Critical-Path-First Based Allocation of Real-Time Streaming Applications on 2D Mesh-Type Multi-Cores

Hazem Ismail Abdel Aziz Ali
CISTER Research Centre/INESC-TEC
Polytechnic Institute of Porto, Portugal
Email: haali@isep.ipp.pt

Luís Miguel Pinho
CISTER Research Centre/INESC-TEC
Polytechnic Institute of Porto, Portugal
Email: lmp@isep.ipp.pt

Benny Akesson
Eindhoven University of Technology
The Netherlands
Email: k.b.akesson@tue.nl

***Abstract*—Designing cost-efficient multi-core real-time systems requires efficient techniques to allocate applications to cores while satisfying their timing constraints. However, existing approaches typically allocate using a First-Fit algorithm, which does not consider the execution time and potential parallelism of paths in the applications, resulting in over-dimensioned systems.**

This work addresses this problem by proposing a new heuristic algorithm, Critical-Path-First, for the allocation of real-time streaming applications modeled as dataflow graphs on 2D mesh multi-core processors. The main criteria of the algorithm is to allocate paths that have the highest impact on the execution time of the application first. It is also able to exploit parallelism in the application by allocating parallel paths on different cores. Experimental evaluation shows that the proposed heuristic improves the resource utilization by allocating up to 7% more applications and it minimizes the average end-to-end worst-case response time of the allocated applications by up to 31%.

I. Introduction

Multi-core architectures integrating several low performance cores on a single chip became popular solutions for high-complexity applications, instead of using a single core with high performance. Many embedded systems incorporate a multi-core processor to satisfy the increasing demands of its applications, since the need for a high processing power at a low power budget is a great concern for such systems [1].

A main category of embedded system applications is streaming multimedia applications; which are becoming increasingly important and widespread. Many streaming applications have high processing requirements and timing constraints that must be satisfied, e.g., H.264 video decoders [1]. This raises the need for a parallelization model for applications to use massive computational power [2], which the dataflow model of computation is able to achieve for streaming applications [3]. Furthermore, since these applications are basically a series of transformations that are applied to a data stream, the dataflow model is a natural paradigm for representing them for concurrent implementation on multi-core processors [3]. A dataflow model is specified by a directed graph, where the nodes are considered as functions and the connections between the nodes, i.e. edges, as channels of data. There are many examples of applications that adopt this model of computation, such as MPEG reconfigurable video coding (RVC) [4] or Adaptive Multi-Rate Wideband Speech Codec (AMR-WB) [5].

As dataflow models are represented by graphs, the problem becomes how to allocate the graphs' nodes on the cores to obtain high performance, and, as tackled in this work, also to address timing constraints. This allocation problem has previously been tackled in several works from a high-performance point-of-view [6]–[10]. However, these approaches do not consider timing constraints and thus cannot be used for allocation of real-time dataflow applications. Another allocation approach uses First Fit (FF), which has been shown to behave as well, and outperform others in some cases, in terms of achieved throughput [11]. However, applying approaches based on FF and that satisfy timing constraints [12] results in over-dimensioned systems, as our experimental evaluation will show.

Dataflow applications are mostly statically scheduled. Static scheduling proved its success in systems that only run dataflow applications. However, for systems that run mixed real-time applications (dataflow and non-dataflow), an optimal dynamic real-time scheduling algorithm, e.g. Earliest Deadline First (EDF), will have a higher schedulability success rate than static scheduling, and enables efficient analysis techniques for such kind of systems.

Therefore, in this paper, we propose a new approach called Critical-Path-First (CPF) for allocation of real-time applications modeled as dataflow graphs on 2D mesh multi-core processors. The new approach uses Partitioned EDF (PEDF) for scheduling applications on the multi-core platform. CPF is based on allocating first, for each application, the paths with highest end-to-end execution time, maximizing path parallelism when possible. This allows maximizing the usage of the available resources and potentiates parallelism, which yields average lower end-to-end response time of the applications. CPF also has a tendency to order the allocation of tasks from heaviest to lightest, which has been shown to provide a better solution than FF [13]. We experimentally show that CPF outperforms the FF allocation algorithm in terms of number of allocated applications and average worst-case response time of the allocated applications without negatively impacting the run-time of the allocation algorithm.

The paper is organized as follows. Section II provides an overview of related work. Afterwards, Section III explains the main concepts necessary to understand the system model and evaluation parameters. The CPF algorithm is detailed in Section IV, while Section V provides the evaluation and discussion of experimental results. Finally, we provide some conclusions in Section VI.

II. Related Work

The problem of task allocation for dataflow applications has been the subject of quite some previous research. The approaches in [6]–[10] address the problem of task allocation on multi-core platforms, taking into account the sizes of the tasks, the communication between them and load balancing. However, these approaches do not take timing constraints into account, which is the main focus of this work.

Other approaches in [14]–[16] address the problem of task allocation and scheduling of real-time streaming applications on multi-core platforms, taking into account the throughput and timing constraints with a main objective of minimizing energy consumption. On the other hand, our main objective is the efficient usage of system resources, maximizing the number of allocated applications, which is a completely different problem.

In [17], the author discusses a static algorithm for allocating and scheduling components of periodic tasks that consist of subtasks (nodes, actors) with precedence constraints across sites in distributed systems (equivalent to cores of a multiprocessor). The algorithm consists of two parts; the first part decides whether a group of communicating subtasks of a task should be assigned to the same site as a cluster, while the second part allocates the clusters of subtasks to the sites in a system based on the ability to find a feasible static non-preemptive schedule for the subtasks, as well as the communication between them. Compared to our work, the approach in [17] first clusters the tasks and then tries to find a feasible schedule. In contrast, we use PEDF [18], a dynamic preemptive scheduling algorithm. This enables us to verify schedulability by simply checking the processor utilization bound $u \leq 1$ while performing allocation, which reduces the time to find a feasible schedule. Furthermore, although both approaches have the goal of meeting the timing constraints of the applications, we also aim for efficient usage of system resources, maximizing the number of allocated applications, and for improving responsiveness, minimizing the average end-to-end worst-case response time of allocated applications.

