# Predictable Execution Adaptivity through Embedding Dynamic Reconfigurability into Static MPSoC Schedules*

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

Advances in semiconductor technologies have placed MPSoCs center stage as a standard architecture for embedded applications of ever increasing complexity. Because of real-time constraints, applications are usually statically parallelized and scheduled onto the target MPSoC so as to obtain predictable worst-case performance. However, both technology scaling trends and resource competition among applications have led to variations in the availability of resources during execution, thus questioning the dynamic viability of the initial static schedules. To eliminate this problem, in this paper we propose to statically generate a compact schedule with predictable response to various resource availability constraints. Such schedules are generated by adhering to a novel band structure, capable of spawning dynamically a regular reassignment upon resource variations. Through incorporating several soft constraints into the original scheduling heuristic, the proposed technique can furthermore exploit the inherent timing slack between dependent tasks, thus retaining the spatial and temporal locality of the original schedule. The efficacy of the proposed technique is confirmed by incorporating it into a widely adopted list scheduling heuristic, and experimentally verifying it in the context of single processor deallocations.

**Categories and Subject Descriptors:** C.4 [Performance of Systems]: *–Reliability, availability, and serviceability*

**General Terms:** Reliability

**Keywords:** Adaptive execution, reconfiguration, multiprocessor task scheduling

## 1. INTRODUCTION

The Multiprocessor System-on-Chip (MPSoC) [1] is rapidly becoming a standard organization for embedded systems, as advances in VLSI fabrication technologies continuously augment the tremendous amount of extant computational power. The increasing computational power in turn engenders integration and most importantly the simultaneous execution of higher number of applications on an MPSoC; illustrative examples include the various video processing and recoding applications executed on the Viper2 set-top box [2]. Meanwhile, the execution environment is becoming more diverse, dynamic and unpredictable in that the amount of computational resources available to an application may vary due to either competing demands from other applications, or the degradation of hardware reliability. Technology scaling has not only accentuated the probability of device failures during execution [3], but also has led to higher temperatures and increased thermal gradients [4], both of which may cause an over-heated processor to be temporarily unavailable during execution.

The increasing possibility of resource variations requires a reconsideration of the critical issue of scheduling the tasks of an application to the processing elements (PEs) of an MPSoC. Traditionally task scheduling can be performed either *statically* during compilation, or *dynamically* at run time. In general, a detailed comparison shows that three fundamental reasons account for the attractiveness of static scheduling for embedded MPSoCs. Firstly, static scheduling is more cost effective, as dynamic scheduling imposes indispensable overhead in collecting workload information and scheduling online. Secondly, the limited application set of a typical embedded system allows the extraction of sophisticated application information and the use of aggressive static optimization techniques, both highly desirable for static schedulers in their quest for global workload balance. Thirdly yet most importantly, most embedded applications need to impose crucial constraints on worst case performance, a requirement frequently frustrated by dynamic schedulers since they can hardly deliver a predictable worst-case schedule as the scheduling decisions are highly impacted by run-time events, such as network or memory contentions.

A number of static scheduling heuristics have been developed for parallel systems [5], with the applicability of these heuristics to MPSoCs also being examined recently [6, 7]. Nonetheless, the schedules generated by these heuristics are typically confined to the case of a fixed number of PEs, thus limiting their usefulness in a dynamic execution environment, as a resource variation, especially a *PE deallocation*, would typically demand a rearrangement of the complete original schedule.

Because of the increasing demand for adaptivity in future MPSoCs, *dynamic reconfigurability* should be embedded into static schedules to either withstand a resource reduction or make use of extra allocated PEs. Traditional scheduling techniques either backup each task [8], or keep additional spare processors [9] for resource reduction tolerance. However, both strategies impose significant costs in terms of the originally attainable performance. As resource deallocations are relatively sparse, the performance of the original schedule is paramount, thus requiring particular attention to the costs imposed on the original schedule. The reconfiguration process furthermore should be very fast and highly predictable in order to minimize the extra reconfiguration overhead and to meet the real-time constraints associated with most embedded applications. Unfortunately, none of the previous run-time reconfiguration techniques are able to meet these goals simultaneously. For example, a scheduling algorithm proposed in [10] achieves reconfiguration through rescheduling on-line and re-executing tasks upon a processor failure, thus not only impos-

*This work is supported in part by a grant from Cal-$(IT)^2$.


*CODES+ISSS'07*, September 30–October 3, 2007, Salzburg, Austria.

ing a considerable reconfiguration overhead but also causing unpredictability in worst case performance.

