Conference Paper

Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures

Daniel Casini
Alessandro Biondi
Geoffrey Nelissen*
Giorgio Buttazzo

*CISTER Research Centre
CISTER-TR-180805

2018/12/11
Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures

Daniel Casini, Alessandro Biondi, Geoffrey Nelissen*, Giorgio Buttazzo

*CISTER Research Centre
Polytechnic Institute of Porto (ISEP-IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto
Portugal
Tel.: +351.22.8340509, Fax: +351.22.8321159
E-mail: grrpm@isep.ipp.pt, giorgio@sssup.it
http://www.cister.isep.ipp.pt

Abstract
This work propose solutions for bounding the worst-case memory demand generated by parallel tasks running on multicore platforms with scratchpad memories. The objective is to propose a feasibility test that verifies whether the memories are large enough to contain the maximum memory backlog that may be generated by the system. Both closed-form bounds and more accurate algorithmic techniques are proposed. We show how one can use max-plus algebra and solutions to the max-flow cut problem to efficiently solve the memory feasibility problem. Experimental results are presented to evaluate the efficiency of the proposed feasibility analyses on synthetic workload and state-of-the-art benchmarks.
Memory Feasibility Analysis of Parallel Tasks Running on Scratchpad-Based Architectures

Daniel Casini*, Alessandro Biondi*, Geoffrey Nelissen†, and Giorgio Buttazzo*

Abstract—This work proposes solutions for bounding the worst-case memory space requirement for parallel tasks running on multicore platforms with scratchpad memories. It introduces a feasibility test that verifies whether memories are large enough to contain the maximum memory backlog that may be generated by the system. Both closed-form bounds and more accurate algorithmic techniques are proposed. It is shown how one can use max-plus algebra and solutions to the max-flow cut problem to efficiently solve the memory feasibility problem. Experimental results are presented to evaluate the efficiency of the proposed feasibility analysis techniques on synthetic workload and state-of-the-art benchmarks.

I. INTRODUCTION

Embedded computing platforms are evolving to increase the amount of parallel hardware available in the architecture and this trend is expected to increase in the future. To take advantage of such a feature and improve the system performance, it is essential to express the intrinsic parallel nature of the application code, which can then be exploited in the allocation phase to properly partition the various code segments on the various computing elements.

It is worth observing that the parallel structure of a software application can typically be modelled by a directed acyclic graph (DAG), where nodes represent sequential computations and edges describe precedence relations among them. Such precedence constraints derive from the fact that nodes communicate by exchanging data, which are typically stored in memory buffers. The place where these buffers are allocated (global or local memory) affects the communication performance and, in general, the timing behavior of the whole system.

When these computing platforms are used in safety-critical systems that must react to events generated by the environment, achieving a predictable timing behavior is mandatory for guaranteeing certain levels of safety or performance. A way to increase predictability at a low architecture level is to use scratchpads as local memories, instead of caches [1]. In addition, a non-preemptive execution of nodes helps containing the scratchpad-related delays due to data eviction from other concurrent nodes.

As it is illustrated in Figure 1, in this context, two fundamental problems need to be solved to guarantee a predictable behavior of the application: (1) a schedulability analysis of real-time applications consisting of a set of DAG tasks with non-preemptive nodes, taking into account scratchpad-related delays; (2) a memory analysis that verifies whether the size of scratchpads are enough to contain the maximum memory backlog. Solving the two problems stated above is essential to address the final goal of partitioning applications on a multicore platform, that is, finding a suitable allocation of nodes on the processors that enhances (or possibly optimizes) the system performance.

To the best of our knowledge, most of the works in the literature that addressed the schedulability analysis of multiple DAG tasks did not consider memory requirements, hence the problem of providing a memory feasibility test for these applications is still open.

Memory feasibility is a non-trivial problem in real-time embedded systems. The amount of memory required by each application dynamically varies during the system execution. The amount of memory space required by parallel tasks does not only depend on the nodes executing at each time instant, but also on the pending data transfers between nodes (belonging either to the same or different tasks) executing on the same processor. Tightly bounding the maximum memory space requirement (MSR) is of key importance as it ensures a safe and correct execution of the system, i.e., data transfers may never be corrupted or aborted due to a lack of memory. In addition, it allows for an optimization of the required memory sizes, and hence a global reduction of the platform cost. Furthermore, achieving a detailed analytical understanding of the MSR generated by parallel applications is of paramount importance to develop suitable partitioning algorithms.

Figure 1. Illustration of a design infrastructure to support partitioned real-time tasks upon multiprocessor platforms. A suitable allocation for the task nodes and the memory buffers used by them depends on both schedulability analysis, to guarantee their timing properties, and memory feasibility, to verify memory space requirements. In the figure, LM denotes local memory.
**Contribution.** This work is focused on analyzing the MSR of parallel applications executed under partitioned scheduling, and does not directly consider their timing properties. As mentioned before, timing properties such as the worst-case response time of a task, is a consequence of partitioning and a MSR feasibility analysis is a basic block that is first required to develop suitable partitioning tools. To this end, a real-time parallel task model is first presented in Section II to cope with data exchanges between execution nodes and memory copy-in/copy-out phases to move data from global memory to scratchpads and vice versa.

Then, a memory feasibility analysis is proposed in Section IV, which consists in verifying whether the available memories in a platform are large enough to contain all the outstanding communication buffers at any time (i.e., studying the maximum memory backlog). While Section IV is focused on closed-form bounds on the MSR, Section V shows that an accurate characterization of the MSR can be achieved by means of algorithmic techniques. Specifically, we show that max-plus algebras can be applied to nested fork-join graphs (a practical restriction of DAGs), and that the worst-case MSR analysis for general DAGs can be mapped to a max-flow cut problem and a special case of the maximum weight independent set problem, which can be solved in polynomial time in the case considered in this paper. Finally, Section VI reports on an experimental study that has been conducted to assess the performance of the proposed analysis techniques applied to both synthetic workload and a state-of-the-art benchmarks.

**II. System model.**

This section introduces the platform and task model considered in this paper. The platform model has been inspired by the main properties of multicore execution platforms commonly used in the automotive industry, namely the AURIX TC2xx and AURIX TC3xx [2] family of processors. The execution model considers the characteristics of real applications developed for automotive applications, including precedence constraints, data dependencies, and data transfers between tasks and memories.

1) **Platform model:** The computing platform is assumed to be composed of $M$ identical cores $p_1, \ldots, p_M$. Each core $p_k$ has direct access to a local instruction scratchpad memory $S^i_k$ and a local data scratchpad memory $S^d_k$. A global memory $G$ is shared among all the cores. The data (resp., instruction) scratchpad of the $k$-th core has a size $sz^i_k$ (resp., $sz^d_k$), while the global memory has a size $sz_G$. Memory sizes are expressed in blocks, which is a logical unit corresponding to the memory granularity in the presence of fragmented memory allocation. The actual definition of a block depends on the target system: it can be a memory page [3] or just a custom chunk composed of a given number of bytes.

2) **Execution model:** The system is composed of a set $\Gamma$ of $n$ real-time applications modelled as sporadic tasks, each described as a directed acyclic graph (DAG). Hence, a task $\tau_i = (V_i, E_i)$ is characterized by a set $V_i$ of nodes (or vertices) and a set of directed edges $E_i$. Each task releases a potentially infinite sequence of instances (i.e., jobs). Each of those jobs must execute all nodes in $V_i$ respecting the precedence constraints defined by $E_i$. A task is said to be pending when it has at least one released but uncompleted node. Each job is subject to an inter-job precedence constraint: namely, only one instance of each task can be pending at the same time.

