Mapping Data-Parallel Tasks Onto Partially Reconfigurable Hybrid Processor Architectures

Krishna N. Vikram, *Member, IEEE*, and Vinita Vasudevan, *Member, IEEE*

***Abstract*—Reconfigurable hybrid processor systems provide a flexible platform for mapping data-parallel applications, while providing considerable speedup over software implementations. However, the overhead for reconfiguration presents a significant deterrent in mapping applications onto reconfigurable hardware. Partial runtime reconfiguration is one approach to reduce the reconfiguration overhead. In this paper, we present a methodology to map data-parallel tasks onto hardware that supports partial reconfiguration. The aim is to obtain the maximum possible speedup, for a given reconfiguration time, bus speed, and computation speed. The proposed approach involves using multiple, identical but independent processing units in the reconfigurable hardware. Under nonzero reconfiguration overhead, we show that there exists an upper limit on the number of processing units that can be employed beyond which further reduction in execution time is not possible. We obtain solutions for the minimum processing time, the corresponding load distribution, and schedule for data transfer. To demonstrate the applicability of the analysis, we present the following: 1) various plots showing the variation of processing time with different parameters; 2) hardware simulations for two examples, viz., 1-D discrete wavelet transform and finite impulse response filter, targeted to Xilinx field-programmable gate arrays (FPGAs); and 3) experimental results for a hardware prototype implemented on a FPGA board.**

***Index Terms*—Data-parallel tasks, divisible load theory, dynamically reconfigurable logic (DRL), hybrid processor architectures, partial reconfiguration.**

I. Introduction

RECONFIGURABLE systems use adaptive hardware to address the varying needs of different applications [1]. The reconfigurable logic, generally a field-programmable gate array (FPGA), augments the functionality of a general-purpose processor (GPP). The current trend is to incorporate the reconfigurable logic fabric (RF) on the same die as the GPP, to alleviate the problem of communication overhead between the GPP and the RF [2]. Despite the reduced communication overhead in such *hybrid processor* architectures, one of the major roadblocks to reconfigurable computing being adopted in the mainstream has been the large delay associated with hardware reconfiguration. Large reconfiguration times mandate the use of applications with large computation times to amortize the reconfiguration overhead.

In the literature, various techniques have been described for reducing the reconfiguration delay overhead. These include configuration compression, configuration caching and prefetching, configuration relocation and defragmentation, utilizing multiple contexts, and using partial runtime reconfiguration (RTR) [2]. Partial RTR (PRTR) allows for changing the functionality of a portion of the RF area, while the remaining area stays active in computation. PRTR has received favourable attention in commercially available hardware implementations [3], [4].

Partially reconfigurable hardware provides the framework to compensate for large reconfiguration times. However, the methodology for using this feature to reduce the execution time of an application remains an open and active area of research. Recent research comprises of static as well as dynamic scheduling algorithms proposed for minimizing the reconfiguration overhead in partially reconfigurable hardware [5]–[9]. These techniques operate at the task/subtask level and can be used for any application.

Among the various applications, signal/image processing, multimedia, and vision applications remain the most attractive for implementation on reconfigurable systems [6], [7], [10]. These target applications comprise of tasks that operate on large amounts of data and possess a high degree of data parallelism [11]. For such tasks, it is possible to have multiple independent processing units (PUs) operating on different parts of the input data. Since the PUs operate independently, each PU can start functioning as soon as the RF area allocated to it is configured. This offers the potential to further minimize the RF reconfiguration overhead and obtain a greater degree of acceleration [12], [13].

However, since the RF is part of a hybrid processor system, the memory bandwidth available to the RF is usually limited. RF access to memory generally occurs over a common bus that connects the RF to the memory system and all PUs utilize this bus for data access. Moreover, for a partially reconfigurable system with a single configuration port, the PUs have to be configured sequentially. Reconfiguration delay and limited data bandwidth are, therefore, two main architectural constraints present in a hybrid processor system. Since the PUs operate on large amounts of data, careful data scheduling is required in order to get the best possible performance. For example, it is intuitively clear that the PUs that are configured earlier should get a larger fraction of the total input, but it is not clear what the optimum load fractions are. To get this, as well as to determine the maximum speedup that can be obtained under these constraints, a quantitative analysis of the system is necessary.

In order to carry out the analysis, we have modified the framework of divisible load theory (DLT) [14] to include partial reconfiguration. Our analysis gives us the solution for the following:

1) optimum number of PUs that are useful in getting the largest speedup (n^*);

Manuscript received June 30, 2005; revised January 28, 2006.

