Towards No-cost Adaptive MPSoC Static Schedules through Exploitation of Logical-to-Physical Core Mapping Latitude

Chengmo Yang and Alex Orailoglu
Computer Science and Engineering Department
University of California, San Diego
9500 Gilman Drive, La Jolla, CA 92093
{c5yang, alex}@cs.ucsd.edu

***Abstract*—The computing engines of many current applications are powered by MPSoC platforms, which promise significant speedup but induce increased reliability problems as a result of ever growing integration density and chip size. While static MPSoC execution schedules deliver predictable worst-case performance, the absence of dynamic variability unfortunately constrains their usefulness in such an unreliable execution environment. *Adaptive static schedules* with predictable responses to runtime resource variations have consequently been proposed, yet the extra constraints imposed by adaptivity on task assignment have resulted in schedule length increases. We propose to eradicate the associated performance degradation of such techniques while retaining all the concomitant benefits, by exploiting an inherent degree of freedom in task assignment regarding the logical to physical core mapping. The proposed technique relies on the use of core *reordering* and *rotation* through utilizing a graph representation model, which enables a direction translation of inter-core communication paths into order requirements between cores. The algorithmic implementation results confirm that the proposed technique can drastically reduce the schedule length overhead of both *pre-* and *post-* reconfiguration schedules.**

I. Introduction

The advances in VLSI fabrication technologies have placed System-on-Chip (MPSoC) center stage as the dominant architecture in the next computer generation. Our ability to exploit such an extant computational power, however, is checkmated not only by parallelism extraction limitations of our current techniques, but furthermore by increasing levels of execution uncertainty within the system. As technology has advanced from 180nm to 45nm, the rate of transient and intermittent faults has increased by three orders of magnitude (from 10^{-6} to 10^{-3}) [1], [2]. Heat buildup, as it can make cores temporally unavailable during execution, furthermore degrades chip resource availability. On the other hand, such an unreliable platform is used to concurrently hold an increasing number of applications that constantly vie for execution resources on the MPSoC, thus furthermore requiring resource demands to be frequently renegotiated at run-time.

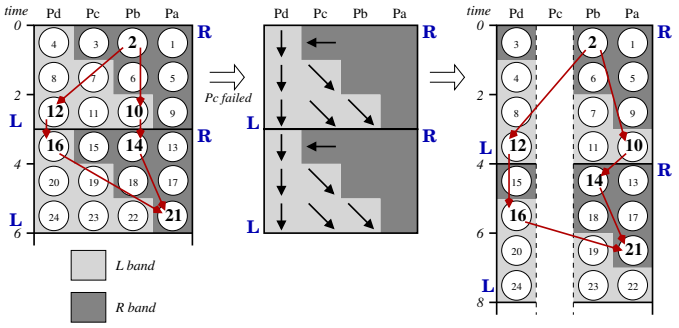
The increasing possibility of resource variations requires the consideration of *execution adaptivity* as a primary design constraint for MPSoCs. Although such adaptivity can be straightforwardly attained by adopting a *dynamic* scheduling approach [3], due to the lack of global program information, such an approach can only make local-optimal decisions that unpredictably affect the overall performance. Static scheduling approaches, on the other hand, offer a set of benefits to embedded applications, including the concealment of scheduling latency, the ability to use complex heuristics, and the predictability of worst case schedules. A number of static scheduling heuristics have been developed for parallel systems [4], with the applicability of these heuristics to MPSoCs also being examined recently [5], [6]. Nonetheless, all these heuristics share a common characteristic in that the generated schedules are confined to the case of a fixed number of processing elements (PEs). This absence of dynamic variability crucially limits the usefulness of such schedules in a more unreliable and unpredictable execution environment since a resource variation, especially a *PE deallocation*, usually dooms the entire schedule to uselessness.

In order to smoothly vary the number of cores dedicated to an application, *dynamic reconfigurability* needs to be incorporated into static MPSoC schedules. The reconfiguration process furthermore should be very fast and highly predictable so as to minimize the extra performance overhead and to meet the timing constraints of real-time applications. To accomplish this challenging task, adaptive static schedules [7], [8] have consequently been proposed. By executing sophisticated planning during scheduling, such static schedules are able to deliver regular and predictable responses to various resource availability constraints. Upon a runtime resource variation, tasks are transferred between adjacent cores in a regular manner, yet no rescheduling decisions need to be made on the fly, thus drastically reducing reconfiguration overhead. However, the attainment of such a regular reconfigurability requires extra constraints to be imposed during task assignment so as to preclude potential semantic violations, thus probably resulting in schedule length increases.