In [19], the authors propose a similar approach as [17]. However, they do not allow subtasks (nodes) of a task (graph) to execute on different sites (cores) and they use a branch and bound (BB) search to find a feasible schedule, while in [17] it is a heuristic search. However, our proposed algorithm detects nodes that can run in parallel and allow them to run on other cores to reduce application end-to-end worst-case response time. Also, the simple processor utilization test, $u \leq 1$, for schedulability is much faster and easier than the exhaustive search of BB.

Stuijk et. al [20] presented a resource allocation strategy that can allocate multiple SDF graphs onto a heterogeneous multi-core platform with throughput guarantees to each individual application. The proposed method can deal with cyclic dependencies and multi-rate SDF graphs without the need to convert them to Homogeneous SDF (HSDF) graphs. The allocation strategy starts by binding actors of the SDF graph to a core on the multi-core platform to satisfy the application throughput constraint. This is done by first considering the actors whose execution time have a large impact on the application throughput. Then, a static order schedule for each core containing actors of the SDF graph is computed. Finally, time slices are allocated to cores based on a binary search algorithm that satisfies the throughput constraint. In contrast, we first consider allocation of actors in the critical path of the application that have a large impact on the end-to-end worst-case response time of the application, instead of individual actors. Also, we use PEDF instead of static scheduling, which has a higher schedulability and enables efficient timing analysis techniques for mixed real-time systems.

Another work in [21] proves the existence of a feasible dynamic non-preemptive Rate Monotonic (RM) schedule for dataflow applications modeled as SDF graphs and then proposes an execution model for it. In [22], the authors provide a throughput analysis for an embedded multi-core system that executes a set of soft/hard real-time data stream applications, modeled as SDF graphs, using TDMA as the scheduling algorithm. Contrarily, our proposed work uses a dynamic preemptive scheduling PEDF, which always allows full processor utilization over RM or static scheduling, which implies a more efficient exploitation of computational resources.

In [12], the authors provide an approach where actors (nodes) of streaming applications are considered as implicit-deadline periodic tasks. They provide results from real streaming applications from the SDF^3 Benchmark [23], and also use PEDF as the scheduling algorithm for periodic tasks (nodes, actors). They use the FF algorithm for the allocation of nodes on the cores, and show that in more than 80% of the cases the throughput resulting from the approach is equal to the maximum achievable throughput. This result is obtained under the condition of deriving the number of processors to guarantee the maximum (ideal) throughput. The authors show that this throughput condition increases the number of processors required for allocation, thus decreasing their individual utilization. Contrarily, we propose the *Critical-Path-First* (CPF) allocation heuristic, which maximizes the usage of system resources and minimizes the average end-to-end worst-case response time of applications.

III. Background / Preliminaries

In this section, we present an overview of the dataflow model of computation considered in this work. This background is essential for the understanding of the system model and proposed allocation algorithm. Furthermore, we also describe the metrics used in the evaluation of our approach.

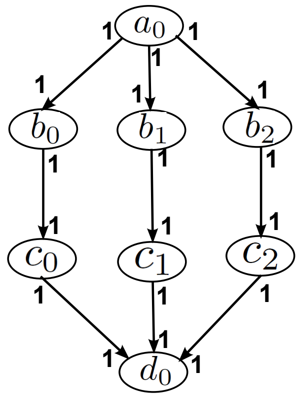
A. Homogeneous SDF (HSDF)

The dataflow model of computation is widely used in modeling and analyzing streaming, Digital Signal Processing (DSP) and concurrent multimedia applications [24], [25]. Its use has been increasingly considered for designing applications for multi- and many-core processors [26]. Homogeneous Synchronous Dataflow (HSDF) [3] is a special case of dataflow graphs in which all rates (production and consumption) associated with actor (node) ports are equal to one. Therefore, when each actor is fired once, the distribution of tokens on all channels return to their initial state. This is referred to as a *complete cycle* or *graph iteration*. Figure 1a shows an example of an HSDF graph.

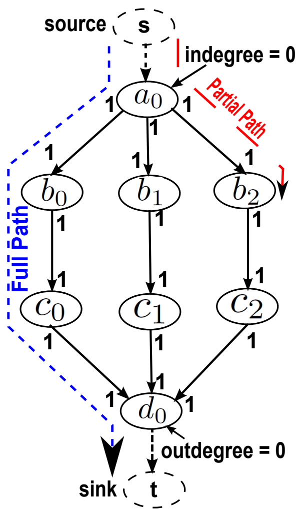
Dataflow graphs, e.g. SDF and Cyclo-Static Dataflow (CSDF) are examples of expressive models that can be converted to an equivalent HSDF graph by using a conversion algorithm, such as the one presented in [25]. This also enables these models to be used with our approach.

B. System Model

This work considers the allocation of applications onto a multi-core processor with identical cores (tiles), interconnected by a 2D mesh Network on Chip (NoC), such as TILE64™



(a) An example HSDF graph.



(b) Adding source s and sink t.

Figure 1: HSDF graph example.

[27]. Each tile is a full-featured processor that includes a nonblocking switch that connects the tile to the 2D mesh network. We consider the cost of communication as a single constant cost per hop, which is based on the assumption of having an extremely low-latency, high-bandwidth channelized NoC for streaming data. All cores share a single OS, but with independent scheduling queues per core, since are tasks pre-allocated to cores.

