^u \wedge k \in \mathbb{N}\}$. The enumerative solution consists in checking the schedulability of the task system for each T in an increasing order, staring from the minimum value and until finding an appropriate one or exceeding the upper bound T^u.

4.1 Priority assignment

This step should be performed before affine relation synthesis. Deadline-monotonic (DM) priority ordering policy assigns the highest priority to the task with the shortest deadline. When deadlines are equal to periods, DM priority ordering is equivalent to RM priority ordering. DM priority ordering is optimal for synchronous tasks systems (i.e. there is an instant at which all tasks are released simultaneously) [26]; i.e. if the system is feasible for a given priority assignment, then it is also feasible under DM priority assignment. Hence, DM priority ordering gives the maximal throughput. For asynchronous task systems, the optimal priority assign-

ment is the one proposed in [2]. It has however a much higher complexity than DM priority assignment.

Buffer minimization

As shown in the overflow and underflow analyses, the priorities affect the buffer sizes computation. Let $w_{i,k}$ be the sum of approximate sizes of channels between p_i and p_k assuming that $\omega_i < \omega_k$. The approximate size of a channel $e = (p_i, p_k, x = u_1(v_1), y = u_2(v_2))$ can be obtained from Equations 4 and 6 to be $\lambda_x^u + \lambda_y^u - \lambda_x^l - \lambda_y^l + \frac{\|v_2\|}{|v_2|}(\frac{n+2d-2}{d})$. Hence, the approximate gain that comes from switching priorities is

$$\frac{\|v_2\|}{|v_2|}(\frac{d-n}{d}) \tag{8}$$

Let us construct a complete directed graph where nodes represent actors and an edge between two nodes p_i and p_k has a weight equal to $w_{i,k}$. Hence, the problem of finding the priority assignment that reduces the buffering requirements is equivalent to the linear ordering problem (LOP) [28] which consists in finding an acyclic tournament in the graph such that the sum of weights of edges in the tournament is minimal. A tournament is a subset of edges containing for every pair of nodes p_i and p_k either edge (i,k) or (k,i) but not both. For N actors, there are $N!$ different priority assignments (i.e. permutations). The LOP is NP-hard; however many exact and heuristic solutions had been proposed in the past decades [28].

The LOP can be formulated as a 0/1 integer linear program. A 0/1 variable $x_{i,k}$ states whether edge (i,k) is present in the tournament or not.

$$\min \sum_{i,k} w_{i,k}\ x_{i,k}$$
$$\forall i < k : x_{i,k} + x_{k,i} = 1$$
$$\forall i < j, i < k, j \neq k : x_{i,j} + x_{j,k} + x_{k,i} \leq 2$$
$$\forall i, k : x_{i,k} \in \{0,1\}$$

This LOP priority assignment strategy reduces the buffering requirements. However, it may jeopardize the throughput; i.e. it results in less processor utilization than what is obtained by the DM priorities. To obtain a compromise solution without enumerating all the $N!$ possibilities, we can constrain the difference between the LOP assignment and DM assignment. The difference between two permutations is taken as the precedence distance between them. If $\tilde{x}_{i,k}$ is the 0/1 variable that states that actor p_i has a higher DM priority than p_k or not, then the precedence distance is equal to $\frac{N(N-1)}{2} - \sum_{i,k} x_{i,k}\ \tilde{x}_{i,k}$. In the ILP formulation, we need to add a constraint that states that the precedence distance is less than some threshold.

4.2 Schedulability analysis

Response time based schedulability analysis is more accurate than utilization based schedulability tests. A worst case response time formulation for the deferrable and sporadic servers can be found in [8]. The worst-case response time R_i of a periodic task $p_i \in P$ is given by

$$R_i = C_i + \sum_{\omega_k < \omega_i} \left\lceil \frac{R_i}{\pi_k} \right\rceil C_k \tag{9}$$

The sum in Equation 9 includes all the sporadic servers since they have the highest priorities (a sporadic server can be considered as a sporadic task). The equation can be solved with a recurrence approach starting with $R_i^0 = C_i$ and until $R_i^{n+1} = R_i^n$. If $R_i \leq d_i$ for all hard periodic tasks, then all these tasks will meet their deadlines. The enumerative solution consists in applying the response time analysis for values $T = kB \in [T^l, T^u]$. This search space can be huge, and need to be reduced.