An important aspect whose consideration has been lacking in previous techniques is that in adaptive static schedules, the existence and quantity of extra timing slacks is determined by the spatial position of PEs. Accordingly, we propose in this paper to exploit an inherent degree of freedom in task assignment regarding the logical to physical PE mapping, which has remained unexploited in previous work, so as to eradicate the schedule length increases imposed by adaptivity while retaining all the concomitant benefits. The adjustment of PE order is not only able to change the direction of an inter-task dependence, but also to preserve the spatial locality associated with tightly scheduled dependent tasks, thus eliminating the need for extra timing slack to ensure semantic correctness. These two benefits are achieved through the use of a *graph representation model* that enables a direction translation of inter-PE communication paths into order requirements between PEs.

The rest of the paper is organized as follows. Section 2 outlines the technical motivation. Section 3 describes in detail the proposed PE remapping approaches, which are incorporated into a representative scheduling algorithm in Section 4. Section 5 experimentally verifies the efficacy of the proposed technique in the context of single processor failures, while section 6 offers a set of brief conclusions.

978-3-9810801-5-5/DATE09 © 2009 EDAA



(a) Band−level partition (b) Reassignment directions (c) Schedule after reconfiguration

Fig. 1. BB reconfiguration and its impact on inter-task dependences

II. Technical Motivation

The increasing possibility of resource variations, together with the global optimization capability and predictable worst-case performance offered by static scheduling techniques, has motivated the development of adaptive static schedules. To achieve dynamic adaptivity, both *pre-* and *post*-reconfiguration schedules should be able to not only preserve the partial execution ordering imposed by inter-task dependences, but furthermore utilize the available hardware resources maximally. Previous research [7] attained these goals through partitioning a static schedule into regular yet shiftable structures, while imposing extra constraints on the *timing* and *spatial* characteristics of inter-task dependences, thus engendering possible schedule length increases.

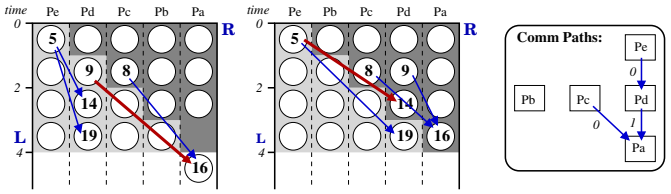
In the ***Block & Band (BB) reconfiguration*** scheme outlined in [7], predictable dynamic reconfigurability is attained through conceptually imposing a *band* structure on static MPSoC schedules. Essentially, a static schedule is partitioned into multiple *Basic Reconfiguration (BR) blocks*, while each BR block is furthermore divided into a *Left (L) band* and a *Right (R) band*. This band-level partitioning method is presented in Figure 1a, with each cycle labeled with a number denoting a task and each column representing one processing element (PE) of the MPSoC. More importantly, the shape of the L and the R band furthermore provides a highly regular task reassignment capability, achievable **independent** of the PE being removed. As an example, Figure 1b uses arrows to indicate the shifting directions of all tasks in order to tolerate the deallocation of P_c. It can be seen that in each schedule block the position of the R band remains intact, while the whole L band is shifted one timing step down and one PE to the right. As a result, in the *post*-reconfiguration schedule presented in Figure 1c, all the tasks within each *BR block* can be completed with one fewer PE, albeit with an additional timing step. Moreover, both the pre- and the post-reconfiguration schedules are able to make **full utilization** of the available hardware resource, since each *BR block* contains $n*(n-1)$ tasks that can be executed either by n PEs in $(n-1)$ timing steps, or by $(n-1)$ PEs in n timing steps.

An important benefit of the adaptive schedules is the inherent regularity. During the reconfiguration process, not only are task movements restricted to shiftings between adjacent PEs, but all the tasks within a single band continue to share the identical timing offset. Accordingly, dependent tasks lying within the same band, such as the task pair (14, 21) in Figure 1a, are able to retain their relative timing position after reconfiguration. Meanwhile, as the L bands are shifted downwards relative to the R bands, the timing slack associated with an *RtoL* dependence (e.g., the one from Task 2 to Task 12 in Figure 1a) will always *increase* after reconfiguration, while the timing slack associated with an *LtoR* dependence (e.g., Task 16 → Task 21 in Figure 4a) will always *decrease*. Accordingly, an *LtoR* dependence may be violated if the original timing slack is insufficient. As can be seen in Figure 1b, reconfiguration causes the original timing slack between the task pair (16, 21) to disappear, thus leaving insufficient time for Task 21 to receive its input.

