CAP: Communication-aware Allocation Algorithm for Real-Time Parallel Applications on Many-cores

Milos Panic*,†, Eduardo Quiñones†, Carles Hernandez†, Jaume Abella†, Francisco J. Cazorla†,‡
*Universitat Politècnica de Catalunya (Spain) †Barcelona Supercomputing Center (Spain)
‡Spanish National Research Council (IIIA-CSIC) (Spain)
{mpanic, equinone,chernand,jabella,fcazorla}@bsc.es

***Abstract*—Critical Real-Time Embedded Systems (CRTES) require additional computing power to match the performance requirements of increasingly complex critical functions. Many-core processors are a key solution to reach the required performance. They allow simultaneous execution of multiple critical functions comprising a number of (parallel) applications.**

These applications, each implementing some functionality of the system, need to communicate among them in order to make the system work. In many-cores the impact of this communication on the timing behavior of applications depends on the allocation of applications across the chip. In this paper we propose *CAP*: an allocation algorithm that takes into account communication among applications and tries to reduce its impact on Worst Case Execution Time estimates of applications. We show that using CAP increases the number of applications that can be scheduled on the many-core platform thus facilitating system integration.

I. Introduction

Critical Real-Time Embedded Systems (CRTES) industry is adding complex functionality to their products to keep the competitive edge, e.g. Advanced Driving Assistance System (ADAS) in automotive domain. These complex functions require significant amount of computational power that currently used single-core processors are not able to provide. Therefore, there is an ongoing shift towards multi- and many-core processors in CRTES.

Many-core processors are used in high-performance computing (e.g. Intel SCC[23]) as well as in embedded systems of today (e.g. Tilera [26], Kalray MPPA [12]). However, CRTES industry is yet to find an efficient way of exploiting their potential. This is due to the fact that CRTES require proofs for functional and timing correctness of the systems and their components, which in the case of many-core processors is not trivial.

Providing proofs for timing correctness of a system requires Worst Case Execution Time (WCET) analysis of its applications. Unfortunately, in the case of many-core processors, WCET estimates of applications depend on the co-running applications and the interference they create when accessing shared hardware resources. This creates a circular dependence between timing analysis and scheduling since WCET estimates depend on co-runners selected by the scheduler and scheduling decisions depend on WCET estimates of applications.

One way to break this dependence is to assume an Upper Bound Delay (UBD) [21] when accessing shared hardware resources in order to obtain time-composable WCET estimates, i.e. each access to a shared hardware resource is assumed to suffer the worst possible interference. However, in the case of many-cores, due to the high number of cores that potentially could access a shared resource simultaneously, UBD for accessing that resource can be several orders of magnitude higher than the actual access latency. This leads to overly-pessimistic WCET estimates, thus wasting most of the potential performance of many-cores.

There are 3 main ways to exploit performance potential of many-core processors: (i) speeding-up an application by executing several of its functions across different cores, (ii) speeding-up the system by executing several applications in parallel, and (iii) a combination of both in order to fully utilize many-core potential. We focus on (iii), as future CRTES are expected to concentrate several systems running a number of parallel applications on the same many-core processor [25]. Those systems may easily have highly critical applications with high performance requirements (e.g., unmanned aerial vehicles), and may need to communicate data across multiple applications (e.g., collision avoidance algorithms operating on visual and radar-provided data) [19].

One way to reduce the pessimism of WCET analysis of CRTES applications running on many-cores is performing compositional timing analysis [9], i.e. analyzing the impact of certain components independently and combining their impact to obtain WCET estimates. In [19] authors propose time-compositional analysis of many-core avionics systems by exploiting the fact that the amount of data sent from one application to another is known at system integration time [2] and thus, its impact on WCET estimates can be accounted for at system integration time.

However, determining the most convenient scheduling of applications onto a many-core platform so that the impact of inter-application communication on WCET estimates at integration time is kept low is still a challenge. Therefore, it is of prominent importance devising algorithms tackling this challenge so that an efficient use of the hardware resources is obtained, thus allowing more applications to be integrated onto the same many-core platform, thus reducing procurement, maintenance and power costs as well as size and weight of the CRTES. To the best of our knowledge no specific scheduling algorithm has been proposed for this problem.

In this paper we propose *CAP*: *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on

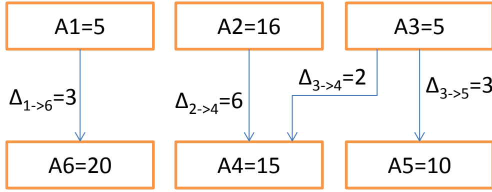

Figure 1: Example of the directed acyclic graph for a CRTES comprising 6 applications

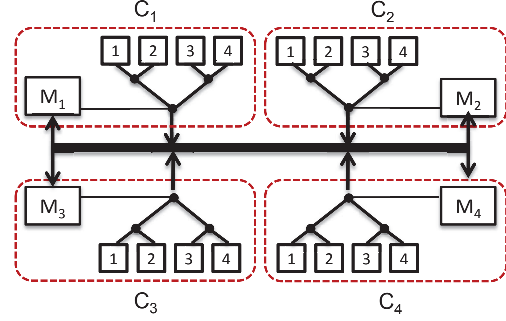

Figure 2: Time-predictable target many-core architecture

Many-cores. *CAP* reduces the impact of communication on guaranteed performance of parallel CRTES applications while facilitating system integration. *CAP* is based on *worst-fit heuristics* used in combination with a non-preemptive time-triggered online scheduler.