Tasks are executed under partitioned scheduling, where each node executes in a non-preemptive fashion. The $j$-th node of task $\tau_i$ is denoted by $v^i_j \in V_i$ and is statically allocated to core $P(v^i_j)$. A node $v^i_j$ is a sub-task of $\tau_i$ and is characterized by a contention-free worst-case execution time (WCET) $C^i_j$ (occurring when it executes in isolation and all its data and instructions are in the local scratchpads). Each node requires $LM^j_i$ blocks of local memory during its execution (e.g., to store local variables in a stack). The set of tasks that have at least one node allocated to $p_k$ is denoted by $\Gamma_k$.

Nodes are connected by edges. Edge $e^i_{j,z} \in E_i$ connects node $v^i_j$ to node $v^i_z$ of $\tau_i$. It defines a precedence constraint between the two nodes, meaning that $v^i_j$ can start executing only after $v^i_z$ is completed.

Each edge also models the data dependencies between nodes: to this purpose, edges are weighted with (i) the amount of data produced by the source node to be consumed by the destination node and (ii) the type of communication channel being used, namely local or global memory. Formally, an edge $e^i_{j,z} = (m^i_{j,z}, \Delta^i_{j,z})$ is characterized by a weight $m^i_{j,z}$ (expressed in memory blocks) and a communication type $\Delta^i_{j,z} \in \{L, G\}$, where $L$ represents a shared-memory communication with a buffer allocated in local scratchpad, and $G$ represents a communication through a buffer allocated in global memory.

Communications that involve the global memory are subject to copy-in and copy-out phases. That is, if node $v^i_j$ communicates with node $v^i_z$ by means of global memory, then $v^i_j$ first saves the output data in its local scratchpad memory when it executes, and then copies the produced data in global memory at the end of its execution (copy-out); similarly, the destination node $v^i_z$ first copies the input data from global memory to its local scratchpad before execution (copy-in), and then accesses the data in scratchpad without suffering any contention during its execution. Copy-in and copy-out phases are performed by the cores (i.e., they consume CPU cycles). They are modelled separately from the node WCET for the sake of logical clearness. A node is said to be completed after the completion of all its copy-out phases. For local memory transactions, the memory buffer used for a communication modelled by edge $e^i_{j,z}$ is allocated when node $v^i_j$ starts executing and de-allocated when node $v^i_z$ completes its execution. As long as a buffer is allocated, the corresponding communication is said to be pending. Note that this allocation can also follow pre-computed (i.e., static) patterns and involve fragmentation techniques [4] (i.e., a buffer is not contiguously allocated in physical memory).

Edges of $\tau_i$ model the intra-job communications (i.e., the communication between nodes released by the same job). Yet, communication between successive jobs of the same task may be required (e.g., to reuse data computed during the previous iteration of a control loop). To this end, each task $\tau_i$ uses a persistent memory buffer with size $PM^i_i$ statically allocated in each scratchpad $S^d_k$. Nodes have a direct access to those buffers.

---

1 Modelling copy-in and copy-out phases separately from the node WCET may also allow for future extensions, such as modelling the use of DMA engines for data transfers.
to store updated data and read data produced by previous jobs. Communications between successive jobs performed through global memory are modelled by an edge from or to a dummy node with zero execution time. This edge is weighted with the amount of data to transfer and labelled with a communication type $\Delta_{v_i,z}^{\text{trans}} = G$.

The time overhead introduced by memory fragmentation is assumed to be negligible or part of the tasks’ WCET, whereas the corresponding space overhead is assumed to be factored within the memory requirements. Tasks are assumed to be independent, i.e., they do not access shared resources. Furthermore, instruction scratchpads are assumed to be large enough to contain the code of the nodes allocated to the corresponding cores, or each core disposes of a dedicated flash memory (as it is the case for the modern AURIX TC3xx platforms produced by Infineon [2]).

To denote direct precedence relations, for each node the sets of immediate predecessors $\text{ipred}(v_{i,s})$ and immediate successors $\text{isucc}(v_{i,s})$ are defined, as $\text{ipred}(v_{i,s}) = \{v_{i,j} \in V_i : \exists (v_{i,j}, v_{i,s}) \in E_i\}$ and $\text{isucc}(v_{i,s}) = \{v_{i,j} \in V_i : \exists (v_{i,s}, v_{i,j}) \in E_i\}$, respectively.

Analogously, the sets of predecessors $\text{pred}(v_{i,s})$ and successors $\text{succ}(v_{i,s})$ denote precedence relations that are either direct (i.e., by means of an edge) or transitive (i.e., by means of a set of edges involving intermediate nodes).

A node $v_{s,j}$ without incoming edges is referred to as a source node, whereas a node without outgoing edges is referred to as a sink node. For the sake of simplicity, this paper assumes a single sink and source node. Note that, when this assumption does not hold, any DAG with multiple source/sink nodes can always be transformed into a DAG with a single source/sink node by adding an extra dummy source/sink node with computation time equal to zero and data transfers of zero size with their successors/predecessors.

Figure 2 reports a sample schedule of a single task $\tau_i$ with 6 nodes partitioned on two processor cores, together with the plot of the MSR for the scratchpad memory of the second core ($S_2^D$). All communications between nodes allocated to the same core are performed with a shared-memory buffer allocated in the corresponding scratchpad (i.e., $\Delta_{v_i,z}^{\text{trans}} = \Delta_{s,z}^{\text{trans}} = \Delta_{s,z}^{\text{trans}} = \Delta_{s,z}^{\text{trans}} = L$), and hence do not involve copy-in and copy-out phases. Communications that involve nodes allocated on different cores require copy-in and copy-out phases, as illustrated in the schedule.

To help the reader in following the adopted notation, Table I summarizes the symbols introduced in the system model.

### III. RELATED WORK

To the best of our knowledge, no approaches are available to study the worst-case MSR of a parallel application executed upon a multicore platform with scratchpad memories. Previous work strictly related to the problem addressed in this paper focused on deriving MSR analysis and memory allocation algorithms for a set of classical periodic tasks executed upon an uniprocessor. Most relevant to us are the works of Marchand et al. [5], Crespo et al. [6], and Puaut [7]. Mechanisms for predictable data allocation in scratchpads have been studied by Puaut and Pais [4] and Whitham and Audsley [3]. Suhendra et al. [8] proposed an iterative scratchpad allocation algorithm aimed at reducing worst-case response times for periodic tasks under preemptive scheduling. The authors presented different methodologies for providing a scratchpad allocation that reserves partitions to tasks that may preempt each other, while ensuring memory space constraints.

Furthermore, note that this work also lies in the intersection of research contributions on parallel real-time tasks and data-flow models, and memory-aware execution models. Due to the large number of available results and space constraints, a detailed literature review cannot be reported here. Therefore, this section is focused on reviewing the papers that are much closer to the present work or representative for the corresponding research domain.

---

**Table I. TABLE OF SYMBOLS**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_k$</td>
<td>$k$-th processor</td>
</tr>
<tr>
<td>$s_k^D$</td>
<td>size of $k$-th data scratchpad</td>
</tr>
<tr>
<td>$S_2^D$</td>
<td>$k$-th data scratchpad</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>set of tasks with at least one node allocated on $p_k$</td>
</tr>
<tr>
<td>$\tau_i$</td>
<td>$i$-th task</td>
</tr>
<tr>
<td>$v_i^j$</td>
<td>$j$-th node of $\tau_i$</td>
</tr>
<tr>
<td>$C_i^j$</td>
<td>WCET of $v_i^j$</td>
</tr>
<tr>
<td>$LM_i^j$</td>
<td>size of local memory of node $v_i^j$</td>
</tr>
<tr>
<td>$PM_i^j$</td>
<td>size of inter-job persistent memory of $\tau_i$ on $p_k$</td>
</tr>
<tr>
<td>$\mathcal{P}(v_i^j)$</td>
<td>core in which $v_i^j$ is allocated to</td>
</tr>
<tr>
<td>$\text{ipred}(v_i^j)$</td>
<td>immediate predecessors of $v_i^j$</td>
</tr>
<tr>
<td>$\text{isucc}(v_i^j)$</td>
<td>immediate successors of $v_i^j$</td>
</tr>
<tr>
<td>$\text{pred}(v_i^j)$</td>
<td>predecessors of $v_i^j$</td>
</tr>
<tr>
<td>$\text{succ}(v_i^j)$</td>
<td>successors of $v_i^j$</td>
</tr>
<tr>
<td>$e_{i,z}^{j,j}$</td>
<td>edge connecting $v_i^j$ to $v_i^j$</td>
</tr>
<tr>
<td>$m_{i,z}^{j,j}$</td>
<td>amount of data produced by $v_i^j$ for $v_i^j$</td>
</tr>
<tr>
<td>$\Delta_{i,z}^{j,j}$</td>
<td>communication type (G or L) related to $e_{i,z}^{j,j}$</td>
</tr>
</tbody>
</table>
A. Parallel tasks and data-flow models