Formally, we consider a system S based on a two dimensional mesh (2D Mesh) homogeneous symmetrical multi-core platform, represented by the set $M = \{m_1, m_2, \ldots, m_n\}$, where n is the number of cores. The 2D Mesh processor runs a set of m periodic applications $A = \{A_1, A_2, \ldots, A_m\}$. Each application A_i is represented by a graph $G_i = \langle V_i, E_i\rangle$, where V is the set of nodes and E the edges connecting them. Each node in this graph is an actor and each edge is a communication channel. Each actor is considered as a periodic task that may be preempted at any time. All actors are scheduled on M using the PEDF scheduling algorithm. Although it is non-optimal [28], it is still suitable as migration of tasks (nodes) is not allowed. All graphs G are Directed Acyclic Graphs (DAG), modelled using the HSDF model of computation. A periodic task $\tau \in V$ is represented by the 3-tuple $\tau = (C_i, T_i, D_i)$, where C_i is the worst-case execution time, T_i is the period, D_i is the deadline of the task. The utilization of task τ_i is denoted by U_i and is defined as $U_i = C_i/T_i$, where $U_i \in (0, 1]$. The set of applications A running on the system S have throughput requirements $\zeta = \{\zeta_1, \zeta, \ldots, \zeta_m\}$ that each application must fulfill. Therefore, in our case the period T_{A_i} of each application A_i is set to the inverse of its throughput requirement ζ_i, and the period T_i of each task in an application A_i is set to its period, $T_i = T_{A_i}$. This means that each actor in the HSDF graph only fire once per iteration of the graph. We do not consider a more complete model where τ_i is augmented with a start instant s_i (an offset).

Note that approaches to enforce precedence constraints between tasks in the DAG (e.g. offline specifying offsets, adapting deadlines [29] or use of semaphores [30]) imply that tasks have either a static or dynamic start instant (an offset) after the start instant of the DAG. However, the exact timing analysis and the associated feasibility tests for such asynchronous task models are complex. We can simplify the analysis by ignoring these offsets and analyze the system as if all tasks are released at the same instant ($s_i = 0$), which is a worst-case assumption that gives a sufficient, although pessimistic, analysis [31]. In our approach, a simple utilization-based feasibility test is used for the decision to allocate DAG paths to cores, thus low complexity is fundamental. Baruah et al. [32] showed that, given an asynchronous periodic task set (s_i, C_i, T_i, D_i), if the corresponding synchronous task set (C_i, T_i, D_i) (obtained by considering all offsets $s_i = 0$) is feasible, then the asynchronous task set is feasible too. Therefore, in this system S, a task set (nodes set) V refers to a synchronous set of implicit-deadline periodic tasks. As a result, we refer to a task τ_i with a tuple $\tau_i = (C_i, T_i)$ by omitting the implicit deadline D_i and the start time s_i.

In this paper, we propose an allocation algorithm for a set of periodic applications A on a 2D Mesh homogeneous symmetrical multiprocessor M with the goal of maximizing the usage of system resources (increasing the number of allocated applications) using the highest critical-path-first allocation criteria. Moreover, we minimize the average end-to-end worst-case response time of the allocated applications R^{av}_A on the system S by enabling application-level parallelism.

C. Evaluation Metrics

Two metrics are used to evaluate our approach: 1) the number of allocated applications N, and 2) the average end-to-end worst-case response time gain of the applications $R^{av}_{A_{gain}}$. We also measured the total utilization of the multi-core processor U_M (the average of all core utilizations, $U_M = \sum_{i=1}^{n} U_{m_i}/n$, where U_{m_i} is the utilization of core i), and the run-time t_r of the algorithm.

The $R^{av}_{A_{gain}}$ represents the gain achieved by the average end-to-end worst-case response time of the allocated applications compared to FF. The average end-to-end worst-case response time of allocated applications R^{av}_A is calculated by computing the average of all applications end-to-end worst-case response times R_{A_i}. This means, $R^{av}_A = \sum_{i=1}^{m_a} R_{A_i}/m_a$, where m_a is the total number of allocated applications on the multi-core processor. After the calculation of R^{av}_A for each allocation algorithm, the average end-to-end worst-case response time gain of the applications is $R^{av}_{A_{gain}} = \frac{R^{av}_{A_{FF}} - R^{av}_{A_{CPF}}}{R^{av}_{A_{FF}}}$, where $R^{av}_{A_{FF}}$ is the average end-to-end worst-case response time of applications allocated using FF and $R^{av}_{A_{CPF}}$ is the average end-to-end worst-case response time of applications allocated using CPF.