Pure run-time reconfiguration techniques can only make decisions that are locally-optimal, possibly causing unpredictable impact on the overall performance. The simultaneous achievement of the aforementioned set of goals necessitates sophisticated planning to be performed statically, motivating the introduction of a predictable reconfiguration scheme. In general, by making sophisticated planning during static scheduling, the proposed ***Block & Band (BB) reconfiguration*** scheme is able to generate a compact schedule with regular and predictable response to various resource availability constraints. A regular reassignment capability is accomplished through performing a group transfer of a set of tasks upon a resource variation during execution, while within each band the relative position of each task is retained intact. More crucially, through the introduction of a set of soft constraints that exploit the flexibility inherent in schedule generation, the proposed light-weight, highly regular and predictable reconfiguration scheme only imposes a negligible impact on the performance of the original schedule.

The rest of the paper is organized as follows. Section 2 describes in detail the proposed reconfiguration scheme, which is incorporated into a representative scheduling algorithm in section 3. Section 4 experimentally verifies the efficacy of the proposed reconfiguration technique in the context of single processor deallocations, while section 5 offers a set of brief conclusions.

## 2. BB RECONFIGURATION SCHEME

This section examines in detail the proposed ***Block & Band (BB) reconfiguration*** scheme with particular emphasis on its conceptual mechanism. We first introduce the main idea through the exposition of the *canonical case*, wherein a program consists of multiple tasks with identical execution time and negligible communication latency. Subsequently, we analyze in detail two critical issues: the impact of reconfiguration on inter-task dependences, as well as the application of the proposed regular reconfiguration technique to arbitrary programs with diverse task execution time and sizable communication latency.

As resource variations, due to either resource competition or unpredictable device failure or thermal stress, are relatively sparse, the single PE deallocations are the most common type of resource variation to be encountered during execution, thus making their handling a crucial aspect of the effectiveness of the reconfiguration process. We select one of the most representative MPSoCs, a chip multiprocessor that consists of multiple identical *processing elements* (PEs), as the architecture of our target system. Each PE is assumed to have private instruction and data caches, while the data sharing and communication among PEs is achieved through the use of a global memory shared by all PEs.

### 2.1 Canonical Case

The main idea of BB reconfiguration is to incorporate dynamic reconfigurability directly into the static schedules. This is achieved through imposing a band structure during static scheduling, together with a group transfer of the whole band upon a resource variation during execution.

The conceptual mechanism of the proposed static partition and band-level task transfer can be clarified possibly by considering a basic example presented in figure 1, wherein 24 tasks of identical execution time are statically scheduled onto an MPSoC consisting of 4 PEs. Figure 1*a* presents the initial scheduling results, with each rectangle labeled with a number denoting a task and each column representing one PE of the MPSoC. The band-level partition of the initial schedule is

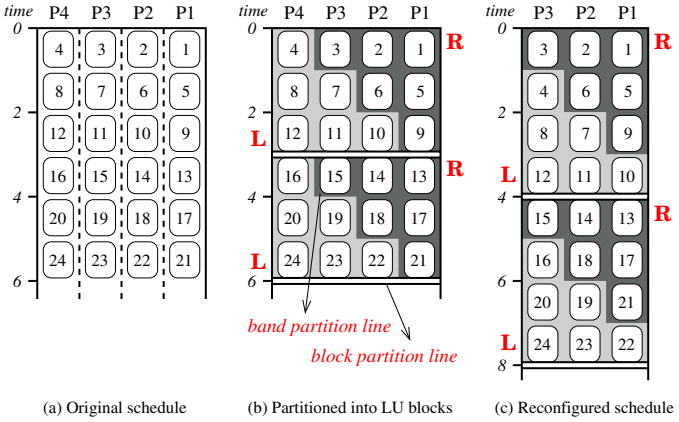


**Figure 1: BB reconfiguration for the canonical case**

presented in figure 1b. As can be seen, the initial schedule is partitioned into two ***Basic Reconfiguration (BR) blocks***, which constitute the minimal reconfiguration units. Each BR block is furthermore divided into two bands, a ***Left (L) band*** and a ***Right (R) band***. To form these two types of partitions, two distinct types of lines are imposed on the original schedule:

- **Block partition line:** the straight horizontal line between two sequential BR blocks.
- **Band partition line:** the staircase line between the L and the R band within the same BR block.