During the last decade, several authors addressed the problem of analyzing parallel tasks running upon multiprocessor platforms under real-time constraints. Both the fork-join task model [9], [10] and the DAG task model [11]–[13] have been studied to deal with parallel applications scheduled under global scheduling, whereas Fonseca et al. [14] and Casini et al. [15] studied the same models under partitioned fixed-priority preemptive and non-preemptive scheduling, respectively. Such models are different from the one presented in the previous section as they do not consider data dependencies between execution nodes and the corresponding MSR.

The explicit flow of data among nodes has been considered in different variations of the dataflow task model that have been proposed over the years. For instance, the Synchronous DataFlow Graph model [16], [17] represents a graph with producer-consumer relations among nodes (also called actors) connected through edges. Each edge represents a FIFO queue, used to direct tokens (i.e., data) among nodes, and it is characterized by three quantities, all of them representing a number of tokens. Specifically, the number of tokens inserted into the queue by a producer, the number removed by a consumer, and the number initially present in the queue.

A different dataflow model, adopted to describe the workload of industrial cellular networked systems scheduled on a heterogeneous platform, has been proposed by Dong et al. [18]. In their work, the authors aim at providing response time bounds without incurring capacity loss for a parallel task model, where each task consists in chain of subtasks with an implicit deadline and it is designated to be executed on a specific type of processor.

Closer to our work, the work by Elliott et al. [19] considers a dataflow model in which each edge is characterized by the number of bytes that a predecessor (i.e., the producer) is producing for a successor (i.e., the consumer). However, they adopted clustered multiprocessor scheduling and a platform model consisting of a cache hierarchy. The authors proposed a timing-driven heuristic assignment designed to make an efficient use of the cache memory. The MSR of the tasks has not been analyzed.

B. Memory-aware execution models

Concerning memory-aware execution models, Pellizzoni et al. [20] proposed the PREdicatable Execution Model (PREM), in which the execution of each task consists of two phases: a memory phase and an execution phase. In the memory phase the data needed by the task is preloaded into a local memory, thus avoiding the need for accessing the shared memory during the execution phase. The PREM execution model was originally conceived for uniprocessor platforms, and later extended to multicores [21]. Other authors considered similar execution models with three phases (copy-in, execution, and copy-out): this is the case of the works of Alhammad et al. [22], [23] and Maia et al. [24].

Kim and Rajkumar [25] focused on implementing a memory reservation scheme to trade the MSR of tasks with timing latencies. Tabish et al. [26], Soliman and Pellizzoni [27], and Biondi and Di Natale [28] considered techniques to preload scratchpad memories similarly as considered in this paper. Finally, Irobi and Juurlink [29] proposed three different strategies for allocating data in scratchpad memories aimed at guaranteeing the schedulability of implicit-deadline periodic tasks on a single core, where computation times are dependent on the amount of memory allocated in the scratchpad.

IV. Memory feasibility

This section presents a memory feasibility analysis for the considered system model. To better illustrate the problem addressed here, a simple motivational example is shown in Section IV-A. Then, MSR bounds are proposed in Section IV-B.

A. Motivational example

Consider the sample parallel task illustrated in Figure 2 scheduled on two cores $p_1$ and $p_2$ (no other tasks are present). As the task progresses in its execution, memory buffers are allocated and de-allocated from memories by following the task model introduced in Section II-2. Once a buffer is allocated, the MSR for a given memory increases, and it decreases when a buffer is released. Figure 2 also illustrates a possible schedule of the task on $p_1$ and $p_2$, together with the corresponding MSR for $S_2$ over time. Note that initially the MSR is zero, and it is increased when node $v_3$ starts executing by allocating (i) all the buffers corresponding to incoming and outgoing edges (for a total of nine memory blocks), and (ii) the local memory of the node (assumed to have size of one block). Once $v_3$ terminates, only the buffers related to intra-core communications (specifically, those related to $v_4$ and $v_5$) are kept in $S_2$, for a total of six memory blocks. However, as $v_4$ directly starts executing, the MSR is increased to eleven to account for (i) the copy-in data originating from $v_2$ (one memory block), (ii) the buffer size of outgoing communications (three memory blocks) and (iii) the local memory requirement of the node (one memory block). Summing all those contributions with the pending communications originating from $v_3$, the MSR reaches 11, its maximum value during the execution of the task.

Clearly, when considering more complex DAG structures and more than one task, the problem of computing the maximum demand generated by a task set becomes non-trivial. The objective of this section is to compute suitable upper bounds for the MSR on each core, hence verifying whether the memories are large enough to contain the maximum memory backlog produced by a task set.

B. MSR bounds

This section aims at deriving closed-form bounds on the MSR generated on each core $p_k$ (with $k = 1, \ldots, M$), hence providing a memory feasibility test. This section only considers the memory feasibility problem with respect to data scratchpads, which are typically the scarce resources (an analysis for the global memory can be performed with techniques similar to those presented in this section). In the following, a top-down approach is adopted, where the maximum MSR is increasingly decomposed into multiple terms.

Given a core $p_k$, at any point in time, either a single node $v_j$ of $\tau_i$ is executing, or none of $\tau_i$’s nodes is. From this basic property, two different MSRs are derived for $\tau_i$ on $p_k$, depending on the case that holds:
(i) \( M_{k}^{\text{EX}} \) is a bound on the maximum MSR in data scratchpad \( S_{k}^{0} \) generated by \( \tau_{i} \) in the case one of its nodes executes on \( p_{k} \); and

(ii) \( M_{k}^{\text{INTRA}} \) is a bound on the maximum MSR in data scratchpad \( S_{k}^{0} \) generated by \( \tau_{i} \) in the case none of its nodes executes on \( p_{k} \).

Since the additional local memory space required by a node when it executes (e.g., to save local variables on a stack) is freed when the node completes, it clearly holds that \( M_{k}^{\text{EX}} \geq M_{k}^{\text{INTRA}} \), \( \forall \tau_{i} \in \Gamma_{k} \).

At any time instant \( t \), the total MSR on core \( p_{k} \) is given by the sum of the MSR of the one task executing on \( p_{k} \) at time \( t \) (if any), and the MSR of all the other tasks in \( \Gamma_{k} \) that are not executing on \( p_{k} \) at time \( t \). Building upon this principle, Lemma 1 establishes a (sufficient) memory feasibility test for core \( p_{k} \).

**Lemma 1:** The nodes running on core \( p_{k} \) are memory-feasible if

\[
\max_{\tau_{i} \in \Gamma_{k}} \left\{ M_{k}^{\text{EX}} + \sum_{\tau_{j} \in \Gamma_{k} \setminus \tau_{i}} M_{k}^{\text{INTRA}} \right\} \leq s z_{k}^{0} - \sum_{\tau_{i} \in \Gamma_{k}} P M_{k}.
\]

**Proof:** At any time instant \( t \), core \( p_{k} \) can be either (i) executing a task \( \tau_{i} \), or (ii) idle.