Reducing the search space

As noticed in [36], we have that $x \leq \lceil x \rceil < x+1$. Thus, $R_i^l \leq R_i \leq R_i^u$ where $R_i^l = \frac{C_i}{1 - \sum_{\omega_k < \omega_i} U_k}$ and $R_i^u = \frac{C_i + \sum_{\omega_k < \omega_i} C_k}{1 - \sum_{\omega_k < \omega_i} U_k}$. A better upper bound is proposed in [12] as

$$R_i^u = \frac{C_i + \sum_{\omega_k < \omega_i} C_k(1 - U_k)}{1 - \sum_{\omega_k < \omega_i} U_k}$$

Following a similar reasoning, we can prove that if the task system is schedulable, then

$$R_i^l = \frac{C_i - \sum_{\omega_k < \omega_i} C_k(1 - U_k)}{1 - \sum_{\omega_k < \omega_i} U_k}$$

If $\exists p_i \in P : R_i^l > d_i$, then the task system is infeasible. Recall that $\forall p_i \in P : \pi_i = \alpha_i T$ and $d_i = \beta_i T$. Let us put $a = \sum_{p_i \in S} C_i(1 - U_i)$ and $hp(i)$ be the set of periodic tasks that have higher priorities than actor p_i. We have that

$$R_i^l > d_i \Leftrightarrow$$
$$\beta_i(1-U_s)T^2 + (a - C_i + \sum_{p_k \in hp(i)} C_k(1 - \frac{\beta_i}{\alpha_k}))T - \sum_{p_k \in hp(i)} \frac{C_k^2}{\alpha_k} < 0$$

The second degree equation admits two solutions; the first solution is negative while the second one t_i^l is positive. Therefore, we have that

$$T < \max_{p_i \in P} t_i^l \implies \text{The task set is unschedulable}$$

Dually, if $\forall p_i \in P : R_i^u \leq d_i$, then the task set is schedulable. We have that

$$R_i^u \leq d_i \Leftrightarrow$$
$$\beta_i(1-U_s)T^2 - (a + C_i + \sum_{p_k \in hp(i)} C_k(1 + \frac{\beta_i}{\alpha_k}))T + \sum_{p_k \in hp(i)} \frac{C_k^2}{\alpha_k} \geq 0$$

If the second degree equation admits no solution or only one solution, then the inequality is always satisfied. If the second degree equation admits two solutions t_i and t_i^u, then both solutions are positive. Therefore, if $T \in [0, \min_{p_i \in P} t_i] \cup [\max_{p_i \in P} t_i^u, \infty[$, then the task system is schedulable. But, we have that $\min_{p_i \in P} t_i = 0$ (try, for instance, the periodic task with the highest priority). Hence,

$$T \geq \max_{p_i \in P} t_i^u \implies \text{The task set is schedulable}$$

To sum up, the task system is schedulable if $T \in [\max_{p_i \in P} t_i^u, T^u]$ and unschedulable if $T \in [T^l, \max_{p_i \in P} t_i^l[$. To find the best T, it is sufficient to perform a dichotomic search in the interval $[\max_{p_i \in P} t_i^l, \max_{p_i \in P} t_i^u[$ knowing that if $R_i \leq d_i$ for some T, then $R_i \leq d_i$ for all $T' > T$.

5. MULTIPROCESSOR SCHEDULABILITY ANALYSIS

The predominant approach to multiprocessor scheduling of hard-real time systems is partitioned. The major advantage of partitioned techniques is that, once an allocation of actors to processors has been achieved, it is possible to apply real-time symbolic schedulability analyses for uniprocessor systems; for instance, the technique proposed in the previous section. Partitioning the hard tasks on m identical processors to optimize some criterion is somehow equivalent to the bin-packing problem and hence NP-hard. Therefore, heuristics are attractive solutions. They consist in sorting tasks by some criteria and then assigning each task to a processor according to some conditions. Sporadic servers are allocated to processors before hard tasks in a way that balance the processor utilization factor of each partition.

Periodic actors are ordered by decreasing priorities (i.e. the task with the highest priority is considered first) computed as described in the previous section (either DM priority assignment or LOP assignment). Let $(V_i)_{i=1,m}$ be the m partitions. We assign task p_k, assuming that higher priority tasks are already assigned, as follows. Let T_i be the value of T returned by the symbolic response time analysis, described in the previous section, and which is the minimum value of T for which $V_i \cup \{p_k\}$ is schedulable on one processor. We assign task p_k to the partition whose T_i is minimum. In case of equality, we break the tie by favoring the partition with the minimum utilization.