In addition to the potential *timing* violations of *LtoR* dependences, adaptive static schedules may also suffer from potential *spatial* variations in inter-processor communications. While traditionally a static scheduler prefers to assign two dependent tasks on the same PE to hide communication latency, two dependent tasks originally scheduled on the same PE may be separated onto two PEs after reconfiguration. This separation in turn creates extra inter-PE communications in the post-reconfiguration schedule, as can be seen from the positions of the dependent task pairs $(2, 10)$ and $(10, 14)$ in Figure 1. More crucially, if the source and the sink tasks (e.g., Task 10 to 14 in Figure 1) are tightly scheduled originally and their relative timing positions remain intact, the inter-PE communication created after reconfiguration will also result in semantic violations.

The preclusion of the aforementioned two types of semantic violations necessitates an explicit timing slack to be inserted between the source and sink tasks, at the cost of course of possible schedule length increases. On the other hand, the analysis outlined also confirms that the potential violations of inter-task dependences are caused by the structural partitions imposed on the adaptive static schedule, which is in turn determined by the spatial position of PEs. This property offers an extra degree of freedom in task assignment for us to exploit, namely, *the logical to physical PE binding order*. Specifically, the adjustment of the PE binding order can vary the direction of a critical *LtoR* dependence, thus enabling the preservation of the associated timing slack and spatial locality in the post-reconfiguration schedule. This benefit is illustrated in the example presented in Figure 2. Figure 2a shows that Task 16 cannot be scheduled in the free timing slot on P_a, as the dependence between Tasks 9 and 16 is an *LtoR* dependence. However, Figure 2b shows that by switching the positions of P_b and P_d, the direction of the dependence between Tasks 9 and 16 is no longer *LtoR*, thus enabling Task 16 to be scheduled in the free timing slot on P_a.

While PE reordering can vary the direction of the *LtoR* dependence under consideration, its effect is *globally* imposed on all the inter-PE task dependences. As a result, a *non-LtoR* dependence, such as the dependence between Tasks 5 and 14 in Figure 2a, may end up becoming an *LtoR* dependence in the post-reordering schedule (Figure 2b). This observation indicates that the PE reordering process needs to be performed in a *conservative* manner with all the inter-PE dependences being considered, so that *pre*-reordering LtoR dependences can be eliminated with **no** extra *post*-reordering LtoR dependences being created. To attain this goal, we have developed a technique which, through the use of a *graph representation model*, directly translates inter-PE communication paths into order requirements between PEs. The ability to capture these order requirements furthermore allows the establishment of a set of mapping criteria, which can be employed to quickly check whether two communication paths have contradicting order requirements.



(a) Schedule task 16: extra timing slack imposed (b) Reorder Pb and Pd: change of dependence directions (c) Communication Paths between PEs

Fig. 2. Global impact of PE reordering on dependence directions

III. Proposed PE Remapping Scheme

In this section we discuss in detail the proposed PE remapping techniques for eliminating potential length increases and consequent performance degradations of adaptive static schedules. In the process of generating an adaptive static schedule, whenever an LtoR dependence would delay the start time of a task T_s, the scheduler should be able to quickly check whether the current PE positions can be changed so that the LtoR dependence can be eliminated without creating any post-reordering LtoR dependences. If the checking results indicate the existence of a non-LtoR mapping, the current PE binding order will be updated, which in turn enables an earlier starting time of task T_s.

Essentially, an *LtoR* dependence holds if the source and the sink tasks straddle a left to right divide and also are scheduled on non-adjacent PEs. Accordingly, an LtoR dependence can be eliminated if, by manipulating the positions of PEs, the two dependent tasks either end up straddling a right to left divide, or are scheduled on adjacent PEs. In other words, each individual communication path imposes an order requirement between the source and the sink PEs, implying that a non-LtoR mapping for the entire set of critical communication paths is possible only if paths do not impose contradictory order requirements. To capture such order requirements, we employ a *graph model* to represent the direction and the criticality of each inter-PE communication path, and furthermore develop a set of *mapping criteria* according to the connection characteristics of the graph. The last part of this section focuses on the preclusion of additional inter-PE communications in post-reconfiguration schedules through a *PE rotation* technique.

An MPSoC composed of multiple homogeneous *processing elements* (PEs) is employed as the architecture of our target system. Each PE is assumed to have a private on-chip cache and share a global off-chip memory. To simplify the analysis in this section, all the tasks under scheduling are assumed to have identical execution time and inter-processor communication overhead. While at first glance this assumption seems to be highly idealized, it actually turns out to be a representative model for the parallel sections of embedded applications. This is because a large portion of embedded applications are composed of regular data processing loops with limited or possibly even no loop-carried dependences whatsoever [9], which can be easily parallelized into a large number of tasks with high regularity.