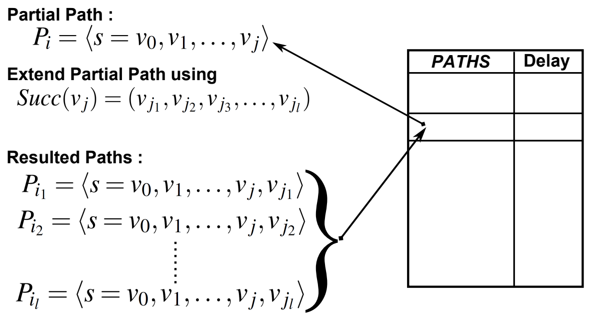
The application end-to-end worst-case response time R_{A_i} represents the end-to-end worst-case response time of a graph iteration. In our case, R_A is the summation of worst-case response times of all actors that affect the execution of the most critical path in the application. For example, assume a multi-core platform that have two cores, and we would like to allocate the application graph shown in Figure 1a on it. Suppose the application actors $(a_0, b_0, b_1, b_2, c_0, c_1, c_2, d_0)$ have a period of 10 and utilizations $(0.3, 0.2, 0.1, 0.1, 0.2, 0.1, 0.1, 0.3)$, respectively. The critical path of the application is (a_0, b_0, c_0, d_0). The allocation of these actors using FF is $\{core_1(a_0, b_0, b_1, b_2, c_0), core_2(c_1, c_2, d_0)\}$, which results in $R_{A_{FF}} = R_{A_{a_0}} + R_{A_{b_0}} + R_{A_{c_0}} + R_{A_{d_0}} + R_{A_{b_1}} + R_{A_{c_1}} + R_{A_{b_2}} + R_{A_{c_2}}$. This is due to the critical path of the application (a_0, b_0, c_0, d_0) spans the two cores (actors (a_0, b_0, c_0) are on $core_1$ and actor (d_0) is on $core_2$). However, a different allocation approach that exploits parallelism inside the application allocates the graph as $\{core_1(a_0, b_0, c_0, d_0), core_2(b_1, b_2, c_1, c_2)\}$, results in $R_{A_{other}} = R_{A_{a_0}} + R_{A_{b_0}} + R_{A_{c_0}} + R_{A_{d_0}}$, which is less than $R_{A_{FF}}$. This is due to the critical path of the application is allocated on a single core ($core_1$). Therefore, an allocation algorithm that uses parallelism inside application can minimize the end-to-end worst-case response time of the application and this reflects in the average end-to-end worst-case response time gain of the applications $R^{av}_{A_{gain}}$.

IV. Allocation Algorithm

The algorithm presented in this section is intended for the allocation of applications modeled as HSDF graphs onto 2D mesh multi-cores at design time. It consists of two main phases. The first phase is finding all possible paths between the actors of the applications on the system, inspired by the algorithm in [33] for finding the K most critical paths (CP). Contrarily to [34]–[36], which intend to develop faster methods to find a single CP, we are concerned with finding all possible paths in the application graphs, and ordering them in a non-increasing order of delays (most CP first) for each application alone, since the CP affect the worst-case response time of the application. The second phase is to allocate the actors of the graph on the cores of the mesh processor using the output information of the previous phase to minimize the average end-to-end worst-case response time of the applications R^{av}_A. We proceed by discussing these two phases in more detail.

A. First phase: Finding all possible paths

In this phase, we calculate all paths for a given DAG in non-increasing order of delays . This enhances the selectivity of CPF over FF, since it gives higher priority to tasks in a high CP than tasks in a less CP. So, there is an indirect ordering of tasks in form of ordered paths (ordered sets of tasks), from which CPF selects these paths and allocate in order on a task-by-task basis in the second phase of the algorithm. This means that CPF first allocates paths (task sets) that have highest impact on the execution time of the application. This gives an advantage to CPF over FF in the sense that it helps in allocating


Partial Path :
$P_i = \langle s = v_0, v_1, \ldots, v_j \rangle$
Extend Partial Path using
$Succ(v_j) = (v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l})$
Resulted Paths :
$P_{i_1} = \langle s = v_0, v_1, \ldots, v_j, v_{j_1} \rangle$
$P_{i_2} = \langle s = v_0, v_1, \ldots, v_j, v_{j_2} \rangle$
$P_{i_l} = \langle s = v_0, v_1, \ldots, v_j, v_{j_l} \rangle$
PATHS
Delay
Figure 2: Path enumeration.

more applications. As noted in [13], a packing algorithm performs worst when tasks with larger utilizations happen to be packed last, because new cores will be needed before earlier cores can be filled up. A better strategy is to order the tasks from the heaviest to the lightest before allocation.

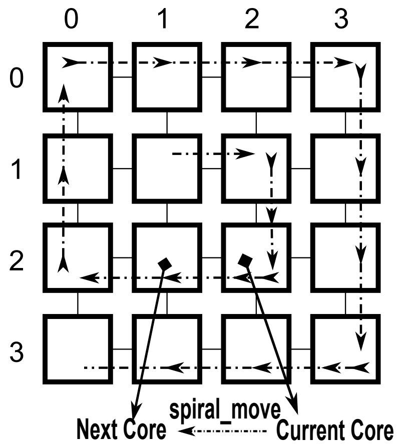
The first phase of the algorithm is divided into two stages:

1) Creation of source and sink actors : First we search the graph G to find all actors with in-degree (out-degree) equal to zero. Actors with zero in-degree (out-degree) are specified as the starting-actors (ending-actors). A dummy source s (sink t) actor that has a zero execution time is inserted at the beginning (end) of the graph G, as shown in Figure 1b. These two actors (s,t) are connected with dummy links to starting and ending actors, respectively. By doing this conversion, all paths in G have a uniform form as $\langle s = v_0, v_1, v_2, \ldots, v_n, v_{(n+1)} = t\rangle$, where $\langle v_i, v_{(i+1)}\rangle$, $v_i \in G$, for $1 \le i \le n$. A path $P = \{v_0, v_1, \ldots, v_{(j-1)}, v_j\}$ is considered a *full path* iff $v_0 = s$ and $v_j = t$. Otherwise, P is considered a *partial path*. For example, path $\{s, a_0, b_0, c_0, d_0, t\}$ in the HSDF application shown in Figure 1b is a *full path*, while path $\{s, a_0, b_2\}$ is a *partial path*. This step is very important to get a uniform path in case of a DAG with multiple starting and/or ending actors.