A noteworthy aspect of the proposed scheme is that the outlined shape of the L and the R band enables a regular reassignment capability upon a resource variation. By comparing figures 1*b* and 1*c*, it can be clearly observed that in both the initial and the reconfigured schedule, *BR blocks*, the minimal reconfiguration units, are executed **sequentially** in the same order. However, in each *BR block* the whole **L** band is shifted in a regular manner relative to the **R** band, that is, one timing step down and one PE to the right. This allows all the tasks within each *BR block* to be completed with one less PE, albeit with an additional timing step after reconfiguration. Crucially, both schedules presented in figures 1*b* and 1*c* are able to make **full utilization** of the available hardware resource. This significant benefit is directly derived from the size of each BR block. As can be seen in figure 1*b*, since each BR block contains $4 * (4 - 1) = 12$ tasks, each BR block can be completed either in 3 timing steps using 4 PEs as in the original schedule, or in 4 timing steps using 3 PEs as in the reconfigured schedule. More generally, the impact of block size on resource utilization can be formally specified as follows:

> ***Block Size constraint:*** *A full utilization of PEs both before and after reconfiguration requires each* ***BR block*** *to contain $n * (n - m)$ tasks in order to tolerate a deallocation of $m$ out of $n$ PEs.*

Another important benefit of the proposed technique is its highly regular task reassignment process, achievable **independent** of the PE being removed. This property can be easily observed on an illustrative example presented in figure 2*b*, which uses arrows to indicate the required shifting directions of all tasks in order to tolerate the deallocation of $P2$. As can be seen, the reassignment process displays high regularity in that not only all the tasks within a single band share an identical timing offset after reconfiguration (*temporal* reassignment), but also task transfers are only performed among adjacent PEs (*spatial* reassignment).

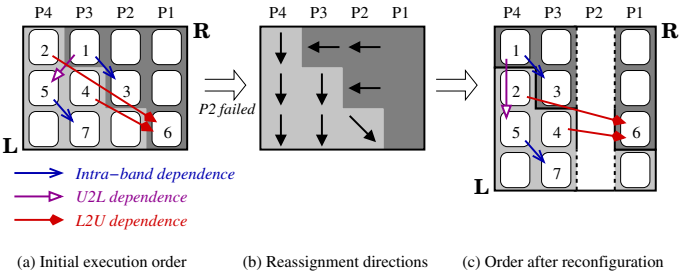


**Figure 2: Reassignment directions and timing behavior of inter-task dependences**

## 2.2 Inter-task Dependence Constraints

One fundamental requirement for a reconfiguration technique is to preserve the partial ordering imposed by inter-task data and control dependences. In the proposed *BB reconfiguration* scheme, because the initial schedule is partitioned into BR blocks and further into multiple bands, the dependences among tasks can be naturally classified into four categories: *inter-block dependences, intra-band dependences, R2L dependences* and *L2R dependences*. In this subsection we will analyze in detail the timing behavior of each category.

Because BR blocks are executed sequentially in the same order both before and after reconfiguration, it is impossible for any violation of *inter-block dependences* to take place. The same property can be observed for *intra-band dependences* since dependent tasks that lie within the same band retain their original relative position after reconfiguration. This property can be clearly observed in figure 2*c* from the relative timing relationship between tasks 1 and 3 and between tasks 5 and 7.

Because the L bands are shifted downwards relative to the R bands, the timing slack associated with each *R2L dependence* will always **increase** after reconfiguration, while the timing slack associated with each *L2R dependence* will always **decrease**. Accordingly, *R2L dependences* can also be naturally preserved, as can be seen between tasks 1 and 5 in figure 2*c*. However, for *L2R dependences*, a violation **may** occur if the original timing slack proves insufficient. As shown in figure 2*a*, while *Task 5* is scheduled to be executed after *Task 3* initially, these two tasks are scheduled at exactly the same timing slot after reconfiguration in figure 2*c*.

The analysis above confirms one of the fundamental advantages of the proposed scheme in that most of the inter-task dependences can be preserved naturally after reconfiguration. The only type of inter-task dependences that **may** be violated after reconfiguration is the *L2R dependences*. These potential violations are quite rare, as an *L2R dependence* only occurs if two dependent tasks not only straddle the left to right direction divide between the bands, but also are scheduled on neither the same nor adjacent PEs. Furthermore, a potential violation can be easily precluded by imposing a certain amount of timing slack between the two tasks in an *L2R dependence*, such as tasks 2 and 6 in figure 2*a*. This property can be formalized as the following *spatial-temporal (S-T) constraint*:

> ***S-T constraint:*** *A dependence between a predecessor task on band $i$ to a task on band $i+k$ (left to right) needs to have an intervening slack of $k$ timing steps in the initial schedule to preserve the correct execution order after reconfiguration.*