A. Graph model of communication paths

The fundamental requirement for the PE reordering technique is to check whether a non-LtoR mapping can be established for all the critical communication paths. Accordingly, the graph representation should capture two pieces of information, the *direction* and the *criticality* of each inter-PE communication path. In this work, the entire set of inter-PE communication paths is represented as a weighted directed graph $G = (P, E, S)$. Each node $p_i \in P$ represents a PE, while an edge $e_{ij} \in E$ indicates the existence of at least one inter-PE communication from node p_i to p_j. The *criticality* of a communication path, represented as a nonnegative weight $s(e_{ij}) \in S$, is defined as the *minimum* timing slack associated with the inter-PE communications path e_{ij}.

To provide a concrete example, the graph representation of the communication paths shown in Figure 2a is presented in Figure 2c. The four inter-PE communications on three different paths are represented as three edges in the graph. The weight of the edge e_{ed} is set to 0, determined by the communication that exhibits minimum timing slack on that path, that is, the communication from Task 5 to 14.

B. PE reordering for eliminating LtoR dependences

The aforementioned graph representation enables the development of a set of mapping criteria for checking whether the entire set of critical communication paths in the schedule can be simultaneously mapped as *non-L2R* dependences. Fundamentally, such a *non-LtoR* mapping can be easily established if all the edges on the graph model exhibit a *uniform* direction, that is, if the graph turns out to be a *directed acyclic graph* (DAG). The absence of a circle in a DAG ensures that all the communication paths can be placed uniformly from right to left. A concrete example is presented in Figure 3a, wherein the PE positions are simply determined through performing a *topological sorting* [10].

If the communication graph contains a set of circles in a nested or intersecting form, the existence of a *non-LtoR* mapping for the entire set of critical communication paths cannot be easily determined. On the other hand, a fundamental property of any direct graph is that the entire graph can be decomposed into a set of disjoint *strongly connected* components[1] which, at the higher level, are connected in an *acyclic* form. This property, together with the fact that a non-LtoR mapping can always be established for a DAG, enables a decomposition of the mapping problem, formalized as follows:

Mapping decomposition: *A non-LtoR mapping can be established for an arbitrary directed graph if and only if a non-LtoR mapping can be established for each strongly connected component of the graph.*

The goal of the PE reordering technique is to develop a set of mapping criteria for strongly connected components, which can be classified as either *basic*, or *nested*, or *intersecting* loops.

A *basic loop* that exhibits a single back edge and no crossing edge turns out to be the simplest mapping case. A *non-LtoR* mapping can be easily established through performing a *clockwise ordering* on the PE positions. As shown in Figure 3b, forwarding loop edges are placed from left to right in an adjacent form, while the back edge exhibits a *right-to-left* direction, specifically, from the rightmost PE to the leftmost PE.

If the graph representation happens to be a *nested loop* with a set of crossing edges, whether a *non-LtoR* mapping can be established or not is determined by the directions of the crossing edges. Specifically, the *clockwise ordering* strategy proposed for a basic loop imposes no requirement on placing a specific loop

[1] A strongly connected component consists of a maximal set of nodes such that for every pair of nodes p_i and p_j, there exists a path from p_i to p_j and a path from p_j to p_i.