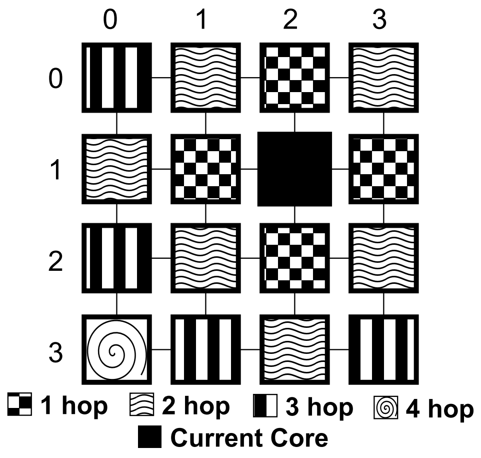
2) Path enumeration : This is an iterative process where all possible paths in the application DAG are generated. A lookup table *PATHS* is generated to save all possible paths and their delay values in a non-increasing order of delays. The process starts by initializing the table with a few partial paths. In this case, these initial partial paths are all single hop paths generated by combining the start actor s and its list of successor actors $Succ(s)$. Then, the process picks up a partial path $P_i = \langle s = v_0, v_1, \ldots, v_j\rangle$ from *PATHS[i]*, as shown in Figure 2, and extends it to a full path. The extension process starts by getting the $Succ(v_j) = (v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_l})$, where l is the number of actors in $Succ(v_j)$. Then, it extends the partial path P_i to its l possible paths, $P_{i_1} = \langle s = v_0, v_1, \ldots, v_j, v_{j_1}\rangle, P_{i_2} = \langle s = v_0, v_1, \ldots, v_j, v_{j_2}\rangle, \ldots, P_{i_l} = \langle s = v_0, v_1, \ldots, v_j, v_{j_l}\rangle$. It removes P_i and inserts its l possible paths in the lookup table in a non-increasing order of delays. The path enumeration process loops until all partial paths in *PATHS* extended to full paths.

B. Second phase: Allocation of actors

The second phase of the approach is an algorithm that allocates the graphs' actors on the multi-core processor with a main criteria of minimizing the average end-to-end worst-case response time of the applications R^{av}_A by enabling application-level parallelism. The first phase creates *PATHS* for each application A_i, called $PATHS_{A_i}$. All $PATHS_{A_i}$ are gathered



(a) *spiral_move*



(b) *find_nearest_core*

Figure 3: Core Selection methodology

Algorithm 1: The Critical-Path-First (CPF) algorithm

```
$PATHS_{A_i}$: Lookup table for all possible paths in application $A_i$ ordered according to criticality.
$PATHS_G$: Global lookup table for all $PATHS_{A_i}$ of all applications on the system $S$.
$P_{A_i}$: A path of application $A_i$ in $PATHS_G$ lookup table, $P_{A_i} = \langle v_0, v_1, v_2, \dots, v_j \rangle$.
$P^p_{A_i}$: Partial path of full path $P_{A_i}$
$LP^p_{A_i}$: List of partial paths.

begin
    $n$ = spiral_move();
    foreach $P_{A_i}$ in $PATHS_G$ do
        if $P_{A_i}$ is Independent then
            foreach $v_j$ in $P_{A_i}$ do
                while (all cores are not tested) and ($v_j$ not allocated) do
                    if $U_{m_n} + u_{v_j} \leq 1$ then
                        allocate $v_j$ on core $m_n$.
                    else
                        $n$ = spiral_move();
                if $v_j$ not allocated then
                    unallocate $\forall v_j \in A_i$ from $M$.
        else // Dependent Path Case
            search for possible $P^p_{A_i}$ in $P_{A_i}$.
            classify found $P^p_{A_i}$ & add them to $LP^p_{A_i}$.
            foreach $P^p_{A_i}$ in $LP^p_{A_i}$ do
                if Head or Tail then
                    find the reference actor (Parent).
                    allocate using find_nearest_core.
                else if Middle then
                    calculate mid-point (core).
                    allocate using find_nearest_core.
                if ($v_j$ in $P^p_{A_i}$) not allocated then
                    unallocate $\forall v_j \in A_i$ from $M$.
```

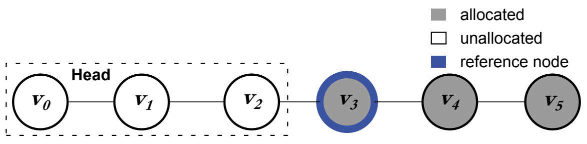
to create the global lookup table of all the paths $PATHS_G$, which will be used as input by the second phase to allocate the graph actors.

1) Definitions:

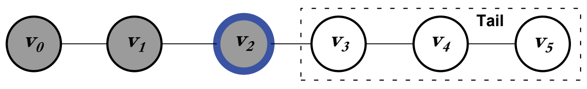
Independent / Dependent Path: A path $P_{A_i} = \langle v_0, v_1, v_2, \dots, v_j \rangle$ of a certain application A_i is said to be *independent* iff all its actors are unallocated. If at least one of P_{A_i} actors is already allocated, the path is considered *dependent*.

Allocation Condition: This is the condition used for checking the ability of the core to take a new actor. In this algorithm, we use the utilization of the core U_{m_i} to formulate the allocation condition, based on the selection of PEDF as a scheduling algorithm. The algorithm cannot add an extra actor unless the summation of total utilization of the core U_{m_i} and the utilization of the actor u_j to be allocated $U_{m_i} + u_j \leq 1$, which is a very simple test that can even be done at run-time.