## 2.3 Applied to General Task Graphs

The last two subsections have clearly shown the high regularity of the proposed BB reconfiguration scheme for the canonical case. While at first sight the canonical case seems to

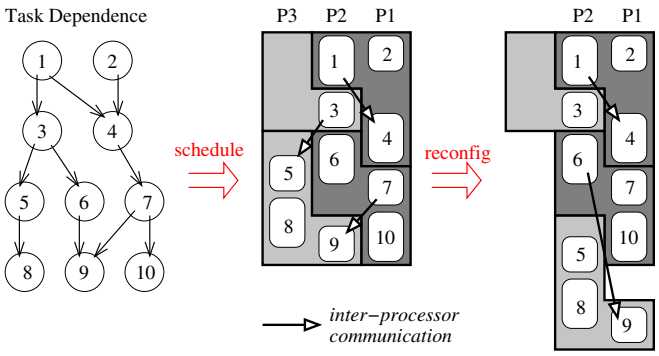


**Figure 3: New characteristics encountered in reconfiguring a complicated schedule**

be highly idealized, it actually turns out to be a representative model for the parallel sections of embedded applications with significant data-level parallelism. This is because a large portion of embedded applications are composed of regular data processing loops with limited or even no loop-carried dependences [11], which can be easily parallelized into a large number of tasks with high regularity, as assumed in the canonical case. However, to further examine the effectiveness of the proposed reconfiguration scheme, in this subsection we extend its application to arbitrary programs with diverse task execution time and non-zero communication latency.

In general, the application of the regular BB reconfiguration scheme to a general task graph results in several new characteristics that are not observed in the canonical case. These characteristics furthermore impact the BB reconfiguration scheme in **two** distinct directions by both providing opportunities in improving the reconfigured schedule, and creating challenges in compensating extra reconfiguration overhead.

**Idle PE cycles in the initial schedule:** As inter-task dependences and communication overhead may strictly constrain the earliest starting time of a task, the initial schedule may display significant underutilization in certain portions. This can be clearly observed in figure 3, in which the PE $P3$ is left idle across the entire timing period of the first *BR block*.

An under-utilized portion of the initial schedule has at least one PE with no task to execute, implying no need of adjusting that part of the initial schedule after reconfiguration. As presented in figure 3, the exploitation of this property allows a sizable reduction in the length of the reconfigured schedule. More specifically, we propose to extend the original *BR block* to contain a **head** or a **tail** region, as shown in figure 4. By mapping the fully parallel portions of the initial schedule into the **body** regions and the under-utilized portions into the **head** or **tail** regions, the schedule lengths of the head and tail regions remain constant after reconfiguration, while only the bands within the body regions need to be shifted.

**Irregularity of partition lines:** Because of diverse task execution time, the heights of the steps on a *band partition line* are not necessarily identical, and the original horizontal *block partition lines* are not necessarily straight. As can be clearly observed from figure 3, both types of irregularity may degrade performance by creating extra timing holes in the reconfigured schedule.

A detailed examination evinces the reconfiguration penalty in terms of schedule length associated with each *BR block* to be constrained by the **maximum height** of the steps on the *band partition line*. This property can be illustrated more clearly by considering the case presented in figure 4. Assume a body region is originally scheduled to commence at time $a$ and end at time $d$, and the positions of the intermediate

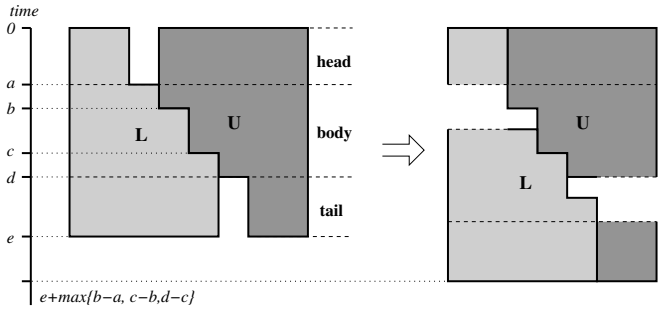


**Figure 4: Minimizing performance degradation in reconfiguration**

two steps on the band partition line are at time $b$ and time $c$, respectively. After reconfiguration, the schedule length of the body region equals

$$\max\{(b-a)+(d-a),(c-a)+(d-b),(d-a)+(d-c)\} = (d-a)+\max\{b-a,c-b,d-c\} \quad (1)$$

Because of this property, during the static scheduling process the step height on each band partition line should be balanced through an adjustment of the task starting times. However, this balancing should be performed without increasing the length of the initial schedule, thus requiring the static scheduler to exploit the timing slacks of non-critical tasks inherent within the initial schedule.