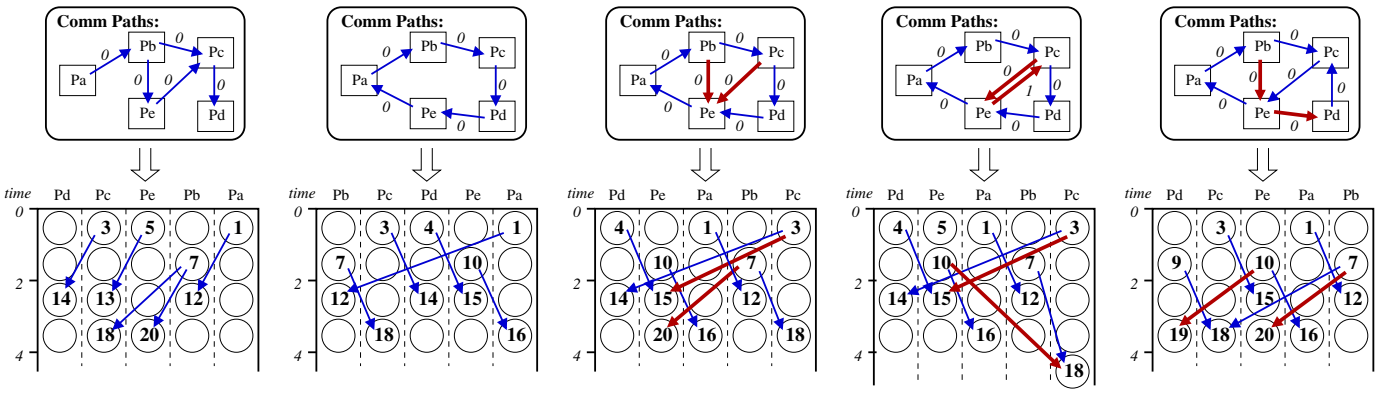


Fig. 3. Establishing non-LtoR mapping for DAG, basic loop, nested loops, and intersecting loops

edge as the back edge. Accordingly, a *non-LtoR* mapping can be established for a nested loop, if all the crossing edges exhibit a *uniform direction* in such a way that the head PEs of all the crossing edges can be rotated to the rightmost positions in the PE sequence. Moreover, a detailed examination indicates that whether two crossing edges exhibit a *uniform* direction or not can be easily checked through a depth-first search. Specifically, two crossing edges e_{ij} and e_{uv} **do not** exhibit a *uniform* direction if and only if the following two conditions both hold:

- All the paths from the edge tail p_j to the head p_i need to traverse both nodes of the other edge, namely, p_u and p_v.
- All the paths from the edge tail p_v to the head p_u need to traverse both p_i and p_j.

The impact of edge directions in mapping a nested loop is concretely illustrated in Figures 3c and 3d. In Figure 3c, the two crossing edges e_{be} and e_{ce} exhibit a *uniform direction* in that there exists a path $(p_e \rightarrow p_a \rightarrow p_b)$ that does not traverse p_c. The two head PEs P_b and P_c thus can be simultaneously placed to the right of the shared tail PE P_e. In contrast, the two crossing edges e_{ce} and e_{ec} in Figure 3d exhibit opposite directions, thus precluding their placement as right-to-left simultaneously. Therefore, in Figure 3d the edge e_{ec} has to be placed as left-to-right, thus requiring a 1-step extra timing slack being explicitly inserted on the communication path from Task 10 to Task 18.

Finally, the most complicated mapping case is the situation when a strongly connected graph is composed of multiple *intersecting loops* that partially share a set of edges in common. In order to establish *non-LtoR* mappings for all the edges, each individual loop should contain no crossing edges of opposing directions. The intersection property furthermore imposes an extra requirement of placing the shared nodes in contiguous positions that separate the disjoint parts of the two loops. Accordingly, a *non-LtoR* mapping cannot be established if this requirement contradicts an order constraint imposed by a crossing edge. The PE reordering algorithm thus checks the compatibility between the disjointness requirement and the crossing edge directions to determine whether the entire graph is mappable or not.

As a concrete example, Figure 4e shows a communication graph wherein two loops (p_a, p_b, p_c, p_e) and (p_c, p_e, p_d), share the edge e_{ce} in common. The shared nodes p_c and p_e need to be placed in the middle of the PE sequence, while the crossing edge e_{be} imposes a requirement of placing p_b to the right of p_e. A detailed examination shows the existence of a non-LtoR mapping, since the PE sequence $(p_d, p_c, p_e, p_a, p_b)$ satisfies both constraints simultaneously.

C. PE rotation for preserving communication locality

The examination presented in Section 2 confirms that the *spatial* movement among schedule bands may change localized communications to inter-processor communications, resulting in possible violations of task dependence. Fundamentally, the block & band partitions imposed on the adaptive schedule may result in **three** possible ways for two tightly scheduled dependent tasks to be separated after reconfiguration, determined by the positions of the source and the sink tasks.

If the source task is in an **R** band and the sink is in the **L** band of the same block, the sink task will be moved rightwards and downwards relative to the source after reconfiguration. This implies that a 1-step timing slack is implicitly inserted in between, thus compensating for the resultant inter-PE communications, as confirmed by the task pairs $(1, 2)$, $(3, 4)$ in Figure 4b.

In the second case, the source and the sink tasks are in either two **L** bands or two **R** bands of consecutive blocks. Illustrative examples include the task pairs $(5, 6)$ and $(10, 11)$ in Figure 4. After reconfiguration, each task pair is still located on the same PE, although a 1-step timing slack is implicitly inserted in between.