**Case (i).** The MSR in the data scratchpad at time \( t \) is given by the demand of the executing task \( \tau_{i} \) plus the memory contribution of all the non-executing tasks (i.e., the tasks in \( \Gamma_{k} \setminus \tau_{i} \)) that is, the MSR at time \( t \) is upper-bounded by \( M_{k}^{\text{EX}} + \sum_{\tau_{j} \in \Gamma_{k} \setminus \tau_{i}} M_{k}^{\text{INTRA}} \). Since any task \( \tau_{i} \) in \( \Gamma_{k} \) can be executing at time \( t \), the maximum value of the previous equation over all tasks in \( \Gamma_{k} \) yields a safe upper-bound on the MSR when a task executes on \( p_{k} \).

**Case (ii).** If no task execute on \( p_{k} \) at a time instant \( t \), we note that the total MSR in the data scratchpad at time \( t \) is upper-bounded by \( \sum_{\tau_{j} \in \Gamma_{k}} M_{k}^{\text{INTRA}} \). Observing that \( M_{k}^{\text{EX}} \geq M_{k}^{\text{INTRA}} \), \( \forall \tau_{i} \in \Gamma_{k} \), then this bound is always smaller than the one considered in case (i).

The lemma follows by noting that \( s z_{k}^{0} = \sum_{\tau_{i} \in \Gamma_{k}} P M_{k} \) expresses the amount of memory in the data scratchpad \( S_{k}^{0} \) of \( p_{k} \) that is actually available to the nodes during the system execution, which is given by the total size \( s z_{k}^{0} \) of the scratchpad minus the size of all inter-job persistent buffers permanently allocated in local memory.

The problem of memory feasibility is now reduced to finding suitable values for the terms \( M_{k}^{\text{EX}} \) and \( M_{k}^{\text{INTRA}} \).

**Computing \( M_{k}^{\text{EX}} \).**

Note that, when a task \( \tau_{i} \) is executing on core \( p_{k} \), there is exactly one node \( v_{j}^{i} \in V_{i} \) that is executing on \( p_{k} \). This allows us to further decompose \( M_{k}^{\text{EX}} \) into two mutually-exclusive contributions:

(i) The maximum memory amount \( m_{j,k}^{i,\text{ISO}} \) used by node \( v_{j}^{i} \) in \( S_{k}^{0} \) when it is executing on \( p_{k} \);

(ii) The maximum memory amount \( m_{j,k}^{i,\text{INTRA}} \) used in \( S_{k}^{0} \) by pending communications originated from nodes in \( V_{j} \setminus \{v_{j}^{i}\} \) when node \( v_{j}^{i} \) is executing.

Intuitively, \( m_{j,k}^{i,\text{ISO}} \) accounts for the MSR of \( v_{j}^{i} \) as it were executing in isolation (i.e., along on \( p_{k} \), while \( m_{j,k}^{i,\text{INTRA}} \) accounts for the intra-task MSR during the execution of \( v_{j}^{i} \). Thanks to these definitions, the MSR \( M_{k}^{\text{EX}} \) of a task \( \tau_{i} \) executing on \( p_{k} \) can be computed by considering all the nodes of \( \tau_{i} \) that may execute on \( p_{k} \), i.e.,

\[
M_{k}^{\text{EX}} = \max_{v_{j}^{i} \in V_{i}} \left\{ m_{j,k}^{i,\text{ISO}} + m_{j,k}^{i,\text{INTRA}} \right\}.
\]

To proceed, it is convenient to make a simple observation.

**Observation 1:** Let \( IE_{j}^{i} \) and \( OE_{j}^{i} \) be the sets of incoming and outgoing edges of node \( v_{j}^{i} \), respectively. Since no particular assumptions are made on the code structure of the nodes, when \( v_{j}^{i} \) executes, it can arbitrarily read from and write in the buffers related to edges in \( IE_{j}^{i} \) and \( OE_{j}^{i} \) in an interleaved manner. This means that all the communications corresponding to \( IE_{j}^{i} \) and \( OE_{j}^{i} \) must be considered pending during the execution of \( v_{j}^{i} \).

The observation above allows deriving the following lemma to compute \( M_{j,k}^{i,\text{ISO}} \).

**Lemma 2:** The maximum amount of memory requested by node \( v_{j}^{i} \) in \( S_{k}^{0} \) when it is executing on \( p_{k} \), is equal to

\[
M_{j,k}^{i,\text{ISO}} = LM_{j}^{i} + \sum_{v_{j}^{i} \in \text{pred}(v_{j}^{i})} m_{j,k}^{i,j} + \sum_{v_{j}^{i} \in \text{succ}(v_{j}^{i})} m_{j,k}^{i,s}.
\]

**Proof:** Following Observation 1, the amount of memory accessed by \( v_{j}^{i} \) in \( S_{k}^{0} \) is composed of three terms: (i) the node-local MSR (of size \( LM_{j}^{i} \)), (ii) the total amount of memory requested to save the input data received from immediate predecessors (i.e., \( \sum_{v_{j}^{i} \in \text{pred}(v_{j}^{i})} m_{j,k}^{i,j} \)), and (iii) the total amount of memory requested to save the output data produced by \( v_{j}^{i} \) for immediate successors (i.e., \( \sum_{v_{j}^{i} \in \text{succ}(v_{j}^{i})} m_{j,k}^{i,s} \)). Note that, even if input data come from global memory, they must be counted in (ii) as those data must be copied in \( S_{k}^{0} \) before \( v_{j}^{i} \) may access them, hence increasing \( v_{j}^{i} \)’s MSR (see execution model in Sec. II-2). Similarly, data produced by \( v_{j}^{i} \) and pushed in global memory must be counted in (iii) as those data must first be saved in scratchpad \( S_{k}^{0} \) before being pushed in global memory.

It remains to compute \( M_{j,k}^{i,\text{INTRA}} \). Unfortunately, the precedence constraints of a task originate a potentially large number of mutually-exclusive execution scenarios, thus making a precise computation of \( M_{j,k}^{i,\text{INTRA}} \) particularly challenging. The following lemma establishes a closed-form upper bound on \( M_{j,k}^{i,\text{INTRA}} \), while more complex techniques to derive tighter bounds are detailed in Section V.

**Lemma 3:** It holds that

\[
M_{j,k}^{i,\text{INTRA}} \leq \sum_{(e_{s,d}, d) \in \{E_{i}\} \setminus \{X \cup Y \cup Z\}} m_{s,d}^{i}
\]

where

\[
X = \{e_{s,d} \in E_{i} | v_{j}^{i} \in \text{pred}(v_{j}^{i}) \wedge v_{i}^{d} \in \text{pred}(v_{j}^{i}) \cup v_{j}^{i}\} \}
\]

\[
Y = \{e_{s,d} \in E_{i} | v_{j}^{i} \in \text{succ}(v_{j}^{i}) \cup v_{j}^{i}\} \}
\]
and

$$Z = \{ e_{s,d} \in E_i | \Delta_{s,d} = G \vee P(v^i_j) = P(v^i_d) \neq p_k \}.$$  

Proof: Since tasks are subject to an inter-job precedence constraint, at most one instance of \( \tau_i \) can be pending when \( v^i_j \) is executing. Therefore, \( M^{i,\text{INTRA}}_{j,k} \) is upper-bounded in any case by \( \sum_{e_{s,d} \in E_i} m^i_{j,s,d} \). Note now, that, when \( v^i_j \) is executing, local communications originated from other nodes of \( \tau_i \) that are pending and hence request memory space in the local scratchpad \( S^i_k \) cannot be related to edges that:

(i) are on a path reaching \( v^i_j \), as they must already be completed (or are being accessed by \( v^i_j \) itself and are therefore already accounted for in \( M^{i,\text{INTRA}}_{j,k} \));

(ii) are outgoing from successors of \( v^i_j \), as the latter must first complete for those to execute;

(iii) are outgoing from \( v^i_j \) itself as those are already being accounted for in \( M^{i,\text{INTRA}}_{j,k} \);

(iv) perform their communications through global memory (i.e., edges for which \( \Delta_{s,d} = G \), as due to the non-preemptive execution model assumed in this work, the execution of other nodes of \( \tau_i \) must have already been completed or have not yet started. Therefore, global data accessed by those nodes must already have been copied out of the scratchpad or have not yet been copied in the scratchpad;

(v) perform local communications in other scratchpad memories (i.e., edges for which \( P(v^i_j) = P(v^i_d) \neq p_k \)).