**Variations in inter-processor communication:** Because of the relative spatial movement of the two bands within each *BR block*, two dependent tasks originally scheduled on the same PE may be separated onto two PEs after reconfiguration and vice versa. As a result, inter-processor communications displayed in the initial schedule may not appear after reconfiguration (such as communications between tasks 3 and 5 and between tasks 7 and 9 in figure 3), while new inter-processor communications may be created in the reconfigured schedule (such as communications between tasks 6 and 9). While the first case can be exploited to further improve the performance of the reconfigured schedule, it is essential to provide mechanisms for compensating for the extra overhead caused by the second case.

A detailed examination shows that a new inter-processor communication can occur either in an *R2L dependence* (between tasks 6 and 9 in figure 3) or an *L2R dependence* across two BR blocks. However, these two cases display diverse timing characteristics. In the latter case the relative timing positions of the L band and the following R band remain constant after reconfiguration, while for an *R2L dependence* an additional timing slack is implicitly inserted in the reconfigured schedule, since each L band is shifted downwards relative to the R band. Consequently, the additional timing slack can be utilized by the static task scheduler to compensate for the overhead of a new inter-processor communication. More specifically, during scheduling if two dependent tasks on the same PE cannot be grouped into the same band, the next preference is to group them into the same *BR block* and to schedule the predecessor task on the **R** band.

## 2.4 Impact Analysis

### 2.4.1 Impact of variations in task execution time

It can be argued that precise delineation of reconfigurable schedules in the face of variations in *task execution time* is a challenging task. Nonetheless, careful consideration shows that the effect of variations on task execution time can be largely minimized. The three fundamental sources of variations, unpredictable architectural events, competing resource utilizations, and input dependent computation variation, can be handled by compilation-based techniques, dedicated resource allocation, and appropriate task parallelization. Unpredictable architectural events, such as branch misprediction or cache misses, result in variations of short duration, typically of only tens of cycles. This small variation can be further reduced through *predicated execution* [12] or *data prefetching* [13]. Resource competition from other applications, which may cause a task in execution to be suspended, can be completely eliminated through adopting a *space sharing* [14] strategy for computational resources. Finally, as parallel loops are typically partitioned into a fixed number of tasks, the number of iterations in each task may be input dependent. This source of variation can be minimized through fixing task granularity. Although this loop parallelization strategy creates an input-dependent number of tasks, the impact on static schedules is still predictable, as the high similarity of these tasks will lead to repetitive portions within the schedule.

### 2.4.2 Impact of underlying memory organization

Because the proposed *BB reconfiguration* scheme is performed at the processor level, its impact on the behavior of the architectural components of each PE, such as the private cache, is negligible. However, as reconfiguration requires tasks to be shifted among PEs, the memory organization of the target MPSoC significantly affects the reconfiguration overhead. Since it is assumed that in the target MPSoC a global memory is shared by all PEs, task reassignment can be achieved through remapping the associated address space, thus requiring no real movement of code or data. Moreover, as the proposed reconfiguration scheme only requires the shifting of tasks between adjacent PEs in the case of single PE deallocation, the elimination of extra code/data transfer only requires local storage to be shared between every two adjacent PEs, thus being accomplished even in an MPSoC with a distributed memory architecture.

# 3. ALGORITHMIC IMPLEMENTATION

In the last section we have presented the conceptual mechanisms underpinning the proposed *BB reconfiguration scheme*. It should be noted that the effectiveness of the proposed *BB reconfiguration* technique does not impose any requirements on the class of the underlying static scheduling algorithm. In this section, we present the implementation of the proposed *BB reconfiguration scheme* through applying the reconfiguration constraints to one of the representative classes of scheduling heuristics, namely, *list scheduling*.

Given a parallel program represented as a weighted *directed acyclic graph* (DAG), the scheduling problem can be formalized as the association of a start time and a processor with each node of the DAG. A typical list scheduling algorithm usually consists of a ***task prioritization*** phase and a ***processor assignment*** phase, and the main difference of the various list scheduling heuristics is the determination of the scheduling order (e.g., DCP[15], CPND[16], etc). In our implementation, the *Critical Path Node Dominate* (CPND) algorithm [16] is selected as the baseline algorithm.