CAP constructs the schedule starting from the consumer applications, selecting the ones with highest *consumer weight* metric first. The consumer weight metric sums up computational requirements as well as the amount of inter-application communication across the producer-consumer chains of dependencies that reach this application. *CAP* schedules applications considering the impact of their communications on already scheduled applications iteratively until all applications are scheduled.

We illustrate the concept of CAP by applying it to many-core processor architectures proposed in [19], and evaluate CAP with a set of randomly generated workloads, which is the common practice in the area of scheduling, emulating future CRTES with more than 10 parallel applications. Overall, we show that the use of CAP allows us to schedule up to 29% more workloads on average compared to basic worst-fit heuristic allocation algorithms while facilitating system integration.

II. Background

A. CRTES applications

CRTES consist of several applications exchanging data. We focus on parallel CRTES applications [17], comprising several processes that communicate among them as well as with other applications. Understanding the impact of communication on the timing behavior of the system is one of the main obstacles for the use of many-core processors in CRTES.

When building CRTES applications, developers rely on the industry system software standards upon which applications are built and executed. This is the case of AUTOSAR [3] and ARINC653 [2] standards in the automotive and avionics industry respectively.

In order to facilitate system integration, these standards differentiate 2 types of communication: (i) communication that occurs among the processes of the same application (*intra-application communication*) and (ii) communication that occurs among the processes of different applications (*inter-application communication*). Each type of communication is implemented through a specific API, e.g. in the avionics domain intra-application communication is done through software structures called buffers and inter-application communication is done through software structures called queues. Those standards also require that the amount of inter-application communication is known at system integration time. This requirement allows us to perform a compositional timing analysis and account for the impact of inter-application communication during system integration.

We can exploit the asymmetry among these 2 types of communication in order to obtain tighter WCET estimates and analyze their impact separately. During timing analysis we consider only the impact of intra-application communication and the impact of sending inter-application communication. We defer the analysis of the impact of inter-application communication on the other applications executed concurrently in other clusters until the system integration phase.

Figure 1 shows an example system comprising 6 applications. Each application has a WCET estimate and few of them communicate. (Section III-A provides details on $\Delta_{i \rightarrow j}$, which represents the impact that each inter-application communication has on the timing behavior of the system).

B. Many-core processors

This paper considers a time-analyzable many-core processor in which cores are organized in isles of execution or *clusters* [19] as target for *CAP*. Clusters prevent intra-application communication from different applications to interfere among them and control the interference of inter-application communication. Current many-core embedded processors already provide such functionality [12].

Figure 2 shows a possible realization of the target architecture. It comprises 16 cores, organized in 4 clusters each containing 4 cores and a memory controller. Cores inside the cluster are connected with a tree network on chip (NoC), and clusters are connected with a bus.

Such organization ensures that intra-application communication never leaves the boundaries of the cluster since all intra-application communication requests access the tree NoC and go to the local memory controller. Inter-application communication requests might have to traverse the boundaries of the cluster depending on the allocation of the applications, accessing the bus as well as the memory of other clusters. In [15], we present other realizations of a clustered architecture with mesh NoC.

In order to perform the analysis, the architecture must provide the following properties for inter-application communication:

(i) it is prioritized over intra-application communication, (ii) execution of inter-application communication is transparent to the application executing in the affected cluster. These properties of inter-application communication bound its effect on the application executing in the destination cluster.

Moreover, the target architecture has to provide support for limiting the maximum delay requests can suffer when accessing shared hardware resources – *Upper Bound Delay* (UBD). UBD-based architectures provide results comparable to other time-predictable architectures like TDMA-based ones [11]. As a result, both intra- and inter-application communication requests are bounded by their respective UBDs.

UBD comprises those delays that intra- an inter-application communication suffers at NoC (Worst-Case Traversal Time – WCTT) and memory controller (Worst-Case Response Time – WCRT). In case of intra-application communication, UBD considers only local interference coming from cores in that cluster. Inter-application communication UBD considers only interference coming from other inter-application communication. Since inter-application communication is prioritized over intra-application communication, the WCET estimate of an application, obtained in isolation, has to be augmented by an addend proportional to the amount of inter-application communication that interferes with it (known at system integration).

III. Allocation Algorithm

Here we present the main contribution of this paper: *CAP*, a *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on Many-cores, whose purpose is reducing the impact of communication on WCET estimates of applications while facilitating system integration. We focus on the allocation of parallel CRTES applications in a many-core processor like the one presented in Section II-B.