Set \( X \) accounts for the edges of case (i) by collecting all edges that connect two predecessors of \( v^i_j \) or a predecessor of \( v^i_j \) and \( v^i_j \) itself; all those edges belong to a path ending in \( v^i_j \). Set \( Y \) accounts for the edges of case (ii) and (iii), while set \( Z \) jointly accounts for cases (iv) and (v). Since only edges from those sets are excluded from \( E_i \) in Eq. (2), the lemma follows.  

\[ \text{Computing } M^{i,\text{LEX}}_{j,k} \]

In this section we derive an upper-bound on the maximum MSR of a task \( \tau_i \) in the data scratchpad \( S^i_0 \) when \( \tau_i \) is not executing on the corresponding core \( p_k \).

Clearly, if \( \tau_i \) never executed on \( p_k \), there cannot be any outstanding communications between nodes of \( \tau_i \) and \( M^{i,\text{LEX}}_{j,k} = 0 \). We are therefore interested to the case where at least one node of \( \tau_i \) executed on \( p_k \). Let \( v^i_j \) be the last such node. We define \( M^{i,\text{LEX}}_{j,k} \) as the maximum amount of memory allocated in \( S^i_0 \) for outgoing communications of \( \tau_i \) when \( \tau_i \) is not executing and \( v^i_j \) is the last node of \( \tau_i \) that executed.

From this definition, \( M^{i,\text{LEX}}_{j,k} \) can be obtained from \( M^{i,\text{INTRA}}_{j,k} \) by considering all nodes of \( \tau_i \) allocated to \( p_k \). Formally,

$$M^{i,\text{LEX}}_{j,k} = \max_{v^i_j \in \tau_i} \left\{ M^{i,\text{INTRA}}_{j,k} \right\}. \quad (3)$$

Finally, the following lemma establishes the value of \( M^{i,\text{LEX}}_{j,k} \).

Lemma 4: The maximum amount of memory allocated in \( S^i_k \) for pending communications of \( \tau_i \), when \( \tau_i \) is not executing and \( v^i_j \) is the last node of \( \tau_i \) that executed on \( p_k \), is equal to

$$M^{i,\text{LEX}}_{j,k} = M^{i,\text{INTRA}}_{j,k} + \sum_{v^i_j \in \text{succe}(v^i_j)} \Delta_{j,s} = L.$$  

Proof: When \( v^i_j \) is the last node of \( \tau_i \) that executed on \( p_k \), the MSR generated by \( \tau_i \) in \( S^i_k \) is given by the MSR when \( v^i_j \) was executing minus the amount of memory \( x \) that \( v^i_j \) de-allocated after its completion, that is, \( M^{i,\text{LEX}}_{j,k} = M^{i,\text{INTRA}}_{j,k} + M^{i,\text{ISO}}_{j,k} - x \). When it terminates, \( v^i_j \) de-allocates its local memory \( M^i_j \), the buffers related to incoming edges, and the buffers related to outgoing edges whose content has been copied-out into the global memory (i.e., edges \( e_{s,d} \) with \( \Delta_{s,d} = G \)). By merging Lemma 2 with this observation, we have that \( M^{i,\text{LEX}}_{j,k} = \sum_{v^i_j \in \text{pred}(v^i_j)} M^i_{j,j} - \sum_{v^i_j \in \text{succe}(v^i_j)} \Delta_{j,s} = G = \sum_{v^i_j \in \text{succe}(v^i_j)} \Delta_{j,s} = L \). The lemma follows.

V. ACCURATE CHARACTERIZATION OF THE MSR

While the previous section provided closed-form bounds for the MSR generated by a task set, this section focuses on algorithmic approaches for computing tighter bounds, specifically by deriving an exact value for \( M^{i,\text{LEX}}_{j,k} \).

First, Section V-A shows that max-plus algebra related techniques can be used to solve the problem in polynomial time for the case where the tasks are modeled by nested fork-join graphs [30], i.e., a more restricted version of DAGs that found practical application in several parallel programming frameworks. Then, Section V-B shows that the problem of computing \( M^{i,\text{LEX}}_{j,k} \) for a general DAG can be mapped to a maximum weight independent set problem [31] and presents two solutions to solve it: the first one is based on an integer linear programming formulation, while the second is a polynomial-time algorithm using the notion of comparability graphs.

A. Computing \( M^{i,\text{LEX}}_{j,k} \) for nested fork-join tasks

To make this paper self-contained, the definition of a nested fork-join (NFJ) graph is recalled. We first define a “fork-join chunk” as a basic block of nested fork-join graph.

Definition 1: A fork-join chunk is a DAG \((V, E)\) where (i) there exists a single source node \( v_s \in V \) and a single termination (sink) node \( v_t \in V \); (ii) if \( V = \{v_s, v_t\} \) then \( v_s \) is directly connected to \( v_t \) (potentially with multiple directed edges); (iii) otherwise, for each node \( v \in V \setminus \{v_s, v_t\} \) there exists a single two-edges path from \( v_s \) to \( v_t \) that traverses \( v \), i.e., \( v_s \) is connected to \( v \) and \( v \) is connected to \( v_t \).

A nested fork-join graph can then be recursively defined by using the notion of fork-join chunk as the base case.

Definition 2: A nested fork-join graph is a fork-join chunk; or it is a DAG resulting from a nested fork-join graph in which at least one node \( v \) is replaced by a nested fork-join graph with source and termination nodes \( v^*_s \) and \( v^*_t \), such that all incoming edges of \( v \) are connected to \( v^*_s \) and all outgoing edges of \( v \) are outgoing from \( v^*_t \).

A simple example of a nested fork-join graph composed of a single fork-join chunk is illustrated in Figure 3(a).
Using Definition 2, a nested fork-join parallel task is defined as any other parallel task (see Section II-2) but enforcing that the DAG \((V_i, E_i)\) complies with the definition of a nested fork-join graph.

For the purpose of MSR analysis, in this section we prove that NFJ graphs can be transformed into equivalent (i.e., with identical MSR), but simpler graphs by recursively applying two simple rules. Such rules are essentially a max-plus algebra applied to graphs and are formally stated in Lemmas 5 and 6.

Consider the particular case of a linear DAG defined as:

**Definition 3:** A DAG \(G = (V, E)\) is said to be linear if there exists an ordered sequence \(s\) of the nodes in \(V\) such that the edges in \(E\) sequentially connect the nodes according to \(s\) and \(|E| = |V| - 1\).

An example of a linear graph is shown on the top of Figure 3(b). Note that a linear DAG does also respect the definition of a fork-join graph.

By using the rule stated in the following lemma, a linear DAG can be transformed into a simpler graph containing only one directed edge and that has the same worst-case MSR than the original DAG.

**Lemma 5:** Consider a task \(\tau_i\) modelled by a linear DAG \(G = (V_i, E_i)\) and let \(v_p \in V_i\) and \(v_q \in V_i\) be the source and termination nodes of \(G\), respectively. The maximum MSR generated by \(\tau_i\) when none of its nodes are executing is equivalent to the worst-case MSR generated by a task \(\tau_i'\) described by a two-node DAG connected by a single edge with weight \(\max\{m_{j,z}\}\).

**Proof:** When none of the nodes in the graph \(G\) is executing, the MSR is only generated by the pending communications between nodes, which are represented by the edges of \(G\). Since the graph is linear, due to precedence constraints all the communications are mutually exclusive in time, i.e., only one of them can be pending at a given point in time. Hence, the maximum MSR generated by the task is given by the maximum weight on the edges of \(G\), that is, \(\max\{m_{j,z}\}\), hence the claim.