For N periodic tasks, we will apply the symbolic response time analysis $N \times m$ times. However, adding task p_k to a partition V_i does not change the worst-case response times of the already assigned tasks. Hence, if T_i' is the minimum value for which V_i is schedulable, then $T_i = \max\{T_i', \min\{T | R_k \leq d_k\}\}$ is the minimum value for which $\{p_k\} \cup V_i$ is feasible.

We can use the approximate schedulability condition proposed in [19] for DM priority assignment. Task p_k can be assigned to partition V_i (i.e. $V_i \cup \{p_k\}$ is schedulable on one processor) if $d_k - \sum_{p_j \in V_i} (C_j + U_j d_k) \geq C_k$. In the symbolic case, this means that p_k will be assigned to the partition V_i with the minimum $\frac{C_k}{\beta_k} + \sum_{p_j \in V_i} (\frac{C_j}{\beta_k} + \frac{C_j}{\alpha_j})$. You may notice that this approximate condition is not just more than $R_k^u \leq d_k$ with $R_k^u = \frac{C_k + \sum_{p_j \in V_i} C_j}{1 - \sum_{p_j \in V_i} U_j}$. However, we have shown in the previous section that there is a better upper bound.

This best-fit allocation strategy aims at maximizing the processor utilization. As shown in the overflow and underflow analyses, the tasks allocation affects the buffer sizes computation. Hence, an allocation strategy that reduces the buffering requirements can be more suitable for systems with strong memory constraints. For instance, a task p_k could be assigned to a partition so that the total sum of sizes of channels that connect p_k to the already assigned tasks is minimized.

6. EXPERIMENTAL VALIDATION

We evaluate our scheduling technique w.r.t. buffering requirements and processor utilization factor by performing an experiment on a set of real-life applications and some randomly generated dataflow graphs.

6.1 Throughput comparison

Figure 6 shows the ratios of processor utilizations to the number of processors ($\frac{U}{m}$) obtained by different tools for a set of 19 real-life applications. Those applications are modeled as (C)SDF graphs (see [5] for more details) and to be scheduled as implicit-deadline periodic task systems with an RM priority assignment policy. `ADF` denotes results obtained by the symbolic response time analysis presented in this paper; while `DARTS` denotes the results obtained by the `DARTS` tool presented in [5]. In the uniprocessor case, `ADF` vastly outperforms `DARTS`. Indeed, this latter uses a simple inexact processor utilization scheduling test which states that the set of N tasks with implicit deadlines are RM schedulable if tasks utilization is less or equal to a utilization bound; i.e. if $U \leq N(2^{\frac{1}{N}} - 1)$. The utilization bounds are more pessimistic in partitioned multiprocessor RM scheduling. As shown in [29], the utilization bound for any fixed-task priority partitioning algorithm is upper bounded by $\frac{m+1}{1+2^{\frac{1}{m+1}}}$. For $m = 2$, the utilization bound is equal to 1.327; while our best-fit partitioning technique with symbolic response time analysis results in utilizations up to 1.995.

We notice that utilization factor of the `Satellite` and `CD2DAT` applications do not increase when adding more processors. This can be explained, for instance in the `Sattelite` application, as follows. We have that $U = \frac{188.12}{T}$ which implies that $T \geq \frac{188.12}{m}$. However, T must be a multiple of $B = 220$. Thus, increasing the number of processors will not improve the utilization factor.

Channels with large value of B are those with mis-matched I/O channels. A channel $e = (p_i, p_k, u_1(v_1), u_2(v_2))$ is said to be a matched I/O channel if values $\frac{\|v_1\|}{|v_1|}$ and $\frac{\|v_2\|}{|v_2|}$ are close to each other. It is said to be a perfectly matched I/O channel if $\frac{\|v_1\|}{|v_1|} = \frac{\|v_2\|}{|v_2|}$. Except 5 graphs, benchmarks used in Figure 5 consist of perfectly matched I/O channels.