A. Problem Definition

We focus on those CRTES with the following properties:

- Applications are periodic with period P_i and have implicit deadline ($D_i = P_i$).
- The amount of inter-application communication is known at system integration time, when the scheduling tables are created, and the maximum impact of communication on applications running concurrently in the destination cluster – $\Delta_{i \to j}$ – can be determined.
- Applications do not assume any specific communication pattern (e.g. point-to-point, broadcast, etc.).
- Data coming from other applications must be available before application starts.
- Applications are not preempted and they run until completion.
- Intra-application communication and inter-application communication are explicitly separated in line with AUTOSAR [3] and ARINC653 [2] standards.

Our system can be represented as a directed acyclic graph (DAG) $\sigma = (\mathcal{A}, \mathcal{D})$. The nodes in $\mathcal{A} = \{A_1, ..., A_n\}$ represent the applications that compose the system. Each application A_i is characterized with a WCET estimate C_i obtained in isolation, The utilization u_i of application A_i is defined as $\frac{C_i}{P_i}$, where $0 \leq u_i \leq 1$. The edges in $\mathcal{D}$ represent inter-application communication, in which $\Delta_{i \to j} \in \mathcal{D}$ represents the precedence constrains among the nodes in $\mathcal{A}$, such that A_j cannot start executing until A_i finishes. It is important to remark that the use of DAGs allows representing a wide range of communication patterns, including point-to-point, broadcasting, etc.

The weight of an edge $\Delta_{i \to j} \in \mathcal{D}$ represents the maximum impact of inter-application communication between A_i and A_j on the WCET of any arbitrary affected application A_m running concurrently with A_i and allocated to the same cluster as A_j. An inter-application communication request (from A_i to A_j) delays intra-application requests increasing the WCET estimate of the affected application A_m by a value I_k. I_k is upper-bounded by inter-application communication request UBD [19].

Therefore, the maximum impact that inter-application communication among applications A_i and A_j can have on application A_m running in the destination cluster, marked as $\Delta_{i \to j}$, is the addition of the impact of each I_k, as shown in Equation 1:

$$\Delta^m{}_{i \to j} = \sum_{k=1}^{N_{req}} I_k \quad (1)$$

where N_{req} is the number of requests of inter-application communication among applications i and j. Then, during system integration, when the allocation of applications is known, the WCET estimate of the affected application A_m must be increased by the corresponding $\Delta_{i \to j}$ values.

For the sake of clarity and to ease the explanation, we assume that all applications have the same deadline. If this was not the case, the DAG of our system would comprise all instances of the applications during one hyper-period of the system (least common multiple of application periods) and we would have to slice it into time slots and apply *CAP* to each time slot independently, following a similar approach to the ones shown in [20], [16].

CAP assigns the *n* parallel applications in $\mathcal{A}$ to a set of *m* identical clusters $sc = \{c_1, \cdots, c_m\}$, respecting precedence constraints in $\mathcal{D}$. Clusters isolate intra-application communication and bound inter-application communication (see Section II-B). *CAP* generates a static partition $\Phi = (\varphi_1, \cdots, \varphi_m)$ in which a subset of $\mathcal{A}$ is assigned to a cluster c_i. Each application can be assigned to only 1 cluster.

In order to guarantee that precedence constraints are respected, *CAP* has to allocate application A_j after every application $A_i \in \mathcal{A}$ finishes if there is inter-application communication from A_i to A_j ($\exists \Delta_{i \to j} \in \mathcal{D}$). Application A_i is called producer application if there is an application A_j that uses the results produced by A_i. A_j is then called consumer application.

B. Mapping Applications to Clusters

Figure 3 shows a pseudo-code implementation of *CAP*. The algorithm takes a DAG $\sigma = (\mathcal{A}, \mathcal{D})$ and a set of m clusters

CAP allocation algorithm
Input $\sigma = (\mathcal{A}, \mathcal{D})$: A DAG of the system ,
$sc = (c_1, \cdots, c_m)$: a set of clusters

Output $\Phi = (\varphi_1, \cdots, \varphi_m)$: A valid allocation of σ into sc

```
1  Φ = ∅
2  deltas = ∅
3  set_consumer_weights(σ)
4  forall (A_i ∈ 𝒜|A_i ∉ Φ; ∄A_j ∈ 𝒜; Δ_{i→j} ∈ 𝒟; A_j ∉ Φ)
5
6    select A_i with highest cw_i from 𝒜
7    update_predecessors_consumer_weight(A_i)
8    A_i.start_time = A_i.end_time = 0
9    A_i.end_time = lat_start_time (A_i.successors())
10
11   if (∃ idle_slot ∈ φ_j | idle_slot.size > C_i +
          deltas(idle_slot) and idle_slot.end_time ≥
          A_i.end_time)
12     A_i.end_time = idle_slot.end_time
13     A_i.start_time = A_i.end_time + C_i + deltas(
           idle_slot)
14     Φ += allocate(c_j, A_i)
15   else
16     c_j = worst_fit(A_i)
17     apply_deltas(c_j, A_i)
18     Φ += allocate(c_j, A_i)
19   endif
20   forall (A_k ∈ Φ and A_k ∉ φ_j)
21     deltas += Δ_{i→k}
22 endfor
23 return Φ;
```

Figure 3: Pseudo-code implementation of the allocation algorithm.

$sc = \{c_1, ..., c_m\}$ as the input and provides a valid allocation Φ as the output.