The transformation described in Lemma 5 is illustrated in Figure 3(b), where it is applied to the set of nodes and edges belonging to the upper-most branch of the NFJ graph presented in Figure 3(a). When Lemma 5 is applied to all branches of the NFJ graph of Figure 3(a), one obtains the DAG shown on the top part of Figure 3(c).

**Lemma 6:** Consider a task \(\tau_i\) described by a DAG \(G = (V_i, E_i)\) in which there exists a set \(E^* \subseteq E_i\) of edges with \(|E^*| > 1\), where all edges in \(E^*\) connect the two same nodes \(v_1 \in V_i\) and \(v_2 \in V_i\). The maximum MSR generated by \(\tau_i\) when none of its nodes are executing is equal to the maximum MSR generated by a task \(\tau_i'\) modelled by a DAG \(G'\) obtained from \(G\) by replacing the edges in \(E^*\) with a single edge connecting \(v_1\) to \(v_2\) and with weight \(\sum_{e_{j,z} \in E^*} m_{j,z}\).

**Proof:** According to the semantic of the edges stated in Section II-2, a set of at least two edges that connect \(v_1\) to \(v_2\) implies a precedence constraint between \(v_1\) and \(v_2\), and the weight of those edges specifies the amount of data produced by \(v_1\) for \(v_2\). The lemma follows by noting that the same semantic is preserved by collapsing such edges in a single edge weighted with the sum of their weights.

Figure 3(c) illustrates the transformation described in Lemma 6.

By repeatedly applying Lemmas 5 and 6, any NFJ graph \(G\) modelling a task \(\tau_i\) can be reduced to a two-nodes graph with a single edge weighted with the maximum intra-task MSR of \(\tau_i\). This claim is exemplified in Figure 3 and can be formally proved with an inductive argument by introducing the notion of nesting level of a NFJ graph.

**Definition 4:** A NFJ graph with nesting level \(k\) is a graph that can be obtained by replacing each node of a fork-join chunk with a NFJ sub-graph with nesting level at most \(k-1\), where nesting level \(k = 0\) corresponds to a linear graph.

**Lemma 7:** The repetitive application of Lemma 5 and 6 on a NFJ graph results in a two-node graph connected by a single edge.

**Proof:** The lemma is proved by structural induction.

**Base case:** Consider a NFJ graph \(G\) with nesting level \(k = 0\), i.e., a linear graph. By applying Lemma 5, to \(G\), \(G\) is reduced to a two-nodes graph connected by a single edge.

**Inductive case:** Assume that a NFJ sub-graph with nesting degree \(\leq k\) can be reduced to a two-nodes graph connected by a single edge. Now, consider a NFJ graph with nesting degree \(k+1\), which by definition is composed of a number of sub-graphs with nesting level \(\leq k\) (see Fig. 4(a)).

By induction hypothesis, each sub-graph of level \(\leq k\) in \(G\) can be reduced to a two-node graph connected by a single edge. Therefore, \(G\) can be reduced to a fork-join chunk where each node is replaced by one of such two-node sub-graphs (see Fig. 4(b)). Consider first the sub-graphs that replace one of the inner nodes of a fork-join chunk: by applying Lemma 5 to each parallel path, they can be replaced by a single edge (see Fig. 4(c)). All edges resulting from this step can be merged into a single edge using Lemma 6 (see Fig. 4(d)). Then, the resulting graph is linear and can be transformed into a two-node graph by applying Lemma 5 again.

Leveraging the result of Lemma 7, Algorithm 1 shows how Lemmas 5 and 6 can be used to compute the term \(M_{j,k}^{\text{INTRA}}\).

The algorithm takes as input the graph \((V_i, E_i)\) modelling the task under analysis, a node \(v_j^i \in V_i\) and the core \(p_k\) on which \(v_j^i\) executes. First, leveraging the results of Lemma 3, the weights of all the edges whose corresponding inter-node communications cannot contribute to the intra-task MSR of \(\tau_i\) when \(v_j^i\) executes on \(p_k\) are set to zero, i.e., the null element of max-plus algebras. Then, Lemma 5 and Lemma 6 are
We first define the notion of a flow-cut for a DAG. We then use that notion to compute $M^{\text{INTRA}}_{j,k}$.

**Definition 5:** A flow-cut of a DAG $G = (V, E)$ is a separation of the nodes in $V$ into two disjoint sets $V_1$ and $V_2$, with $V_1 \cup V_2 = V$, such that for all nodes $v \in V_2$, $\text{succ}(v) \cap V_1 = \emptyset$.

The following theorem establishes that the pending communications of a task when one of its nodes executes can be represented with a flow-cut.

**Theorem 1:** Let $v^j_i \in V_i$ be a node of task $\tau_i$ that is executing on core $p_k$. Let $G'$ be the DAG obtained from $(V_i, E_i)$ by setting the weight of all edges in sets $X$, $Y$ and $Z$ to 0 (where $X$, $Y$ and $Z$ are defined as in Lemma 3). The intra-task MSR of $\tau_i$ in $S^0_k$ can be represented as a flow-cut of the graph $G'$.

**Proof:** As already expressed in Lemma 3, edges in $X$, $Y$ and $Z$ do not generate intra-task MSR in the local scratchpad $S^0_k$ when $v^j_i$ executes. Therefore, setting their weight to 0 is safe.

Now, let $V^*$ be equal to $V_i \setminus v^j_i$. Due to the non-preemptive execution policy assumed in this paper, when $v^j_i$ is executing, each node in $V^*$ can either (i) be completed, or (ii) did not yet start executing. Now, consider a flow-cut $(V_1, V_2)$ of DAG $G'$. Nodes falling in case (i) are placed in $V_1$, while nodes in case (ii) are placed in $V_2$. By recalling Definition 5, this is a valid cut as, due to precedence constraints, the successors of nodes that are not yet executed at time $t$ cannot be completed at time $t$. Furthermore, $v^j_i$ can be placed into $V_2$ as it did not complete yet at time $t$. The theorem follows.

With the above theorem in place, the problem of computing $M^{\text{INTRA}}_{j,k}$ can be reduced to the problem of finding the maximal flow-cut$^2$ in a DAG where a maximal flow-cut is defined below.

**Definition 6:** A maximal flow-cut of a DAG is a flow-cut $(V_1, V_2)$ such that the sum of the weights of the edges connecting nodes in $V_1$ to nodes in $V_2$ is maximal.

This problem can be solved with an ILP. Let $x^i_{s,d} \in \{0, 1\}$ be a binary variable defined for each edge in $E_i$ such that $x^i_{s,d} = 1$ iff $e^i_{s,d}$ is an edge traversed by the flow-cut. The ILP is then formulated as follows:

**An ILP based solution**

repeatedly applied until the graph is reduced to a two-node graph with a single edge (Lines 6 and 7). Specifically, the pseudo-statement at line 6 applies the transformation specified by Lemma 5 to all the linear sub-graphs in $(V_i, E_i)$, while the one at line 7 applies the transformation specified by Lemma 6 to all pairs of nodes in $(V_i, E_i)$ that are connected by more than one edge.

Note that whenever the two lemmas are applied, the number of edges in $E_i$ is reduced by at least one. The number of iterations in the while loop is therefore bounded by the number of edges in the graph. Further, each iteration may require to go once through the whole graph (i.e., once through all edges). Therefore, the complexity of the algorithm is implicitly bounded by the number of edges, i.e., it is upper-bounded by $O(|E_i|^2)$. Finally, it is worth mentioning that a recursive implementation of the algorithm with linear complexity $O(|E_i|)$ is also possible.