In general, the implementation of the proposed reconfiguration scheme requires four additional functions to be added into the static task scheduling process:

- Precluding potential violations of *L2R dependences*.
- Exploiting under-utilized portions of the initial schedule.
- Balancing step heights on *band partition lines*.
- Compensating extra inter-PE communication overhead.

Among these four functions, the first one is an essential constraint that needs to be obeyed by the initial schedule,

while the other three are optimization constraints for reducing the length of the reconfigured schedule.

## 3.1 Initial Schedule Generation

In our algorithmic implementation, the *task prioritization* phase of the baseline algorithm is retained intact, while the *processor assignment* phase is slightly modified to eliminate potential violations of *L2R dependences* after reconfiguration. As can be observed from figure 2, the existence of an *L2R dependence* requires two dependent tasks not only to straddle the left to right divide, but also to be scheduled on non-adjacent PEs. In other words, a predecessor task $v_j$ scheduled on a PE, denoted as $proc(v_j)$, results in a *L2R dependence* **only if** the descendent task $v_i$ is scheduled on a PE $P_k$ which is at least two PEs to the right of $v_j$, that is, $P_k \leq proc(v_j) - 2$.

The S-T constraint presented in Section 2.2 indicates that the prevention of a potential dependence violation between tasks $v_j$ and $v_i$ only requires an extra timing slack $T_{ex}$ to be imposed in the case of $P_k \leq proc(v_j) - 2$. According to Equation (1), ideally the value of $T_{ex}$ should equal the maximum height of the steps on the *band partition line*. However, in this phase the generation of the initial schedule has not been completed yet, implying that neither the *BR blocks* nor the *L/R bands* have been formed. As a result, the existence of an *L2R dependence* needs to be estimated. In our implementation $T_{ex}$ is estimated based on the assumption that the task execution time forms a *normal distribution* with mean $T_\mu$ and variance $T_\sigma^2$: we set $T_{ex} = T_\mu + T_\sigma$ to approximate the relative delay of the L band after reconfiguration.

In sum, the modified *processor assignment* phase still schedules each task in the pre-computed priority order to minimize its start time, with the scheduling of a task $v_i$ formalized as follows:

$$ST_{min}(v_i) = \min_k\{ST(v_i, P_k)\} \tag{2a}$$

$$ST(v_i, P_k) \geq \max_{v_j \in pred(v_i)}\{FT(v_j) + c(e_{ji}) + T_{ex}(P_k, proc(v_j))\} \tag{2b}$$

$$T_{ex}(P_k, P_j) = \begin{cases} T_\mu + T_\sigma & \text{if } k \leq j-2, \\ 0 & \text{otherwise.} \end{cases} \tag{2c}$$

As can be seen in Equation (2*a*), the earliest start time, $ST_{min}(v_i)$, is calculated by in turn placing the task $v_i$ on every PE $P_k$. Equation (2*b*) shows that the start time of $v_i$ on $P_k$ is constrained by the last incoming communication, with $c(e_{ji})$ denoting the communication overhead between the tasks $v_j$ and $v_i$, and $FT(v_j)$ the finish time of task $v_j$. Furthermore, it is clear that the **only modification** to the baseline algorithm is presented in Equation (2*c*), which adds an extra timing slack of $T_\mu + T_\sigma$ to the calculation of the start time $ST(v_i, P_k)$ if $v_i$ is planned to be scheduled at least two processors to the right of $v_j$. However, most of the time this increase in data ready time will not delay the start time of task $v_i$. This is because Equation (2*a*) selects the minimal start time of $v_i$ among all the PEs, resulting in $v_i$ being scheduled onto another PE $P_l$ that satisfies the condition of $P_l > proc(v_j) - 2$. The impact of this extra timing slack can be further reduced by scheduling the first task of the critical path on the rightmost PE, thus minimizing the occurrence possibility of *L2R dependences* on the critical path.

## 3.2 Reconfiguration Incorporation

After generating the initial schedule, two additional phases are appended to the baseline algorithm to incorporate reconfigurability. One is a ***schedule partition*** phase, employing the remaining three constraints to guide the partition of the initial schedule into *BR blocks* and *L/R bands*. The other is a ***reassignment optimization*** phase, wherein the semantic correctness of the reconfigured schedule is verified.

The *schedule partition* phase forms *BR blocks* and *L/R bands* in the following **three steps**.

1. Identify the fully parallel regions of the initial schedule.
2. Determine the band and block *partition lines*.
3. Balance the step height on each *band partition line*.