CAP allocates a set of application inside a given time slot (in this case equal to the deadline of applications). It constructs the schedule by assigning offsets from the end of the given time slot to application. It starts the allocation with consumer application first. This facilitates accounting for inter-application communication impact $\Delta_{i \to j}$. At the moment when producer application is allocated, its consumer is already allocated and *CAP* can detect whether $\Delta_{i \to j}$ impacts other applications. Note that once *CAP* produces the schedule, the whole schedule can be shifted to the the beginning of the time slot, if the system integrator prefers having slack at the end of the time slot, instead of at the beginning.

CAP starts by assigning *consumer weights* to all applications in $\mathcal{A}$ (line 3). Consumer weight of application A_l (cw_l) is initially computed as the highest sum of all C_k and $\Delta_{i \to j}$ across all chains of dependencies that have application A_l as the consumer. For example in Figure 1, initial consumer weight of A_4 is computed as $cw_4 = max(16{+}6{+}15, 5{+}2{+}15) = 37$, while the rest of them are $cw_1 = 5, cw_2 = 16, cw_3 = 5, cw_5 = 18, cw_6 = 28$. We use the consumer weight metric to identify consumers that belong to the longest chains of dependencies and create highest pressure on the allocation algorithm possibly containing $\Delta_{i \to j}$ with high impact.

CAP does all allocations in a single loop. Before each iteration of the main loop (line 4), *CAP* updates the list of applications that are ready for allocation. It puts the following types of applications into the list: (i) independent applications, (ii) consumer only applications and (iii) producers with all of their consumers already allocated. Among them it selects the one with highest consumer weight metric (line 6) for allocation. In the example in Figure 1, applications A_4, A_5 and A_6 are ready for allocation and *CAP* selects A_4 since it has the highest consumer weight.

Once the application is allocated, *CAP* updates consumer weight metric for all of its predecessors (line 7). In this step, we add the $\Delta_{j \to i}$ to consumers weight of A_i predecessor for each communication in which A_i is consumer. It helps us keeping track of high $\Delta_{j \to i}$ values when choosing the next application to allocate. In the example, selecting A_4 leads to updating consumer weights of A_2 and A_3: $cw_2 = cw_2 + \Delta_{2 \to 4} = 22; cw_3 = cw_3 + \Delta_{3 \to 4} = 7$.

In order to allocate the application, we compute its starting and ending time in the schedule (expressed as the offset from the end of the time slot). Since we start from the end of the time slot, we try to allocate applications as late as possible in the schedule. We determine the latest ending time of the application A_i by examining the earliest starting time of its successors in σ (line 9) to guarantee that all producer applications will finish before any consumer starts, e.g. A_3 has to finish before A_4 and A_5 start.

CAP uses starting and ending times of an application A_i in order to delimit the interval of the time slot when the applications affected by $\Delta_{j \to i}$ have to be given additional resources to compensate for the impact of $\Delta_{j \to i}$ (see Section III-C and Figure 4(e)). Also those applications that could be potentially affected by inter-application communication will have their starting time shifted and therefore the difference between its start_time and end_time will be greater than their respective WCET estimates.

After computing the latest ending time of an application, *CAP* assigns the application to a cluster. In order to do this allocation in a single loop and maintain low complexity, *CAP* looks for idle bubbles in the schedule and tries to fit the current application there (lines 11-14) respecting the dependencies and WCET constraints. Idle bubbles in the schedule exist if the application cannot start as late as it could in the assigned cluster but has to start earlier due to dependencies. This step is designed for small chains of dependencies and independent applications that are allocated late in the algorithm and tries to maximize utilization of the processor.

If *CAP* cannot find a suitable idle bubble in the schedule, it uses worst-fit heuristics to choose the cluster where A_i is allocated (line 16). We check if there are existing $\Delta_{k \to j}$ in the interval between start and ending time of A_i in the cluster c_j. If there is any inter-application communication targeting the cluster c_j during that interval, we update the starting time of A_i to accommodate a $\Delta_{k \to j}$ that could affect the application (line 17). For example, when allocating A_1 to c_1, we have to shift A_1 start time to accommodate for $\Delta_{3 \to 4}$ (see Section III-C and Figure 4(e)). Then we allocate the application to the cluster c_j (line 18).

The last step of the loop consists of adding all $\Delta_{j \to k}$ to

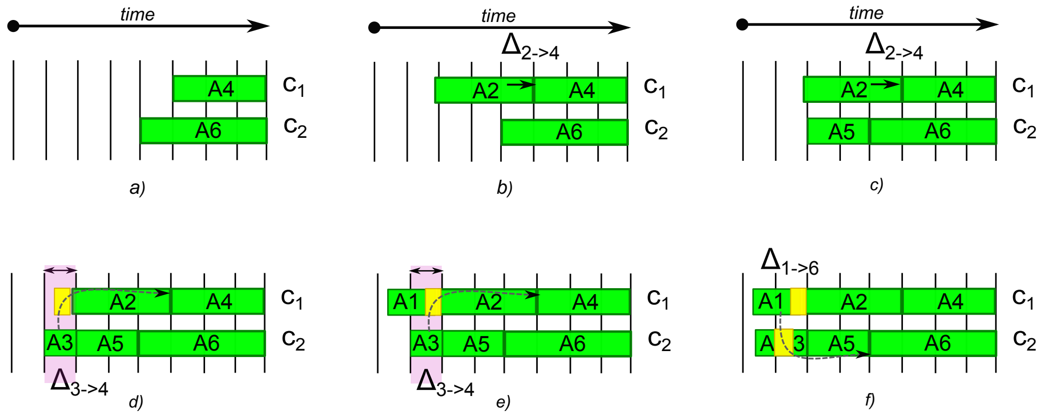

Figure 4: Allocation of the example applications from Figure 1

the list of inter-application communications that cross cluster boundaries and creating *zones of communication impact* in the schedule as well as updating existing ones (lines 20-21).