**B. Computing $M^{\text{INTRA}}_{j,k}$ for DAG tasks**

The very efficient solution presented in the previous section cannot be applied to the general case in which a task is described by an arbitrary DAG. Nevertheless, in the following, we demonstrate that the computation of $M^{\text{INTRA}}_{j,k}$ can be seen as a variant of the max-flow problem over DAGs. We propose two solutions to solve this problem: an Integer Linear Programming (ILP) formulation and a polynomial time algorithm that solves the maximum-weight independent set problem for an equivalent comparability graph [31].
A solution based on comparability graphs

Considering the definition of a flow-cut (Definition 5), two different edges $e_x$ and $e_y$ can be cut by the same flow-cut if and only if they do not belong to the same path in the graph. That is, there is no precedence constraints (direct or transitive) between $e_x$ and $e_y$. Precedence constraints between edges of a DAG can be represented by a comparability graph. We first recall the definition of a comparability graph.

Definition 7: Let $S$ be a strictly partially ordered set. A comparability graph $C$ of $S$ is an undirected graph whose nodes are the elements in $S$ such that any two node $x$ and $y$ in $C$ are connected if $x < y$ or $y < x$ in $S$.

The set of edges $E$ of a DAG $G = (V, E)$ can be seen as a strictly partially ordered set such that for every pair $(e_x, e_y)$ of edges in $E$, $e_x < e_y$ if $e_x$ must be "executed" before $e_y$. Therefore, $E$ can be represented as a comparability graph $C$. Figure 5 shows an example of a DAG $G$ and the comparability graph $C$ built from the precedence constraints between the edges of $G$.

From the discussion above, it results that the comparability graph $C$ has the following useful property:

Property 1: Let $C$ be the comparability graph obtained from a set of edges $E$ of a DAG $G = (V, E)$. Any two nodes $e_x$ and $e_y$ in $C$ are not connected by an edge in $C$ if there is no precedence constraint between $e_x$ and $e_y$ in $G$.

From that property and recalling Definition 5, it holds that the set of edges cut by a flow-cut in a DAG $G = (V, E)$ is an independent set of nodes in the comparability graph $C$ obtained from $E$. Figure 5 shows that the flow-cut composed of edges $\{e_{2,4}, e_{1,3}\}$ in $G$ is equivalent to an independent set (i.e., no edge connects its elements) composed of the two red nodes in the comparability graph $C$.

Now, assume that each node in the comparability graph $C$ obtained from $G = (V, E)$ is weighted with the weight $m_{j,z}$ of the corresponding edge in $E$, then we have the following theorem.

Theorem 2: The maximum flow-cut of $G = (V, E)$ cuts the edges in $E$ that corresponds to the maximum-weight independent set of the comparability graph $C$ obtained from $G$, where each node in $C$ is weighted with the weight of the corresponding edge in $E$.

Proof: Thanks to Property 1, any independent set in $C$ is a valid flow-cut in $G$ and vice-versa.

Further, if the independent set is the maximum-weight independent set in $C$, then the sum of the weights of the nodes in the independent set is maximal, that is, there is no other independent set such that the sum of the weights is larger. Since the weights of the nodes in $C$ are the weight of the edges in $G$, this means that there is no other flow-cut of $G$ such that the sum of the weights on the cut edges is larger. Therefore, the set of edges in the maximum-weight independent set of the comparability graph $C$ form a maximum flow-cut of $G$.

Note that generating a comparability graph can be performed in polynomial time. Moreover, besides being an NP-hard problem for general graphs, the maximum-weight independent set for a comparability graph can also be computed in polynomial time [33]. Therefore, the computation of the maximum intra-task MSR $M^{\text{intra}}_{j,k}$ can be done in polynomial time. Algorithm 2 summarizes the approach for calculating $M^{\text{intra}}_{j,k}$.

While this solution provides considerable benefits in terms of computational complexity, it provides less flexibility for modelling other constraints related to future model extensions in comparison to the ILP-based approach. It is however more scalable against the number of nodes and edges in the graph.

The solution devised in Section V-A for NFI tasks suffers from the same limitations and advantages than the comparability graph approach presented here, but it allows for a very efficient implementation with a complexity linear in the number of edges. The solution of Section V-A is therefore more appropriate when a task is known to be NFI.

VI. EXPERIMENTAL RESULTS

This section reports the results of two different experimental studies that have been conducted to evaluate the approaches presented in this paper. Both are aimed at empirically assessing how the memory demand varies with respect to: (i) the adopted strategy for computing the terms $M^{\text{intra}}_{j,k}$ (the closed-form bound presented in Section IV or the algorithmic approaches presented in Section V), and (ii) how nodes are partitioned. The two studies differ by the type of workload used for the evaluation: in the first one we generated synthetic task sets, whereas in the second one we used a realistic case study based on applications in the field of digital signal processing.
1) Synthetic Workload: The technique we adopted to synthetically generate task sets composed of DAG tasks is based on the generator made available online by Melani et al. [34]: details on this generator and its configuration are reported in Appendix A. Note that this generator also produces a WCET $C_{i,j}$ for each node of the generated graphs and a period for each task. For each node the amount of local memory $LM_{i,j}$ is generated proportionally to the WCET, with uniform distribution in the interval $[K_{LM}^L C_{i,j}, K_{LM}^U C_{i,j}]$, where $K_{LM}^L, K_{LM}^U$ are two scale factors such that $0 < K_{LM}^L < K_{LM}^U < 1$.

The amount of memory exchanged with each edge $m_{i,j,s}$ is generated proportionally to the WCET of $v_{i,j}$ and its number of outgoing edges $n_{out}$, using uniform distribution in the interval $[K_{E}^L C_{i,j}, K_{E}^U C_{i,j}]$, where $K_{E}^L, K_{E}^U$ are two scale factors.

In the charts reported in this section, we set $K_{LM}^L = K_{E}^L = 0.2$ and $K_{LM}^U = K_{E}^U = 0.9$ (except in Figure 6(a) and (b), where $K_{E}^L$ varies) to have heterogeneity in local memory requirements and edge weights.

Tasks have been partitioned with the first-fit and worst-fit heuristics w.r.t. to the utilization (see Appendix A), and we assumed that each consecutive pair of nodes allocated to the same core communicates through the local scratch-
pad. Task sets for which a partitioning could not be found according to at least one partitioning heuristic have been discarded since in this paper we are interested about memory feasibility, not schedulability. In each chart, we analyzed four different configurations: (i) first-fit partitioning with closed-form bound (denoted as $\text{FF}_L$); (ii) first-fit partitioning with algorithmic characterization of the intra-memory interference $M_{j,k}^{\text{intra}}$ (denoted as $\text{FF}_L^{\text{Alg}}$); (iii) worst-fit partitioning with closed-form bound ($\text{WF}_L$); and (iv) worst-fit partitioning with algorithmic characterization of $M_{j,k}^{\text{intra}}$ ($\text{WF}_L^{\text{Alg}}$). For each value of the x-axis, the corresponding value on the y-axis is the average on 500 different task sets.

Figures 6(a) and (b) show the variation of MSR as a function of $K^*_L$, for 4 and 8 processors, and 5 and 10 DAG tasks, respectively. As expected, when the minimum memory requirement for each edge increases (and then also $K^*_L$), the overall memory requirement increases. When the first-fit heuristic is adopted, the MSR is much higher in comparison to worst-fit. The reason behind this result is that worst-fit tends to spread nodes of the same DAG among processors, whereas first-fit tends to pack entire DAGs on a single processor. Spreading nodes of a DAG among processors increases the size of set $Z$ (as defined in Lemma 3), thus also increasing the number of edges that can be neglected in the computation of the intra-memory inference, when both the closed-form bound and the algorithmic approaches are considered. For the same reason, the pessimism introduced by the closed-form bound decreases when worst-fit is adopted. The reduction in local MSR will however be at the cost of increased communication delays as nodes must more often access the global memory.