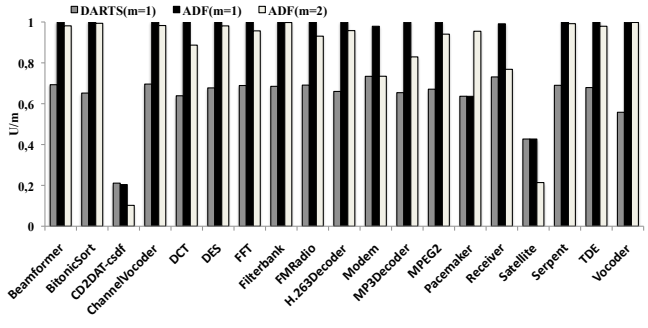

Figure 5: Comparison of achieved processor utilization factors.

6.2 Buffering requirements comparison

From Equation 8, the gain in buffering requirements that comes from switching priorities for a perfectly matched I/O channel is null. Thus, the previous benchmarks do not allow us to measure the benefit that comes from playing on priorities. Therefore, we will compare the buffering requirements of 38 randomly generated SDF graphs for different configurations, as depicted in Figure 6. The graphs are generated by the SDF^3 tool [44] with the following setting. Production and consumption rates follow a normal distribution with a

mean equal to 5 and a variance equal to 4. Worst-case execution times follow a normal distribution with a mean equal to 1000 and a variance equal to 300. The graphs are generated with different number of nodes (from 6 to 80) such that the average degree of each node is 2. The graphs are scheduled as implicit-deadlines tasks systems on one processor for DM, LOP, and constrained LOP priority assignments.

S denotes the improvement in buffering requirements when using LOP priority assignment compared to the DM priority assignment. The average improvement is equal to 37.99% with a low standard deviation (0.07). This is a good improvement but comes at the price of a throughput decline with an average equal to 48.31% as denoted by F. Thus, LOP priority assignment can be used for systems with strong memory constraints. We should also note that LOP assignment does not decrease the throughput of 31.57% of the graphs. Those graphs are graphs whose factor B is too high that changing the priorities does not affect the throughput.

In another configuration, we have constrained the LOP priority assignment so that the precedence distance between it and the DM assignment is upper bounded by $\frac{N(N-1)}{10}$. The obtained results are denoted by S^* and F^*. We obtained a slightly less buffering improvement (average equal to 34.99%) but for much less throughput decrease (average equal to 21.83%) and the constrained LOP assignment doses not change the utilization in 44.73% of the cases.

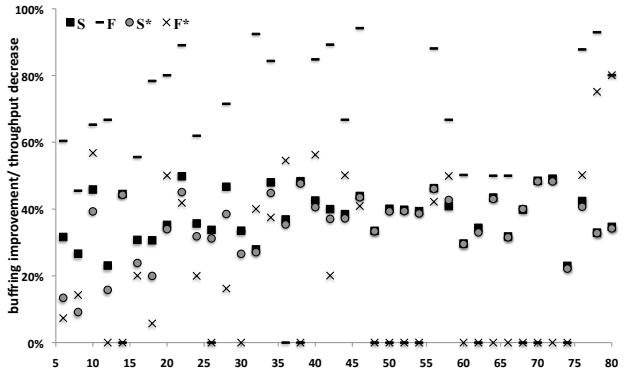

Figure 6: Comparison of buffering requirements.

7. CONCLUSION

We have presented a dataflow design model of SCJ Level 1 applications in which a set of periodic and aperiodic handlers are scheduled by a fixed-priority scheduler on uniprocessor or multiprocessor systems. We have also provided the necessary tools to compute the buffer sizes and the scheduling parameters that maximize the throughput or minimize the buffering requirements. We have also shown that symbolic response time analysis of dataflow graphs presented in this paper is more accurate than the symbolic schedulability analyses based on the utilization bounds. Our ongoing work aims at applying our scheduling approach to schedule AADL specifications on multiprocessor architectures.

8. REFERENCES

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pr-emtive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[2] N. C. Audsley. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, 2001.

[3] J. Auerbach, D. F. Bacon, R. Guerraoui, J. H. Spring, and J. Vitek. Flexible task graphs: a unified restricted thread programming model for Java. *SIGPLAN Not.*, 43(7):1–11, 2008.

[4] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. Rajan, H. Roeck, and R. Trummer. Java takes flight: time-portable real-time programming with Exotasks. *SIGPLAN Not.*, 42(7):51–62, 2007.

[5] M. Bamakhrama and T. Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the 9th ACM International Conference on Embedded Software*, pages 195–204, 2011.