If the source task is in an **L** band and the sink is in the **R** band of the subsequent block, reconfiguration only moves the sink task rightwards relative to the source. As the tasks are separated yet no timing slack is inserted in between, insufficient time is left for the sink task to receive its inputs, as shown in the positions of task pairs $(7, 8)$ and $(4, 9)$ in Figure 4b.

A detailed examination indicates that in the final case, an inter-PE communication is created due to the left shift of the **L** band on which the source task located. This implies that all the post-reconfiguration communications of this type exhibit an identical direction such that the sink tasks (Task 8 and 9 in Figure 4b) are located one PE to the *left* of the corresponding sources (Task 7 or 4). Accordingly, if the entire subsequent BR block is rotated one PE to the right in the *post*-reconfiguration schedule, the spatial locality of the dependent tasks, such as $(7, 8)$ and $(4, 9)$ in Figure 4c, can be naturally preserved.

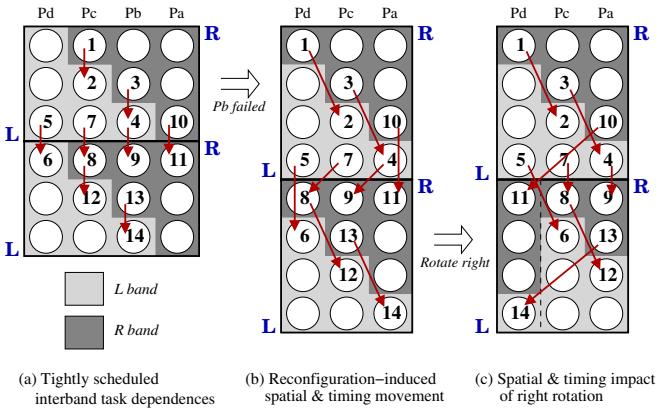


Fig. 4. Preserving spatial locality of tightly scheduled dependent tasks through PE Rotation

While this right rotation preserves the spatial locality of the *third* type of inter-task dependences, it also violates the spatial locality of the *second* type of dependences, as the sink tasks (Tasks 6 and 11 in Figure 4c), located in subsequent BR blocks, are rotated one PE to the right of the source (Tasks 5 and 10). This violation, however, requires **no** explicit timing slack to compensate for the created inter-PE communications, since an implicit timing slack has already been inserted between the source and the sink tasks in the *post*-reconfiguration schedule.

IV. Algorithmic Implementation

An important aspect of the proposed PE reordering and PE rotation techniques is that the conceptual mechanisms underpinning these techniques do not impose any requirements on the class of the underlying scheduling heuristics. However, in this section the two techniques are incorporated into one of the representative classes of scheduling heuristics, namely, *list scheduling*. A typical list scheduling algorithm usually consists of a *task prioritization* phase, where the scheduling order of each task is determined, and a *processor assignment* phase, where each task is assigned to a PE that minimizes its start time. The main difference of the various list scheduling heuristics (DCP [11], CPND [12], etc) is the determination of the scheduling order. In our implementation, the Dynamic Critical Path (DCP) algorithm [11] is selected as the baseline algorithm, while the scheduling constraints presented in [7] are employed to make the schedule reconfigurable.

The main scheduling procedure is outlined in Algorithm 1. As can be seen, the algorithm maintains an ordered list of all the ready tasks, from which the task with the highest priority, denoted as T_s, is selected for scheduling. The content of the ready list and furthermore the scheduling priorities of all the ready tasks are updated dynamically upon the scheduling of the current task, shown from Line 12 to 17 in the algorithm. Meanwhile, the procedure ScheduleT iteratively places T_s on every PE to calculate the earliest starting time. In this process, the timing constraints imposed by adaptivity may delay the starting time of T_s, if the task is placed on a PE that, according to the current PE order, is at least two PEs to the right of a predecessor.

To implement the PE reordering technique, **two** procedures have been added to the main scheduling procedure. Firstly, the communication path graph is updated (Line 11) if the scheduling of the current task T_s creates an inter-PE communication that either reduces the *weight* of an existing edge, or needs to be