K. N. Vikram is with Siemens Corporate Technology, Bangalore, 560100, India (e-mail: vikramkn@ieee.org).

V. Vasudevan is with the Department of Electrical Engineering, Indian Institute of Technology Madras, Chennai 600036, India (e-mail: vinita@iitm.ac.in).

Digital Object Identifier 10.1109/TVLSI.2006.884052

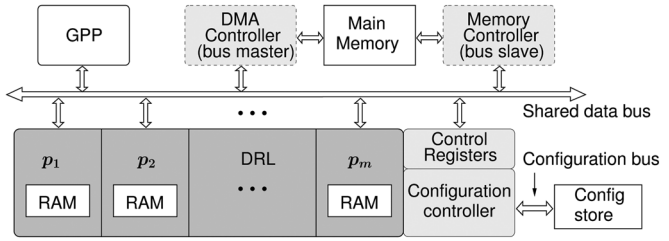

Fig. 1. Architecture model used for analysis of the hybrid processor architecture (a modified version of that presented in [16]). The block "GPP" includes the main processor as well as its associated cache.

2) actual processing time using n^* PUs;
3) corresponding load distribution.

In the analysis, we consider two general cases: 1) when load transfer to a PU is not possible in parallel with PU configuration/computation and 2) when load transfer to a PU is possible when the PU is either undergoing configuration or active in computation. Case 1) corresponds to the situation "without front-end" and case 2) corresponds to the situation "with front-end," in DLT parlance [14]. The analysis itself is quite general and does not assume anything about relative values of the reconfiguration time, bus speed, or the computation times. Therefore, it can also be used for multi-FPGA systems in which the configuration is carried out sequentially.

The rest of the paper is organized as follows. In Section II, we present the system architecture model used in our analysis. Section III gives a background on DLT, as well as the computation and communication model used. Section IV provides a motivating example using the case of two PUs. Section V provides a detailed analysis and the solution for total processing time for m PUs. In Section VI, a discussion of the analysis, its applicability, and limitations are presented. In Section VII, we present hardware simulation examples for 1-D discrete wavelet transform (DWT) and finite impulse response (FIR) filter, as well as details of an experiment carried out on an FPGA board. Section VIII contains the conclusions of this paper.

II. System Architecture Model

The system considered is a hybrid processor architecture. In the literature, various schemes of coupling between the dynamically reconfigurable logic (DRL) and the GPP have been proposed [2]. In this paper, it is assumed that the DRL has direct access to memory through a common bus. This loosely coupled architecture allows many local memory banks to be associated with the DRL and is, therefore, more suitable for data-parallel applications.

Fig. 1 shows the system architecture model. If the DRL is a slave, data transfer to the DRL is initiated and performed by a controller that performs direct memory access (DMA). The DMA controller is a bus master that fetches data from memory and sends it to the PUs. If the DRL is a bus master, the data transfer is performed by the PUs themselves, in which case a memory controller interfaces to the main memory. The memory controller is a bus slave which accepts requests from any bus master and provides the requested data from memory. The GPP is also a bus master; it typically controls the various operations and might also perform some tasks which are not mapped to the RF.

The DRL can be configured to accommodate m PUs, $p_1, \ldots, p_m$. Each PU has a local RAM required for storing data. This is similar to distributed memory multiprocessor architectures. Image processing and computer vision applications can be efficiently mapped onto such architectures [17]. The local RAM could either be an external SRAM [18] or the BlockRAMs present in Virtex FPGAs from Xilinx. The local RAM of all the PUs are a part of the GPP address space and, therefore, accessible by the GPP.

Reconfiguration of the DRL is under the control of a configuration controller (CC). The CC is programmed by the GPP to perform the required sequence of reconfigurations. The configuration data is typically stored in Flash memory, whose contents can be changed by the GPP whenever necessary. The starting address and size of configuration data is programmed into control registers in the CC by the GPP, before application execution begins. This is possible since the configuration strategy is determined offline. The CC is, therefore, quite simple, compared to the CC model described in the literature earlier [19]. As shown in Fig. 1, there is a separate configuration bus. For the analysis, we have ignored the overheads due to GPP control commands and the bus protocol. This is a good approximation since this overhead is typically small for a large input data size.

Before a quantitative analysis of the described system is carried out, we need to define the model for data computation and communication. Since this is based on DLT, we first present a brief background on DLT.

III. Background on DLT and Model for Computation and Communication

DLT has its origins in the paper by Cheng and Robertazzi [20], which was motivated by the requirement for processing large amounts of data in distributed intelligent sensor networks. DLT concerns itself with the analysis of parallel and distributed systems using linear models for data computation as well as communication, with the objective of obtaining the minimum possible processing time. In general, the theory can be applied to data-parallel tasks that operate on large amounts of data. The following basic assumptions form the foundations of DLT:

1) application load is arbitrarily divisible and the different load parts can be processed independently, without any precedence constraints;
2) time required for data transfer to any PU is linearly proportional to the amount of data transferred;
3) computation time at each PU increases linearly with the amount of data processed.