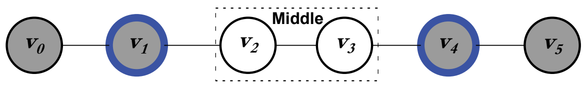
Core Selection: The selection process of a new core for assigning actors is done by two different methods, as shown in Figure 3, depending on the type of path to be allocated (independent/dependent). For independent paths, the selection is performed by *spiral_move*. As shown in Figure 3a, every time the *spiral_move* function is called it returns the next core in the spiral path. The *spiral_move* function is called when the current core fails the allocation condition. The spiral path for core selection is initialized only once at the beginning of the allocation process and advances by one core each time the allocation condition fails. For dependent paths, as they are partially allocated, allocation of child (unallocated) actors is done as near as possible to their parent (allocated) actors to reduce communication cost. The function *find_nearest_core* starts searching for a suitable core (a core that passes the allocation condition) one hop away from the reference core (defined in Section IV-B2), where the first core that passes the allocation condition test is selected. If not possible, it searches for a suitable core two hops away, and so on, until finding a possible core. The search criteria starts by finding the nearest core in this order: North, South, East and West. The first core that the *find_nearest_core* function finds is returned for allocation. Figure 3b shows the searching regions, classified according to the distance from the reference core.



(a) Class Head partial path



(b) Class Tail partial path



(c) Class Middle partial path

Figure 4: Partial path classes

2) Critical-Path-First (CPF) Algorithm:

The proposed approach, shown in Algorithm 1, picks a path P_{A_i} in order of criticality from $PATHS_G$. The selected path P_{A_i} is checked whether it is independent or dependent. P_{A_i} is always independent by definition if it is the most CP in the application A_i.

If the path P_{A_i} is independent, the algorithm allocates its actors $\langle v_0, v_1, v_2, \ldots, v_j\rangle$ onto the multi-core processor $M = \{m_1, m_2, \ldots, m_n\}$. For each actor v_j, the allocation process checks the allocation condition for the current core. If the condition is true, it assigns the actor v_j to the current core. Otherwise, the next core is selected using *spiral_move* and the process is repeated again.

On the other hand, if path P_{A_i} is dependent, the algorithm searches its partial paths $P^p_{A_i}$ (unallocated path sections) and classifies them into three classes: Head, Middle and Tail. Figure 4 shows the three classes of partial paths. For each partial path $P^p_{A_i}$, the algorithm determines a reference allocated actor (parent) and uses its core as a reference core in the process of selecting the nearest possible core. This reference actor (parent) is determined according to the $P^p_{A_i}$ class. In case of $P^p_{A_i}$ being a Head, the reference actor is the successor of the last actor in the partial path, as shown in Figure 4a. In case of a Tail, the reference actor is the last allocated actor before the partial path, as shown in Figure 4b. In the case of a Middle, the reference core is selected differently. The class middle partial path is surrounded by two allocated actors (parents), as shown in Figure 4c. The reference core is thus selected by computing the middle core between the parents. The location of the computed reference core is given to *find_nearest_core* as an input to find the possible nearest core to allocate the child actors.

The CPF approach uses two different techniques for allocating independent and dependent paths. This is because independent paths can be allocated on any set of cores that have enough capacity to accommodate the path. However, unallocated parts (children) of a dependent path need to be allocated near to their parents to decrease communication between child and parent actors. The partial path classification discovers potential parallelism in the application, since, from the definition, the full path (containing the partial path) shares some of its actors with another allocated path. This feature is an advantage and this knowledge allows to allocate these parallel sections on different cores (if possible), thus, enhancing the performance and reduce the end-to-end worst-case response time of the application R_{A_i}. If the heuristic fails in the allocation of any path P_{A_i}, the heuristic unallocates all previously allocated actors of the application A_i.

Table I: Benchmarks used in evaluation

No.	Application Name	Ref
1	H.263 decoder	[23]
2	H.263 encoder	
3	MP3 decoder (granule level)	
4	MP3 decoder (block level)	

V. Evaluation and Results

The proposed approach has been evaluated by implementing an allocation tool and experimenting on a set of streaming applications. These streaming applications are taken from the SDF^3 Benchmark [23]. The objective of the implemented heuristic is to allocate HSDF graphs onto a multi-core processor with the aim of maximizing usage of system resources by increasing the number of successfully allocated applications N, and minimizing the average end-to-end worst-case response time of allocated applications R^{av}_A by exploiting application-level parallelism.

A. Experimental Setup

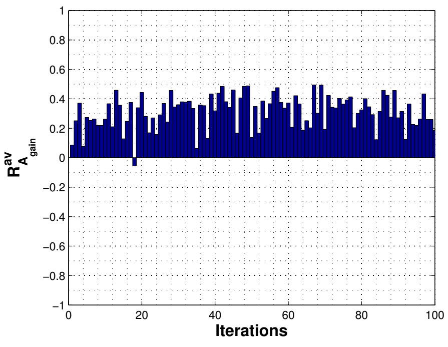
The selected set of streaming applications, presented in Table I, comprises HSDF DAG graphs. The allocation tool instantiates randomized combinations of these applications to create sets of 500 applications.

In this evaluation, five experiments have been carried out in order to assess the suitability of the proposed approach under different types of applications, since the applications in Table I mainly consist of two types: high ($u_i \geq 0.7$) and low ($u_i \leq 0.3$) total utilization. High utilization applications are the *MP3 decoder (granule level)* and the *MP3 decoder (block level)*. Low utilization applications are the *H.263 encoder* and the *H.263 decoder*. The five experiments use different weights for the random generator to achieve a range of High/Low applications: 100% High - 0% Low, 80% High - 20% Low, 60% High - 40% Low, 40% High - 60% Low and 20% High - 80% Low. Each experiment is performed 100 times (iterations) with different random sets and in each iteration the allocation tool generates a new set of 500 applications. For each iteration, the tool tries to allocate as many applications as possible on the multi-core platform using this approach. The same set of randomly generated applications are also applied, in the same order, as input to a FF allocation algorithm (implemented in the same tool) for comparison of the two evaluation metrics, previously described in Section III-C. FF is used since it has been shown to outperform other bin-packing algorithms in terms of achieved throughput, although with higher jitter [11].