Figures 6(c) and (d) show the memory feasibility ratio when the size of the scratchpads is varied, considering 4 and 8 processors, and 5 and 10 DAG tasks, respectively. The memory feasibility ratio represents the ratio between the number of generated configurations that fit in each scratchpad when a specific node partitioning scheme is used, divided by the number of cores and the number of task sets. Scratchpads sizes are derived as $C_{\text{max}} \times N_{\text{max}} \times n \times K_S$, where $C_{\text{max}}$ and $N_{\text{max}}$ are the maximum WCET per node and the maximum number of nodes of the generated DAGs, respectively (see Appendix A). $n$ is the number of tasks used in each experiment, and $K_S$ a scale factor. Also in this case, a worst-fit partitioning scheme performs better than first-fit: for instance, Figure 6(d) shows that when $K_S = 0.2$, $\text{WF}_L^{\text{Alg}}$ is able to satisfy the memory requirement for 88% of the generated task sets, whereas $\text{FF}_L^{\text{Alg}}$ guarantees only 22%. Again, this is because first-fit favors local communication while worst-fit increases the number of communications through global memory. Therefore, memory feasibility and schedulability have contending objectives that will need to be balanced using appropriate partitioning algorithms.

2) The STR2RTS Case Study: The second experimental study is aimed at evaluating the MSR for realistic applications. To this purpose, we used the STR2RTS Benchmark Suite [35]. STR2RTS is derived from the StreamIT Benchmark [36] code, which consists of digital signal processing applications. In STR2RTS, each application is represented as a DAG. For each benchmark, it contains an XML description of the DAG representing the application, which includes: (i) a list of nodes, (ii) their precedence constraints, (iii) their WCETs, (iv) the amount of local memory needed by each node, and (v) the amount of memory assigned to each edge. Table II lists the seven representative benchmarks we selected for our experimental evaluation.

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>FFT4</td>
<td>Precise Fast Fourier Transform</td>
</tr>
<tr>
<td>B2</td>
<td>FilterBankNew</td>
<td>Multi-rate signal processing</td>
</tr>
<tr>
<td>B3</td>
<td>FMRadio</td>
<td>FM radio</td>
</tr>
<tr>
<td>B4</td>
<td>AudioBeam</td>
<td>Audio beam-forming</td>
</tr>
<tr>
<td>B5</td>
<td>Beamformer</td>
<td>Beam-forming</td>
</tr>
<tr>
<td>B6</td>
<td>CFAR</td>
<td>Constant False Alarm Detection</td>
</tr>
<tr>
<td>B7</td>
<td>FFT2</td>
<td>Fast Fourier Transform</td>
</tr>
</tbody>
</table>

We considered each benchmark separately (i.e., each one as a task set with a single DAG task). Since STR2RTS does not contains data concerning minimum inter-arrival times and relative deadlines, we tried two different partitioning heuristics. The first heuristic partitions nodes according to worst-fit with respect to the WCETs, thus tending to scatter nodes between cores and reducing data locality. In the second heuristic, we attempted to maximize parallelism while keeping decent local communications at the same time. A detailed description of this second heuristic is reported in Appendix B. Figures 7 and 8 (in Appendix) show the MSR of each processor when the two different partitioning schemes are adopted, considering a platform composed of $M = 4$ processors. To better evaluate the differences among the different benchmarks, we adopted a logarithmic scale on the y axis. Interestingly, both figures show that the closed form bounds are very close to the algorithmic bounds for such realistic workloads. Visible differences can only be appreciated for benchmark B2 on CPU4 in Figure 7; in Figure 8 appreciable differences can be seen for B2 on CPU2 and CPU4 and a small difference for B3 on CPU2 and CPU3. We conclude that the algorithmic approaches increase accuracy and hence should be used whenever possible. Nevertheless, whenever a very fast analysis is required, for instance when it is used as part of an iterative process for optimizing system parameters, the closed form bounds showed to be a viable alternative, providing very good estimations of the worst-case local memory consumption.

VII. CONCLUSION AND FUTURE WORK

This paper presented a memory feasibility analysis for parallel tasks running upon a scratchpad-based multicore platform. Both closed-form bounds and algorithmic solutions have been presented. A fine-grained characterization of the memory space requirement has been achieved by deriving an efficient technique based on max-plus algebra that applies to NFJ tasks, and reducing to max flow-cut problems for the case of DAG tasks, which are shown to be solvable in polynomial time by transformation to a maximum-weight independent set problem. The approaches have been evaluated with synthetic workload and a state-of-the-art benchmark. Closed-form bounds have been found tight on the benchmark. Future work will target the design of partitioning algorithms that integrate both memory feasibility and schedulability analysis to assign sub-tasks to processors.
APPENDIX A
SYNTHETIC WORKLOAD GENERATION

This work adopted the DAG task generator presented in [34]. Recently, the same generator has also been adopted for the experimental evaluation of other approaches concerning the schedulability analysis of DAG tasks [13, 37]. Each DAG is generated starting from a fork-join chunk composed of two nodes connected in series. Then, nodes are recursively expanded by replacing them with fork-join graphs. NFJ tasks are converted into DAGs by randomly adding edges with a probability $p_{\text{add}}$ among arbitrary selected nodes, without introducing cycles. During the recursive expansion of the fork-join chunk, each node has a probability $p_{\text{fork}}$ to fork and a probability $p_{\text{term}}$ to be a termination node. The number of nested forks is limited by a maximum depth. The number of branches generated by a fork node is randomly picked in the interval $[2, n_{\text{par}}]$ with uniform distribution.

To allow a partitioning strategy based on scheduling parameters, we also generated a worst-case execution time, a relative deadline computation time required by a DAG is

\[ C_i = \sum_{j \in V_i} C_{i,j}. \]

The utilization of a DAG is defined as $U = \frac{C_i}{T_i}$. Given a number of tasks and a target overall system utilization $U = \sum_{i \in C_i} C_{i,j}$, individual tasks utilizations are generated with the UNUfast algorithm [38], consequently deriving $T_i = C_i / U_i$. For each task, we considered relative deadlines equal to minimum inter-arrival times, i.e., $D_i = T_i$. Concerning the generation of each graph $G_i$, we set $p_{\text{fork}} = 0.8$, $p_{\text{term}} = 0.2$, $p_{\text{add}} = 0.2$, $n_{\text{par}} = 6$, and the maximum number of nesting equal to 2. With this configuration, each generated graph has at most $N_{\text{MAX}} = 50$ nodes.

APPENDIX B
RANK-BASED HEURISTIC

Definition 8: The rank $[39]$ of a node $v_{ij} \in V_i$ is defined as:

\[ \text{rank}(v_{ij}) = \begin{cases} 0 & \text{if } v_{ij} \text{ is source} \\ \max_{v_{ik} \in \text{pred}(v_{ij})} \text{rank}(v_{ik}) + 1 & \text{otherwise.} \end{cases} \]

Algorithm 3 Pseudo-code for a rank-based partitioning of nodes.

1: procedure RANKBASEDPARTITION($T_i$, $m$)
2: \hspace{1cm} $d$ \hspace{1cm} m is the number of processors
3: for all $\tau_i \in \Gamma$ do
4: for all $v_{ij} \in \tau_i$, in topological order do
5: \hspace{1cm} compute $\text{rank}(v_{ij})$ according to Definition 8
6: end for
7: end for
8: for all $v_{ij} \in \tau_i$, sorted w.r.t. to $\text{rank}(v_{ij})$ do
9: if $v_{ij}$ is the first node of $\tau_i$ to be partitioned then
10: $c \leftarrow 0$
11: else if two consecutive nodes with the same rank then
12: $c \leftarrow (c + 1)/m$
13: else
14: $c \leftarrow \text{cpu in which a } v_{ik} \in \text{pred}(v_{ij}) \text{ is allocated}
15: end if
16: allocate $v_{ij}$ to core $c$
17: end for
18: end procedure

Acknowledgment

This work has been partially supported by the RETINA Eurostars Project E10171 and by National Funds through FCT (Portuguese Foundation for Science and Technology) within the CISTER Research Unit (CEC/04234).

References