C. Example

Figure 4 illustrates how *CAP* allocates the example in Figure 1 into a 2-cluster many-core. In the first 2 steps, *CAP* selects applications with highest consumer weights $A_4 = 37$ and $A_6 = 28$, allocates them to clusters c_1 and c_2 respectively (Figure 4(a)). It updates consumer weights of their predecessors by adding $\Delta_{2->4} = 6$ to consumer weight of A_2 making it $cw_2 = 22$, as well as consumer weights of A_1 and A_3.

After this, A_2 has the highest consumer weight, and *CAP* allocates it to c_1 (Figure 4(b)). There is communication between applications A_2 and A_4, but they are allocated to the same cluster c_1. Thus, this communication has no impact on any other application (its impact is included in the WCET estimate of A_2 and it does not use resources of any other application). Since there are no predecessors of A_2 in the DAG, *CAP* does not update any consumer weights and it selects the application with highest consumer weight, i.e. A_5 and allocates it to the cluster c_2 (Figure 4(c)).

The next application for allocation is A_3. Based on worst-fit heuristics, *CAP* allocates it to cluster c_2. Since $\Delta_{3\to4}$ exists and applications A_3 and A_4 are allocated to different clusters, *CAP* creates an interval of communication impact in cluster c_1. This interval (marked purple in Figure 4(d)) makes *CAP* shift the start of the applications affected by the amount of interference that can be created inside it ($\Delta_{3\to4}$ - marked yellow). A_2 was already allocated inside this interval and its start has to be shifted by $\Delta_{3\to4} = 2$ time units to compensate for the impact inter-application communication between A_3 and A_4.

Finally, *CAP* allocates A_1 to c_1 (see Figure 4(e)). A_1 communicates with A_6 allocated to c_2. Again, *CAP*, same as before, treats $\Delta_{1\to6}$ creating another zone of communication impact in cluster c_2. This zone causes the shift of the start time of A_3 and *CAP* must ensure that the zone of communication impact created by A_3 is also updated accordingly. Figure 4(f) represents the final allocation of the example.

In this example $\Delta_{3\to4}$ affects multiple applications, A_1 and A_2. In order to avoid "double accounting" of communication impact (once per each application) as well as to prevent starting A_2 application while A_1 has not finished, *CAP* requires simple support from the operating system (online scheduler). *CAP* has to detect cases where 2 (or more) applications are affected by 1 zone of communication impact, e.g. A_1 and A_2 in Figure 4(e). Starting times of applications have to be shifted by $\Delta_{3\to4}$ and the operating system must start A_2 only if: (i) the start time of A_2 has passed and (ii) application A_1 has finished.

IV. Evaluation methodology

CAP targets future CRTES comprising several parallel applications. To evaluate its effectiveness we use randomly generated application-sets and allocate them to the clustered many-core processors presented in Section II-B.

In order to better resemble the communication requirements of real systems, the randomly generated application-sets are based on the avionics system presented in [19], and comprising 3D obstacle and stereo camera image generators, that create input for 2 collision avoidance parallel applications. An additional application checks the results of collision avoidance applications and compares them.

In order to emulate a more complex system with higher workload, we consider two different scenarios: (i) randomly-generated application-sets comprising between 11 and 16 parallel applications allocated onto a 16-core, 4-cluster many-core; and (ii) randomly-generated application-sets comprising between 43 and 64 parallel applications allocated to a 256-core, 16-cluster many-core processor. In both scenarios, the targeted many-core architecture is the one presented in Figure 2.

Our random application-set generator creates DAG representing application-sets with a given utilization level assuming that applications fully utilize clusters resources assigned to them. This means that a parallel application utilizes all the cores available in the cluster where it is assigned and that clusters execute only one application at a time. WCET estimates of applications are random values from an interval. Impact of communication among applications is also a random value with the following constraint: randomly created communication edges cannot form loops in the graph.

	WCET(cycles)	$\Delta_{i->j}$
Compositional	500,000-3,000,000	100,000-3,000,000
Composable (1.5-2.15)	750,000-6,450,000	N/A
Composable (1.3-1.6)	650,000-4,800,000	N/A

Table I: WCET intervals in cycles, assuming 1GHz processor

CAP allows the use of compositional timing analysis and much tighter WCET estimates w.r.t. traditional time-composable WCET. In order to fairly evaluate *CAP*, we create a time-composable copy of randomly created DAGs.