The positions of the fully parallel regions in the initial schedule determine the total number of BR blocks as well as the position of the *body* region of each BR block, thus strongly impacting the length of the reconfigured schedule. An extreme case that may occur is that the initial schedule may have no fully parallel regions, implying that no change of initial schedule is needed in the case of a single processor deallocation.

If a search of the initial schedule identifies at least one fully parallel region, the *band and block partition lines* of each BR block will subsequently be determined. The goal of this step is to minimize the occurrence possibility of extra inter-processor communications. Consequently, each pair of dependent tasks tightly scheduled on the same PE is placed into the same band or into two consecutive R (or L) bands as much as possible.

Once the band partition lines have been determined, the next step is to balance the step height. To retain the length of the initial schedule intact, only the tasks whose latest start time, $ST_{max}$, exceeds the current start time $ST$ can be delayed. The latest start time, $ST_{max}$, of a task $v_j$ is calculated as follows:

$$ST_{max}(v_j) = w(v_j) + \max_{v_i \in succ(v_j)}\{c(e_{ji}) + ST_{max}(v_i)\} \tag{3}$$

wherein $w(v_j)$ and $c(e_{ji})$ denote the execution time of task $v_j$ and the communication cost between tasks $v_j$ and $v_i$, respectively.

The *schedule partition* phase determines the shape of each band, enabling the timing offset needed by each band after reconfiguration to be calculated. The **primary goal** of the *reassignment optimization* phase is to minimize the timing offset needed by each band after reconfiguration. The only requirement is to ensure that each task in the reconfigured schedule has all the necessary incoming communications ready on time. More specifically, the new initiation time of each task is verified in the same scheduling order generated in the *task prioritization* phase. The start time of a task $v_i$ after reconfiguration, denoted as $ST_{new}(v_i)$, should satisfy the following inequality:

$$ST_{new}(v_i) \geq \max_j\{FT(v_j, proc_{new}(v_j)) + c(e_{ji})\} \tag{4}$$

with $c(e_{ji})$ denoting the communication overhead between the tasks $v_j$ and $v_i$, and $FT(v_j, proc_{new}(v_j))$ the finish time of node $v_j$, which is reassigned to PE $proc_{new}(v_j)$ after reconfiguration. If an unnecessary timing hole is detected and the whole band can be commenced earlier without causing any resource conflict, a reduction in the timing offset of the specific band and all the subsequent bands will be performed.

# 4. EXPERIMENTAL RESULTS

To evaluate the proposed reconfiguration scheme, both the base-line CPND algorithm [16] and our own modifications presented in the last section are implemented in C++. The application set under test is composed of typically parallel algorithms, such as *LU decomposition*, *Laplace equation solver*, and *Gaussian elimination*. The DAG representations of these task graphs can be found in [15]. We furthermore use TGFF [17] to generate 100 random task graphs in order to represent a large spectrum of parallelized embedded applications. For these task graphs, the number of tasks is varied from 20 to 100, with the ratio of the upper to lower bound of the execution

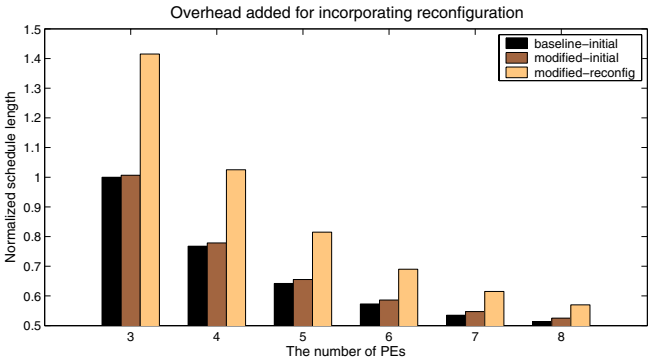


**Figure 5: Impact of reconfiguration on the initial schedule**

time distribution set to 10. The frequency of inter-task dependences is controlled through varying the value of the average *out-degree* (the average number of edges per node), while the communication overhead is controlled through varying the average *communication-to-computation ratio (ccr)*. In addition, in the process of generating the initial schedule, the number of PEs considered in our experiments is varied from 3 to 8.