Algorithm 1 Task Scheduling with PE Reordering

```
1:  Readylist = {all root tasks};
2:  while Readylist ≠ φ do
3:     T_s = t_i ∈ Readylist with highest priority;
4:     Readylist = Readylist −{T_s};
5:     {T_s.core, T_s.startTime} = ScheduleT (T_s, POrder);
6:     if extra timing slack has delayed T_s.startTime then
7:        save = ReorderP (T_s.core, POrder, CommPath);
8:        T_s.startTime = T_s.startTime − save;
9:     end if
10:    FT(T_s.core) = T_s.startTime + T_s.exeLength;
11:    UpdateComm(T_s, CommPath);
12:    AdjustPriority(Readylist);
13:    for t_i ∈ Child(T_s) do
14:       if all parents of t_i have been scheduled then
15:          Readylist = Readylist +{t_i};
16:       end if
17:    end for
18: end while
```

mapped as a new edge in the graph. More crucially, upon the detection of a potential LtoR dependence that delays the starting time of the current task T_s, the scheduler invokes the procedure ReorderP (Line 7), which checks the communication path graph to determine whether the current PE order can be adjusted to eliminate this timing slack. The procedure ReorderP first performs a depth-first search to classify the edges, and then iteratively checks all the strongly connected components whose connectivity is affected by the task just being scheduled. If the checking results indicate the existence of a non-LtoR mapping for all the communication paths in the current schedule, the current PE order will be updated, which in turn enables an earlier starting time of the task T_s (Line 8).

Once the *pre*-reconfiguration schedule has been generated using Algorithm 1, the technique outlined in [7] is employed to partition the fully parallel regions of the schedule into *BR blocks* and furthermore into *L* and *R bands*. If, as a result of this partition, two dependent tasks tightly scheduled on the same PE are separated into an **L** and a subsequent **R** band, the proposed PE rotation technique is applied. In this case, a *right rotation hint* is inserted between these two bands, so that the entire subsequent BR block would not be delayed in the *post*-reconfiguration schedule.

V. Experimental Results

In this section, the proposed PE remapping scheme is evaluated in the most common cases of resource variation, namely, single core failures. The scheduling algorithm outlined in the last section is implemented in C, while the *Dynamic Critical Path* (DCP) algorithm [11] and the BB reconfiguration scheme [7] are selected as the baseline scheduling policies. The application set under test is composed of typically parallel algorithms, such as *FFT*, *LU decomposition*, *Laplace equation solver*, and *Gaussian elimination*. DAG representations of these task graphs can be found in [11]. Meanwhile, in order to represent a large spectrum of parallelized embedded applications, we also use TGFF [13] to generate 200 random task graphs. The number of tasks is varied from 20 to 120, with the ratio of the upper to lower bound of the execution time set to 5. The frequency of inter-task dependences is controlled through varying the value of the average *out-degree* (the number of edges per node), while

TABLE I. RESULTS OF LTOR COMMUNICATION REDUCTIONS AND SCHEDULE LENGTH INCREASES

	LtoR communication		***Pre-reconfig* schedule length ↗**		***Post-reconfig* schedule length ↗**	
cores	edge/node = 1 w/ → w/o remap	edge/node = 3 w/ → w/o remap	edge/node = 1 w/ → w/o remap	edge/node = 3 w/ → w/o remap	edge/node = 1 w/ → w/o remap	edge/node = 3 w/ → w/o remap
3	6.15% → 4.40%	8.62% → 7.20%	0.27% → -0.51%	1.49% → 0.01%	11.85% → 2.58%	8.57% → 4.56%
4	8.78% → 6.59%	12.76% → 11.04%	0.63% → -0.75%	2.71% → 0.14%	10.60% → 2.68%	8.86% → 4.97%
5	10.01% → 7.53%	14.79% → 13.12%	1.12% → -0.89%	3.70% → 0.21%	11.12% → 3.56%	10.12% → 5.90%
6	10.70% → 7.81%	15.79% → 14.31%	1.88% → -0.76%	4.36% → 0.31%	10.82% → 4.45%	11.36% → 6.72%
7	11.00% → 7.79%	16.12% → 14.85%	2.41% → -0.71%	4.80% → 0.32%	11.55% → 5.42%	12.23% → 7.49%
8	11.12% → 7.53%	16.00% → 15.01%	2.70% → -0.62%	5.10% → 0.27%	11.91% → 6.46%	12.35% → 7.52%

the communication overhead is controlled through varying the average *communication-to-computation ratio (ccr)*.

The scheduling results achieved with and without PE remapping are listed in Table I, with both the amount of LtoR communications as well as the increases in *pre-* and *post-*reconfiguration[2] schedule length being reported. Each set of results are furthermore reported for two distinct configurations (*out-degree* = 1 or 3) so as to show the impact of inter-task dependence frequency. The number of cores considered in this experimental process is varied from 3 to 8.