Table I shows the intervals used by our random generator to choose WCET estimates of the applications and $\Delta_{i->j}$. As shown in Table I, we assume a significant amount of inter-application communication, in order to create more pressure on *CAP*. In the case of the time-composable approach, we consider 2 scenarios: row 2 of Table I represents the case from [19], where time-composable WCET estimates of two industrial parallel avionics applications presented in [19] are 1.5x and 2.15x higher w.r.t. to compositional ones, while row 3 represents an optimistic case where it is possible to conduct an improved composable timing analysis so that time-composable WCET estimates are only between 1.3x and 1.6x higher than compositional ones. In case of the time-composable approach, the weight of the communication impact edges is set to 0, since the WCET estimates already account for all possible program interactions, but the edges are maintained to keep the precedence constraints.

Then we allocate the initial DAG with *CAP* and its time-composable counterpart with a *b*asic *a*llocation algorithm based on *w*orst-*f*it heuristics (*BAWF*).

BAWF works similarly to the algorithm presented in [20]. It allocates applications to clusters based on their time-composable WCET estimates (rows 2 and 3 of Table I). For each application it selects the cluster with lowest scheduled utilization respecting preceding constraints.

V. Results

We evaluate *CAP* by applying it to randomly generated application-sets allocating them to a many-core processor with 4 clusters as presented in Section II-B. We also apply it to a larger version of this processor, comprising 16-clusters with 16 cores each. A currently used processor with these number of cores and clusters is Kalray MPPA [12].

A. 4-cluster many-core

In the case of 4-cluster many-core, *CAP* is used with a series of randomly generated application-sets. For each utilization value in [1.92, 4), with an utilization increment of 0.08, we create 1,000 application-sets, 30,000 in total. Each application-set comprises between 11 and 16 applications and between 7 and 12 inter-application communication dependencies. Utilization of 1.92 means that 48% of the clusters are used by an application-set, and utilization of 4 means that all 4 clusters are fully utilized during the time-slot.

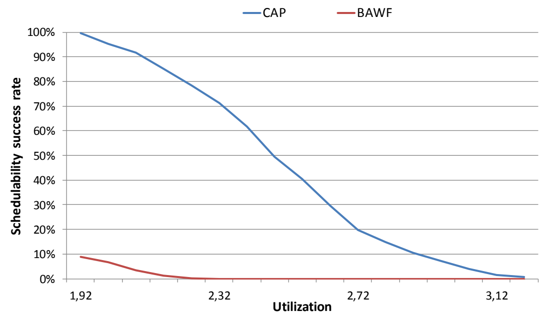

Figure 5: Schedulability success rate - CAP vs. BAWF (Composable 1.5-2.15) on a 4-cluster many-core

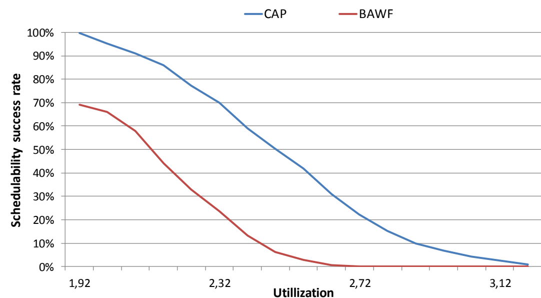

Figure 6: Schedulability success rate - CAP vs. BAWF (Composable 1.3-1.6) on a 4-cluster many-core

We compare *CAP* against a time-composable approach using worst-fit heuristics - *BAWF*.

Figure 5 compares the schedulability success rates[1] of application-sets using *CAP* (labeled as *CAP*) and the basic worst fit algorithm (labeled as *BAWF*) considering the scenario in row 2 of Table I when allocating tasks to a 4-cluster many-core. Utilization increases from 1.92 up to 3.2. We observe that *CAP* is superior to *BAWF* in this scenario, being able to allocate most of the application sets (95.7%) at utilization 2, while *BAWF* is able to allocate only 6.7% of the application sets. *CAP* schedulability success rate decreases as we increase the utilization, but it is still able to allocate around 50% of the application sets at utilization 2.56.

Figure 6 compares *CAP* and *BAWF* in an optimistic scenario (row 3 of Table I). *CAP* still outperforms *BAWF* by allocating 28.9% additional application-sets on average. In the case of an utilization around 2.64, *CAP* is able to allocate around 40% of the application-sets, whereas *BAWF* can hardly allocate few of them.

B. 16-cluster many-core

In the case of 16-cluster many-core, *CAP* is used with a series of randomly generated application-sets. For each utilization value in [4.8, 12.8), with an utilization increment of 0.32, we create 1,000 application-sets. Each application-set comprises between 43 and 64 applications and between 40 and 60 inter-application communication dependencies. Utilization of 4.8

[1] Schedulability success rate represents the percentage of the application sets that an algorithm is able to allocate at a given utilization level.