These assumptions hold good in a variety of applications, including signal/image processing and vision applications [21], and form the basis of our computation and communication model.

The notation that we use for our computation and communication model is given below. For convenience, this is the same as the notation used in [14] and [22]. The standard PU and the standard bus are those which are used as reference. These are "conveniently defined fictitious units" (quoted from [14]).

$T_{\rm cp}$ Time taken to process entire load by a standard PU.

$T_{\rm cm}$ Time taken to transfer entire load on a standard bus.

w Constant that is inversely proportional to the speed of a PU. Each PU can process the entire load in duration $wT_{\rm cp}$.

z Constant that is inversely proportional to the speed of the data bus. The entire load can be transferred over the bus in a duration $zT_{\rm cm}$.

α_i Fraction of total load assigned to PU p_i.

T_i Finish time of PU p_i. This corresponds to the instant p_i finishes computing its allocated load.

$T_f(m)$ Optimum processing time for m PUs, defined as $\max(T_1, T_2, \ldots, T_m)$.

From the definitions above, it is clear that the standard PU has $w = 1$, while the standard bus has $z = 1$. Even though in practice we deal only with the quantities $wT_{\rm cp}$ and $zT_{\rm cm}$ instead of w and z, w serves as a way to compare PUs with different speeds, whereas z can be used to compare buses with different speeds or bandwidths.

In addition to the notation presented, we use the following:

T_r Time taken to configure/reconfigure a single PU in the DRL. In this paper, we use the terms *configure* and *reconfigure* interchangeably.

As explained previously, given m PUs, we need to find the optimum load distribution so that the overall processing time is minimized. This can be expressed as

$$T_f(m) = \min_{\alpha \epsilon \{A\}} (\max(T_1, T_2, \ldots, T_m)). \tag{1}$$

Here, $\{A\}$ is the set of all possible load distributions. Given $\alpha_i, i = 1, .., m$ (i.e., a particular load distribution), each of the PUs finish in times $T_1, T_2, \ldots, T_m$. The finish time for the task is $\max(T_1, T_2, \ldots, T_m)$. The above equation indicates that we need to find the load distribution that gives the minimum finish time. In [23], it has been proven that for bus networks, the solution to the problem above gives the condition that all PUs stop computing at the same time, i.e., $T_1 = T_2 = \cdots = T_m$. This can be explained intuitively as follows. If any one of the PUs completes execution earlier, it is possible to allocate more load to that PU and, thus, achieve a smaller overall processing time.

The normalization equation for the load is

$$\sum_{i=1}^{m} \alpha_i = 1. \tag{2}$$

Using the notations given in this section, the time taken to transfer a load fraction α_i to p_i is $\alpha_i z T_{\rm cm}$, while the time taken by p_i for processing it is $\alpha_i w T_{\rm cp}$. Under the linearity assumption, the ratio of the processing time of a load to the time taken to transfer the load over the bus, is a constant for a given task

$$\sigma = \frac{wT_{\rm cp}}{zT_{\rm cm}}. \tag{3}$$

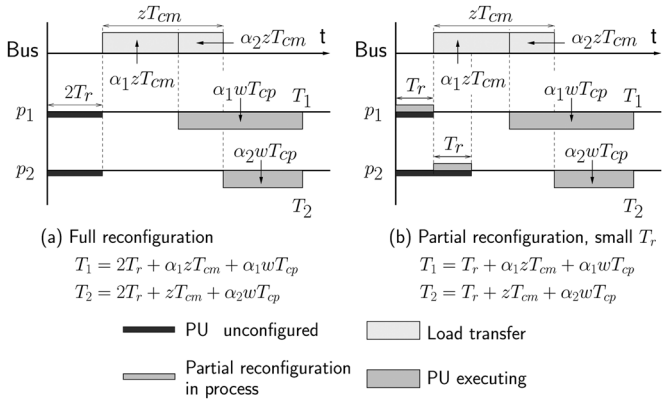

Fig. 2. Timing diagram of load distribution, for the case of full reconfiguration and partial reconfiguration with small T_r. The label "Bus" corresponds to the data bus. During partial reconfiguration, the PUs are configured one by one. In both cases, $\alpha_1 + \alpha_2 = 1$ and $T_f(2) = T_1 = T_2$ as explained in Section III.

The computation and communication model considered provides a tractable model for determining the solution for processing time [24]. However, reconfiguration introduces an additional dimension to the analysis using DLT. In fact, we show that there is also an upper limit to the number of PUs that are useful in computation. This is demonstrated with the help of a motivating example in Section IV.