[6] M. A. Bamakhrama and T. Stefanov. Managing latency in embedded streaming applications under hard-real-time scheduling. In *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–92, 2012.

[7] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. D. Simone. The synchronous languages 12 years later. In *Proceedings of the IEEE*, pages 64–83, 2003.

[8] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In 68-78, editor, *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[9] G. Bernat and A. Burns. Multiple servers and capacity sharing for implementing flexible scheduling. *Real-Time Syst.*, 22(1/2):49–75, 2002.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44:397–408, 1996.

[11] E. Bini and M. Di Natale. Optimal task rate selection in fixed priority systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 399–409, 2005.

[12] E. Bini, T. H. C. Nguyen, P. Richerd, and S. K. Baruah. A response-time bound in fixed-priority scheduling with arbitrary deadlines. *IEEE Trans. Comput.*, 58(2):279–286, 2009.

[13] A. Bouakaz and J.-P. Talpin. Buffer minimization in earliest-first scheduling of dataflow graphs. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*, 2013.

[14] A. Bouakaz, J.-P. Talpin, and J. Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design*, 2012.

[15] G. Buttazzo and M. Di Natale. HARTIK: a hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 404–409, 1993.

[16] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric automata. In *the 29th IEEE Real-Time Systems Symposium*, pages 80–89, 2008.

[17] D. Clark. HIC: an operating system for hierarchies of servo loops. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1004–1009, 1989.

[18] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, 2011.

[19] N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 118–127, 2006.

[20] L. Fribourg, R. Soulat, D. Lesens, and P. Moro. Robustness analysis for scheduling problems using the inverse method. In *19th International Symposium on Temporal Representation and Reasoning*, pages 73–80, 2012.

[21] A. Gamatié. Design of streaming applications on MPSoCs using abstract clocks. In *Design, Automation and Test in Europe Conference*, pages 763–768, 2012.

[22] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[23] JSR-302. *Safety critical Java technology specification*, 2010.

[24] E. A. Lee and E. Matsikoudis. The semantics of dataflow with firing. In G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, chapter 4, pages 71–94. Cambridge University Press, 1 edition, 2009.

[25] E. A. Lee and D. G. Messerchmitt. Static scheduling of synchronous dataflow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, 1987.

[26] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

[28] R. Martí and G. Reinelt. *The linear ordering problem. Exact and heuristic methods in combinatorial optimization*. Berlin: Springer, 2011.

[29] D.-I. Oh and T. P. Baker. Utilization bounds for N-processor rate monotonic scheduling with static processor assignment. *Real-Time Syst.*, 15(2):183–192, 1998.

[30] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[31] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow graphs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 3235–3238, 1995.

[32] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 110–119, 2009.

[33] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety-critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded systems*, pages 95–101, 2010.

[34] SAE Aerospace (Society of Automative Engineers). Aerospace standard as5506a: Architecture analysis and design language (AADL). *SAE AS5506A*, 2009.

[35] M. Schoeberl and J. R. Rios. Safety-critical Java on a Java processor. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and*, pages 54–61, 2012.

[36] M. Sjödin and H. Hansson. Improved response-time analysis calculations. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 399–408, 1998.

[37] I. M. Smarandache, T. Gautier, and P. L. Guernic. Validation of mixed signal-alpha real-time systems through affine calculus on clock synchronisation constraints. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, volume 2, pages 1364–1383, 1999.

[38] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In *Proceedings of the 10th International Workshop on Java Technologies for Real-Time and*, pages 44–53, 2012.

[39] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: a safe programming construct for high-frequency hard real-time applications. *SIGPLAN Not.*, 41(6):283–294, 2006.

[40] J. H. Spring, F. Pizlo, R. Guerraoui, and J. Vitek. Reflexes: abstractions for higly responsive systems. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 191–201, 2007.

[41] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: high-throuput stream progrmming in Java. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 211–228, 2007.

[42] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[43] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[44] S. Stuijk, M. Geilen, and T. Basten. Sdf^3: Sdf for free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design*, pages 276–278, 2006.

[45] D. Tang, A. Plsek, and J. Vitek. Static checking of safety critical Java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 148–154, 2010.

[46] A. Wellings and M. Kim. Asynchronous event handling and safety critical Java. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and*, pages 53–62, 2010.