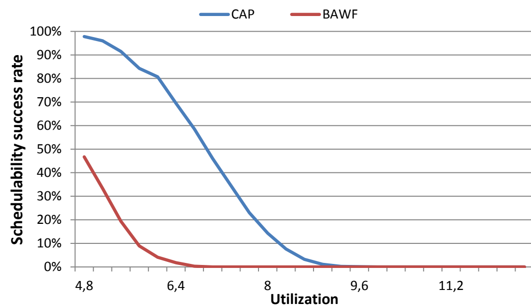

Figure 7: Schedulability success rate - CAP vs. BAWF (Composable 1.5-2.15) on a 16-cluster many-core

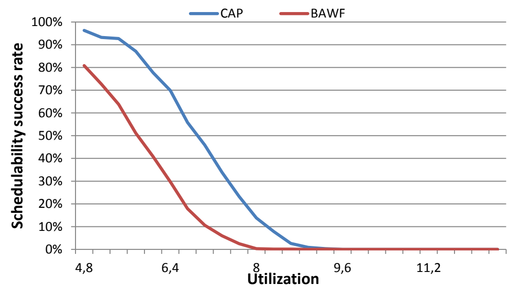

Figure 8: Schedulability success rate - CAP vs. BAWF (Composable 1.3-1.6) on a 16-cluster many-core

means that 30% of the CPU capacity is used by an application-set, and utilization of 12.8 represents 70% of the CPU capacity.

Figure 7 shows the schedulability success rates of application-sets using *CAP* and *BAWF* considering the scenario in row 2 of Table I when allocating applications to a 16-cluster many-core. Utilization increases from 4.8 up to 12.8. We observe that *CAP* outperforms *BAWF* in this scenario, being able to allocate most of the application sets (97.8%) at utilization 4.8, while *BAWF* is able to allocate only 46.7% of the application sets. Even though, both algorithms under-utilize many-core resources in this case, *CAP* can still allocate around 50% of the application sets at utilization of 7.04.

Figure 8 shows the more optimistic scenario (row 3 of Table I) for *BAWF* algorithm, when it works with tighter time-composable WCET estimates. *CAP* again has higher schedulability success rates w.r.t. *BAWF*, being able to allocate 26.8% more application sets on average.

In the case of 16-cluster many-cores, the difference between time-composable and time-compositional WCET estimates could be higher than the values we extracted from [19] due to the higher core count and higher interference. If that was the case, the advantage of *CAP* would become even more obvious.

We can observe that utilization of a 16-cluster many-core is low (Figures 7 and 8). This is an inherited limitation of partitioned time-triggered non-preemptive scheduling, required by the time-compositional analysis from [19]. Extending the analysis and *CAP* to support preemption as well as other techniques for improving utilization [14] remains as future work and one of the obstacles for using high core-/cluster-number many-cores in CRTES.

C. Algorithm complexity

Even though *CAP* is an offline allocation algorithm, and its performance is not crucial for the performance of the system, it is a light-weight allocation algorithm. Allocating a 60+ application set from previous section to a 16-cluster many core takes less than 10 seconds in a typical laptop.

All allocations are done in a single loop (line 4 in the algorithm 4). At the beginning of the loop, there is a search through a list of applications for the one with highest consumer weight metrics (line 6). These 2 operations define algorithm's complexity as $O(N^2)$, where N is the number of applications in the application set. Note that there is another loop (lines 20-21), but since $k < N$, it does not affect algorithm complexity.

VI. Related Work

Scheduling of parallel CRTES applications to clustered many-core processors relates to scheduling of processes to multi-core processors. We can split research on multi-core scheduling [4], into 2 categories w.r.t. allocation of processes to cores: (i) partitioned approaches that allocate each process to a single core and later use single processor scheduling for each core and (ii) global approaches that allow processes to migrate from one core to another at run-time.

We opted for a partitioned approach to benefit from predictability of inter-application communication and to improve WCET estimates of applications and throughput of the system. Finding an optimal allocation using partitioned approaches is an NP-hard problem [6] and so sub-optimal solutions are derived using, for instance, bin-packing heuristics [1], [5], [28].

Most scheduling works proposed in the CRTES domain consider independent processes that do not communicate among them. Lakshaman et al. [14] presented a preemptive fixed-priority partitioned scheduling for multi-cores that relies on task-splitting to improve utilization bounds. However, this approach requires using time-composable WCET estimates and has additional costs of preemption and task migration, mitigating the benefits of the compositional analysis from [19].

Paolieri et al. [22] present an interference-aware allocation algorithm that uses multiple WCET estimates per application when constructing the schedule. The WCET estimate value is chosen from a structure called WCET-matrix that contains a set of WCET estimates obtained under different execution scenarios. *CAP* requires only 1 WCET estimate per application and takes into account only the interference of inter-application communication when creating the schedule.

Wieder and Brandenburg [27] propose a real-time partitioned scheduling of independent tasks in which accesses to the shared resources are protected with spin locks. They provide an ILP formulation for finding optimal partitioning w.r.t. schedulability analysis of the MSRP protocol [18]. Their approach cannot be applied to these application-sets, as they have run-after data dependencies and would require use of synchronization mechanisms across applications, with which the analysis presented in [19] is incompatible.