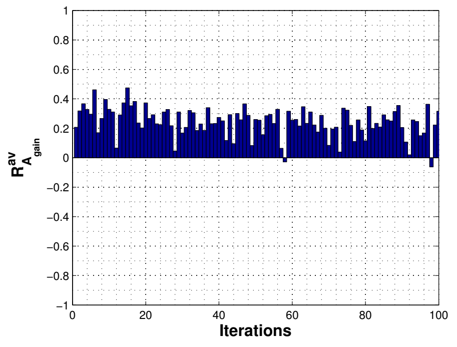
The multi-core platform follows the model explained in Section III-B. The size of the platform is an 8x8, 64 core 2D mesh. It has been modeled as a two dimensional array, where each element represents a core.

B. Experimental Results

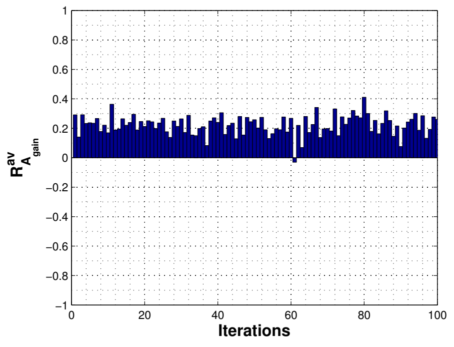
The evaluation of this approach is based on comparing to the results of the FF allocation algorithm with two metrics



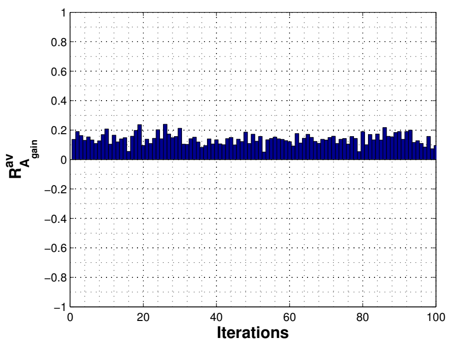
(a) Results of data set (100% High – 0% Low)



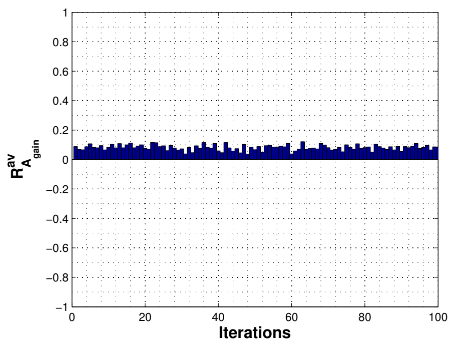
(b) Results of data set (80% High – 20% Low)



(c) Results of data set (60% High – 40% Low)



(d) Results of data set (40% High – 60% Low)



(e) Results of data set (20% High – 80% Low)

Figure 5: Average worst-case response time gain of the applications $R^{av}_{A_{gain}}$

Table II: Summary of results

High%*/*Low%	**100%/0%**		**80%/20%**		**60%/40%**		**40%/60%**		**20%/80%**	
Mean of	**CPF**	**FF**	**CPF**	**FF**	**CPF**	**FF**	**CPF**	**FF**	**CPF**	**FF**
$\boldsymbol{N}$	64.1	64.3	98.1	92.1	124.3	117.9	173.6	168.8	300.1	294.9
$\boldsymbol{t_r}$ (sec)	2.9	3.1	2.3	2.9	1.8	2.1	1.3	1.3	0.7	0.4
$\boldsymbol{R^{av}_{A_{gain}}}$	31.2%		24.4%		22.3%		14.1%		8.2%	

(N, $R^{av}_{A_{gain}}$), previously discussed in Section III-C.

The results of the five experiments are summarized in Table II and shown in Figure 5. Table II shows a comparison between CPF and FF with respect to the mean of the two metrics and the algorithm run-time t_r after one hundred iterations. These two metrics are: 1) the number of allocated applications N and 2) the average end-to-end worst-case response time gain of the applications $R^{av}_{A_{gain}}$. Figure 5 shows the average end-to-end worst-case response time gain of the applications $R^{av}_{A_{gain}}$ over one hundred iterations for five input data sets 100% High - 0% Low, 80% High - 20% Low, 60% High - 40% Low, 40% High - 60% Low and 20% High - 80% Low. The results in Table II show that CPF succeeds in using the resources efficiently by allocating up to 7% more applications in the (80% High - 20% Low) case. This is due to the selectivity nature of CPF that enables the allocation of actors in paths that have higher impact on end-to-end application execution time first, previously discussed in Section IV-A. Also, Figure 5 shows that *CPF* outperforms FF in $R^{av}_{A_{gain}}$, which means that the average end-to-end worst-case response time of the applications that uses *CPF* is lower than the others that use FF. This can be confirmed from Table II, since it shows that the $R^{av}_{A_{gain}}$ reached a value up to 31.2% in the (100% High - 0% Low) case. This is due to the main two features of CPF : 1) the allocation of independent paths on different cores, and 2) the method of allocating partial paths (Head, Tail, Middle) on different cores than their parents. These two features enable parallelism in each application and prevents (as much as possible) less CPs from interfering with the highest CP in the same application, which reflects in the end-to-end worst-case response time of each application. In terms of run-time t_r, both algorithms executes in a few seconds, showing that the added complexity is negligible. Also, the utilization of the multi-core platform for both algorithms is approximately 99%.

VI. Conclusions

In this paper, we propose to use a Critical-Path-First (CPF) approach for the allocation of real-time streaming applications, modeled as dataflow graphs, on 2D mesh multi-core processors. The main goal is to address timing constraints and maximize the overall usage of the system resources by allocating paths that have the highest impact on the end-to-end execution time of the application first. Also, CPF is able to minimize the average end-to-end worst-case response time of the applications allocated on the system by enabling application-level parallelism. The evaluation results demonstrate that *CPF* allocates up to 7% more applications, and minimizes the average end-to-end worst-case response time of the allocated applications up to 31% over FF.