Without PE reordering, the schedule engine reduces the amount of LtoR communications through incorporating the S-T constraint outlined in [7] into the baseline DCP algorithm. The first set of results in Table I shows that on average 10% and 14% of all the inter-PE communications need to be mapped as LtoR dependences for the low and the high out-degree cases, respectively. As a comparison, the PE reordering technique can effectively reduce the amount of LtoR communications to 7% in the low out-degree case and 13% in the high out-degree case. The reason for the relatively lower reduction in the latter case is that an application with strong inter-task dependences exhibits a large number of edges in the communication path graph, thus limiting the possible reordering choices.

The primary performance consideration in generating an adaptive static schedule is the minimization of the performance overhead on *pre*-reconfiguration schedule, as such an overhead is expended despite the possible absence of a resource variation within the system. The second set of results in Table I confirms that in the high out-degree case, the PE reordering technique can drastically reduce the performance overhead imposed by adaptivity from 3.7% to 0.2%. In the low out-degree case, the schedule engine, with PE reordering being enabled, can even slightly improve the performance of the baseline DCP schedule by 0.7%. This improvement is achieved by reassigning a task, if its starting time is not delayed thereafter, to a PE that can reuse an existing communication path. This probabilistically leads to a reduction in the total number of distinct communication paths, thus providing superior scheduling choices for subsequent tasks.

The last set of results shows that the PE rotation technique, applied on the adaptive schedules optimized through PE reordering, is able to significantly reduce the length overhead of post-reconfiguration schedules by 63% and 41% for the low and the high out-degree cases, respectively. Meanwhile, the amount of performance overhead increases as the number of PEs grows, since the post-reconfiguration schedule displays an increasing amount of under-utilization.

[2]Post-reconfiguration schedules are compared to baseline schedules of one fewer PE to evaluate relative schedule quality.

VI. CONCLUSIONS

We have presented two PE remapping techniques, capable of improving the quality of adaptive static schedules through exploiting a degree of freedom inherent in task assignment concerning the logical to physical PE binding. Using a graph representation of all the inter-PE communication paths, a *PE reordering* technique is proposed to eliminate critical inter-task dependences that are expected probabilistically to increase the *pre*-reconfiguration schedule length. Meanwhile, a *PE rotation* technique preserves the spatial locality associated with tightly scheduled dependent task pairs, thus eradicating the need for explicit timing slacks in *post*-reconfiguration schedules. The experimental results clearly confirm the efficacy of these remapping techniques in generating highly regular and predictable adaptive schedules, and furthermore drastically reducing the performance overhead both before and after reconfiguration.

REFERENCES

[1] P. Shivakumar, S. W. Keckler, D. Burger, M. Kistler, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *DSN'02*, June 2002, pp. 389–398.

[2] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The impact of technology scaling on lifetime reliability," in *DSN'04*, June 2004, pp. 177–186.

[3] G. Manimaran and C. S. R. Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis", *IEEE Trans. on Parallel and Distributed Systems*, 9(11):1137–1152, Nov 1998.

[4] A. Radulescu and A. J. van Gemund, "Low-cost task scheduling for distributed-memory machines," *IEEE Trans. on Parallel and Distributed Systems*, 13(6):648–658, June 2002.

[5] K. Albers and F. Slomka, "Efficient feasibility analysis for real-time systems with EDF scheduling," In *Proc. DATE'06*, pp. 492–497, 2005.

[6] Y. Cho, S. Yoo, K. Choi, N.-E. Zergainoh, and A. A. Jerraya, "Scheduler implementation in MPSoC design," In *Proc. ASP-DAC*, pp. 151–156, Jan. 2005.

[7] C. Yang and A. Orailoglu, "Predictable execution adaptivity through embedding dynamic reconfigurability into static MPSoC schedules," In *Proc. CODES-ISSS'07*, pp. 15–20, Oct. 2007.

[8] Z. Ma and F. Catthoor, "Scalable performance-energy trade-off exploration of embedded real-time systems on multiprocessor platforms," In *DATE'06*, pp. 1073–1078, Apr. 2006.

[9] P. Ranganathan, S. Adve, and N. P. Jouppi. "Performance of image and video processing with general-purpose processors and media ISA extensions," In *Proc. 26th ISCA*, pp. 124–135, May 1999.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Second ed. MIT Press & McGraw-Hill, 2001.

[11] Y.-K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors," *IEEE Trans. on Parallel and Distributed Systems*, 7(6):506–521, June 1996.

[12] I. Ahmad and Y.-K. Kwok, "On exploiting task duplication in parallel programming scheduling," *IEEE Trans. on Parallel and Distributed Systems*, 9(8):872–892, 1998.

[13] R. P. Dick, D. L. Rhodes, and W. Wolf. "TGFF: Task graphs for free," In *Proc. Workshop Hardware/Software Codesign*, pp. 97–101, 1998.