Along this line, in [7], [8] authors propose scheduling and mapping of mixed-criticality applications on multi-core

platforms. They present the global time-triggered scheduling algorithm that uses barriers for synchronization. However, they allow concurrent execution of applications from only 1 criticality level, in order to derive timing guarantees. *CAP* does not impose such a restriction, since it is only required to know the amount of communication among applications, regardless of the criticality level.

In the automotive domain, there are few recent multi-core scheduling proposals [16], [10], [20]. Monot et al. [16] present a scheduling algorithm for multi-source AUTOSAR applications that allows communication only for functions executing on the same core, so each core can be scheduled independently. Faragardi et al. [10] present a scheduler that reduces communication delays for AUTOSAR applications on multi-cores and Panic et al. [20] present an allocation algorithm that parallelizes legacy single-core AUTOSAR applications. All those solutions consider time-composable WCET estimates, and do not consider the impact of communication on the affected applications, limiting the performance potential of many-core processors. Sinnen et al. [24] propose a task scheduling mechanism similar to [20], that targets general purpose processor architectures where the task scheduler is aware of the cost of inter-processor communication, addressing the system throughput instead of the WCET of applications.

In the avionics domain, Kim et al. [13] propose a scheduler for multiple IMA applications on a multi-core. This proposal considers inter-application communication, but imposes a severe constraint: while inter-application communication executes nothing else is executed in other cores.

VII. Conclusions

In this paper we present *CAP*, a *C*ommunication-aware *A*llocation Algorithm for Real-Time *P*arallel Applications on Many-cores, that significantly reduces the impact of communication on guaranteed performance of parallel CRTES applications while facilitating system integration.

CAP exploits the fact that inter-application communication is known at system integration time enabling the use of tight WCET estimates. It constructs the schedule starting from consumer applications first giving higher priority to those belonging to the chains of dependencies with longest utilization. For each inter-application communication that crosses the cluster boundaries, *CAP* creates a zone of communication impact inside which applications allocated to the destination cluster are given additional computational resources.

We evaluate *CAP* with sets of randomly-generated application-sets, based on the case studies presented in [19] that represent future complex CRTES targeting clustered many-core processors [12], [19]. We compare *CAP* with a baseline state-of-the-art allocation algorithm that uses time-composable WCET estimates. *CAP* is able to allocate up to 29% more workloads on average to many-core processors with 4-clusters compared to the baseline algorithm.

Therefore, *CAP* stands as an important step towards the efficient use of many-core processors in CRTES so that their performance potential can be fully exploited.

Acknowledgments

The research leading to these results has been funded by the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA). This work has also been funded by the Ministry of Science and Technology of Spain under contract TIN2012-34557. Miloš Panić is funded by the Spanish Ministry of Education under the FPU grant FPU12/05966. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] A. Burchard, et. al. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44(12), 1995.

[2] ARINC Inc. *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4*, 2012.

[3] AUTOSAR GbR. *AUTomotive Open System ARchitecture (AUTOSAR)*, Jan. 2013.

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

[5] S. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operation Research*, 1978.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. Macmillan Higher Education, 1990.

[7] G. Giannopoulou, et. al. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT*, 2013.

[8] G. Giannopoulou, et. al. Mapping mixed-criticality applications on multi-core architectures. In *DATE*, 2014.

[9] S. Hahn, et. al. Towards compositionality in execution time analysis – definition and challenges. In *CRTS*, 2013.

[10] H.R. Faragardi, et. al. Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores. In *ETFA*, 2013.

[11] J. Jalle, et. al. Deconstructing bus access control policies for real-time multicores. In *SIES*, 2013.

[12] Kalray. *Kalray MPPA 256 Many-Core Processor*.

[13] J. Kim, et. al. Optimized scheduling of multi-ima partitions with exclusive region for synchronized real-time multi-core systems. In *DATE*, 2013.

[14] K. Lakshmanan, et. al. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*, 2009.

[15] M. Panic, et. al. Parallel many-core avionics systems. Technical Report UPC-DAC-RR-2014-13, Universitat Politecnica de Catalunya.

[16] A. Monot, et. al. Multisource software on multicore automotive ecus - combining runnable sequencing with task scheduling. *IEEE TIE*, 2012.

[17] H. Ozaktas, et. al. Automatic WCET analysis of real-time parallel applications. In *WCET*, 2013.

[18] P. Gai, et. al. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS*, 2001.

[19] M. Panic, et. al. Parallel many-core avionics systems. *EMSOFT*, 2014.

[20] M. Panic, et. al. Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores. *CODES+ISSS*, 2014.

[21] M. Paolieri, et. al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[22] M. Paolieri, et. al. Ia^3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS*, 2011.

[23] J. Rattner. Single-chip cloud computer: An experimental many-core processor from Intel Labs.

[24] O. Sinnen, et. al. Toward a realistic task scheduling model. *IEEE Trans. Parallel Distrib. Syst.*, 2006.

[25] T. Ungerer, et. al. parMERASA - multi-core execution of parallelised hard real-time applications supporting analysability. In *DSD*, 2013.

[26] Tilera Corporation. *Tile Processor, User Architecture Manual, release 2.4, DOC.NO. UG101*, 2011.

[27] A. Wieder and B. B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *SIES*, 2013.

[28] Y. Oh and S.H.Song. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Syst.*, 1995.