Acknowledgment

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within VIPCORE (FCOMP-01-0124-FEDER-015006) and SENODs (FCOMP-01-0124-FEDER-012988) projects; also by FCT and ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/79872/2011 and projects EU FP7 288008 T-CREST and 288248 Flextiles, Catrene CA104 Cobra and CA505 BENEFIC, CA703 OpenES, and NL STW 10346 NEST.

References

[1] M. Kim et. al, "H.264 decoder on embedded dual core with dynamically load-balanced functional paritioning," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*, sept. 2010.

[2] V. Pankratius et. al, "Parallelizing bzip2: A case study in multicore software engineering," *Software, IEEE*, pp. 70–77, nov.-dec. 2009.

[3] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, sept. 1987.

[4] E. Bezati et. al, "RVC-CAL dataflow implementations of MPEG AVC/H.264 CABAC decoding," in *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, oct. 2010, pp. 207–213.

[5] H. Ali and M. N. I. Patoary, "Design and implementation of an audio codec (AMR-WB) using dataflow programming language CAL in the opendf environment," Master's thesis, Halmstad University, IDE, 2010.

[6] Y. Liu et. al, "Allocating tasks in multi-core processor based parallel system," in *Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, 2007, pp. 748–753.

[7] P.-Y. Richard Ma et. al, "A task allocation model for distributed computing systems," *Computers, IEEE Transactions on*, vol. C-31, no. 1, pp. 41–47, jan. 1982.

[8] R. Ennals et. al, "Task partitioning for multi-core network processors," in *Proceedings of the 14th international conference on Compiler Construction*, ser. CC'05, vol. 3443. Springer-Verlag, 2005, pp. 76–90.

[9] V. Lo, "Heuristic algorithms for task assignment in distributed systems," *Computers, IEEE Transactions on*, vol. 37, pp. 1384–1397, nov 1988.

[10] J. D. Evans and R. R. Kessler, "A communication-ordered task graph allocation algorithm," IEEE Transactions on Parallel and Distributed Systems, Tech. Rep., 1992.

[11] J. Guo and L. Bhuyan, "Load balancing in a cluster-based web server for multimedia applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 11, pp. 1321–1334, nov. 2006.

[12] M. Bamakhrama and T. Stefanov, "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications," in *Proceedings of the ninth ACM international conference on Embedded software*, ser. EMSOFT '11, 2011, pp. 195–204.

[13] P. Hoffman, *The Man Who Loved Only Numbers: The Story of Paul Erdos and the Search for Mathematical Truth*, ser. Biography-science. Hyperion Books, 1999.

[14] J. Hu and R. Marculescu, "Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, 2004, pp. 234–239.

[15] R. Xu et. al, "Energy-aware scheduling for streaming applications on chip multiprocessors," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, 2007, pp. 25–38.

[16] Y. Wang et. al, "Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 2, pp. 14:1–14:32, Apr. 2011.

[17] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 6, no. 4, pp. 412–420, apr 1995.

[18] J. M. López et. al, "Utilization bounds for edf scheduling on real-time multiprocessor systems," *Real-Time Syst.*, vol. 28, no. 1, pp. 39–68, Oct. 2004.

[19] D.-T. Peng and K. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," in *Distributed Computing Systems, 1989., 9th International Conference on*, jun 1989, pp. 190–198.

[20] S. Stuijk et. al, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, june 2007, pp. 777 –782.

[21] T. Parks and E. Lee, "Non-preemptive real-time scheduling of dataflow systems," in *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, vol. 5, may 1995, pp. 3235–3238 vol.5.

[22] M. Bekooij et. al, "Dataflow analysis for real-time embedded multiprocessor system design," in *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, ser. Philips Research Book Series, P. Stok, Ed. Springer Netherlands, 2005.

[23] S. Stuijk et. al, "SDF^3: SDF for free," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, june 2006, pp. 276–278.

[24] S. Bhattacharyya et al, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 151–166, 1999.

[25] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 1st ed. Marcel Dekker, Inc., 2000.

[26] P. Poplavko et. al, "Task-level timing models for guaranteed performance in multiprocessor networks-on-chip," in *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2003, pp. 63–72.

[27] D. Wentzlaff et. al, "On-chip interconnection architecture of the tile processor," *Micro, IEEE*, vol. 27, no. 5, pp. 15–31, sept.-oct. 2007.

[28] J. Carpenter et. al, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *HANDBOOK ON SCHEDULING ALGORITHMS, METHODS, AND MODELS*. Chapman Hall/CRC, Boca, 2004.

[29] H. Chetto et. al, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, pp. 181–194, 1990.

[30] M. Spuri and J. Stankovic, "How to integrate precedence constraints and shared resources in real-time scheduling," *Computers, IEEE Transactions on*, vol. 43, no. 12, pp. 1407 –1412, dec 1994.

[31] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[32] S. Baruah, L. Rosier, and R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, pp. 301–324, 1990.

[33] S. H. Yen et. al, "Efficient algorithms for extracting the k most critical paths in timing analysis," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*.

[34] J. Park and J. A. Abraham, "A fast, accurate and simple critical path monitor for improving energy-delay product in dvs systems," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ser. ISLPED '11. IEEE Press, 2011.

[35] M. Schulz, "Extracting critical path graphs from mpi applications," in *Cluster Computing, 2005. IEEE International*, sept. 2005, pp. 1–10.

[36] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Distributed Computing Systems, 1988., 8th International Conference on*, jun 1988, pp. 366–373.