The simulation results are presented in figure 5, normalized to the schedule length of the baseline algorithm generated for 3 PEs. As the primary performance consideration of the proposed reconfiguration scheme is to retain the performance of the initial schedule intact, we first evaluate the effectiveness of our algorithmic implementation through comparing the length of the initial schedule generated by the baseline (the left one in each group of bars in figure 5) and the modified algorithm (the middle one in each group of bars in figure 5). As can be seen, for all the cases the overhead of schedule length introduced by the proposed reconfiguration scheme is negligible, only within 4%. A more detailed examination shows that this overhead wanes slightly as the number of PEs increases. This is because the prevention of *L2U dependences* limits the choices of PEs for the modified algorithm, while the baseline algorithm can make more use of an additional PE if the total number of PEs is small. As the number of PEs increases, however, the baseline algorithm reaches the maximum amount of parallelism, while the modified algorithm enjoys an increased number of choices in spatial assignment.

The effectiveness of our algorithmic implementation can also be seen by comparing the length of the initial schedule (the middle one in each group of bars in figure 5) and the reconfigured schedule (the right one in each group of bars in figure 5) generated by the modified algorithm. Here it needs to be noted that while the initial schedule is generated for $n$ ($3 \leq n \leq 8$) PEs, the reconfigured schedule presented in the same column is generated by reconfiguring the initial schedule for $n-1$, i.e. one less, PEs. As can be seen, the reconfiguration overhead imposed on the initial schedule ranges from 12% to 41%. Nonetheless, our reconfiguration scheme is still quite effective, because the reconfigured schedule for $n$ ($4 \leq n \leq 8$) PEs is of similar length as the initial schedule for $n-1$ PEs. Moreover, the reconfiguration overhead decreases significantly as the number of PEs increases, since using more PEs results in a larger part of under-utilized portions in the initial schedule that do not need to be reconfigured in the case of a single PE deallocation.

# 5. CONCLUSIONS

We have presented an effective technique that allows reconfigurability to be incorporated into static schedules in order to withstand processor deallocations at runtime due to either resource competition or unpredictable device failure or thermal stress. Through statically partitioning the initial schedule into multiple bands, a regular reassignment capability is embedded into the static schedule to perform a group transfer of a set of dependent tasks to a new PE upon execution-driven resource variation. Furthermore, through the incorporation of several soft constraints into the scheduling process, the proposed technique can exploit the inherent timing slack between dependent tasks, thus minimizing the performance overhead introduced by reconfiguration. This advantage has been clearly confirmed by the experimental results, which show that only less than 4% of overhead is imposed on the length of the initial schedule to accomplish the highly regular and predictable reconfiguration scheme for the toleration of single processor deallocations.

# 6. REFERENCES


[1] W. Wolf. The future of multiprocessor systems-on-chips. In *Proc. 41st DAC*, pages 681–685, 2004.

[2] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital TV systems. *IEEE Design and Test of Computers*, 18(5):21–31, Sept. 2001.

[3] M. A. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *IEEE Micro*, 23(6):76–83, Nov. 2003.

[4] International Technology Roadmap for Semiconductors (ITRS), 2005 Edition. Process integration, devices, and structures. http://www.itrs.net/Links/2005ITRS/PIDS2005.pdf

[5] 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.

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

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

[8] C. Gond, R. Melhem, and R. Gupta. Loop transformations for fault detection in regular loops on massively parallel systems. *IEEE Trans. on Parallel and Distributed Systems*, 7(12):1238–1249, Dec. 1996.

[9] M. Chean and J. Fortes. The full-use-of-suitable-spares (FUSS) approach to hardware reconfiguration for fault-tolerant processor arrays. *IEEE Trans. on Computers*, 39(4):564–571, April 1990.

[10] S. Chabridon and E. Gelenbe. Failure detection algorithms for a reliable execution of parallel programs. In *14th Int'l Symp. on Reliable Distributed Systems*, pages 229–238, 1995.

[11] 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*, pages 124–135, May 1999.

[12] C. Panis, U. Hirnschrott, A. Krall, G. Laure, W. Lazian, and J. Nurmi. FSEL – selective predicated execution for a configurable DSP core. In *Proc. ISVLSI'04*, pages 317–320, Feb. 2004.

[13] D. F. Zucker, R. B. Lee, and M. J. Flynn. Hardware and software cache prefetching techniques for MPEG benchmarks. *IEEE Trans on Circuits and Systems for Video Technology*, 10(5):782–796, Aug. 2000.

[14] K. K. Yue and D. J. Lilja. An efffective processor allocation strategy for multiprogrammed shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(12):1246–1258, Dec. 1997.

[15] 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.

[16] Y.-K. Kwok, I. Ahmad, and J. Gu. Fast: A low-complexity algorithm for efficient scheduling of DAGs on parallel processors. In *Proc. Int'l Conf. Parallel Processing (ICCP)*, pages 155–157, Aug. 1996.

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