IV. Motivating Example

We consider the case when there are two PUs of equal speed to be configured in the DRL ($m = 2$ in Fig. 1). Let us consider the case without front-end. We need to determine the configuration sequence and load distribution to the PUs such that the processing time is optimum. We have two options for distributing the load as follows.

1) *Using full reconfiguration:* In this case, the strategy is to first configure both the PUs by adopting full reconfiguration of the DRL. This is followed by optimal load distribution. This situation, shown in Fig. 2(a), is the same as the situation in DLT literature [23], except for an overhead of $2T_r$ for reconfiguration. Since the PUs finish simultaneously, we can equate T_1 and T_2 to get one equation in α_1 and α_2. The normalization (2) with $m = 2$ gives us another equation. These two equations are enough to solve for the two unknowns (load fractions) α_1 and α_2

$$\alpha_1 = \frac{zT_{\rm cm} + wT_{\rm cp}}{zT_{\rm cm} + 2wT_{\rm cp}} = \left(\frac{1+\sigma}{1+2\sigma}\right) \tag{4}$$

$$\alpha_2 = \frac{wT_{\rm cp}}{zT_{\rm cm} + 2wT_{\rm cp}} = \left(\frac{\sigma}{1+2\sigma}\right) \tag{5}$$

where σ is given by (3). The optimum processing time is $T_f = 2T_r + \alpha_1(zT_{\rm cm} + wT_{\rm cp})$, which is

$$T_f(2) = 2T_r + \frac{(1+\sigma)^2}{1+2\sigma} zT_{\rm cm}. \tag{6}$$

If we use partial reconfiguration, it is possible to initiate load transfer as soon as one of the PUs is configured. Using partial reconfiguration will, therefore, give a smaller processing time. This is now analyzed.

2) *Using partial reconfiguration:* The strategy adopted here is to partially reconfigure the DRL to accommodate p_1, followed by partial reconfiguration to accommodate p_2. As soon as p_1 is configured, load transfer to p_1 is initiated. Load transfer to p_1 is done in parallel with configuration of p_2. The load distribution, however, depends on the value of the reconfiguration time T_r. The different cases are considered separately in the sections that follow.

A. Small T_r

If the configuration time T_r is sufficiently small, it is possible for the configuration of p_2 to be completely hidden in the load transfer time for p_1. This situation is shown in Fig. 2(b). The configuration of p_2 does not affect the load distribution. Therefore, the load fractions are the same as that for the full reconfiguration case given by (4) and (5). The optimum processing time is now given by $T_f(2) = T_1 = T_r + \alpha_1(zT_{\rm cm} + wT_{\rm cp})$, i.e.,

$$T_f(2) = T_r + \frac{(1+\sigma)^2}{(1+2\sigma)} zT_{\rm cm}. \tag{7}$$

This is smaller than that for full reconfiguration by an amount equal to T_r. The configuration time of p_2 will be hidden as long as $T_r \leq \alpha_1 zT_{\rm cm}$, which gives the condition

$$T_r \leq \left(\frac{1+\sigma}{1+2\sigma}\right) zT_{\rm cm} \tag{8}$$

for (7) to hold true.

B. Large T_r

We now consider the case when T_r is so large that (8) is violated. If the same load fractions are used, p_2 will not be ready (configured) to accept data immediately after the load is delivered to p_1. One possible scheme is to feed as much data as possible to p_1 till p_2 becomes ready, followed by the transfer of the remaining load to p_2. In this case, $\alpha_1 zT_{\rm cm} = T_r$ and the finish times of the PUs are given by

$$\begin{aligned} T_1 &= T_r + \alpha_1 zT_{\rm cm} + \alpha_1 wT_{\rm cp} = (2+\sigma)T_r \qquad (9)\\ T_2 &= 2T_r + \alpha_2 zT_{\rm cm} + \alpha_2 wT_{\rm cp} \\ &= (1-\sigma)T_r + (1+\sigma)zT_{\rm cm}. \qquad (10) \end{aligned}$$

The simplifications in terms of σ are based on $\alpha_1 zT_{\rm cm} = T_r$ and $\alpha_1 + \alpha_2 = 1$. Since (8) is violated, (9) and (10) indicate that $T_1 > T_2$. This is shown in Fig. 3(a). The situation depicted in this timing diagram is valid as long as $T_r < zT_{\rm cm}$.

However, since $T_1 > T_2$, the processing time can be reduced by transferring a portion of load meant for p_1, to p_2. This means that some portion of the reconfiguration time of p_2 will be uncovered, giving rise to an idle time θ, on the data bus. The situation is depicted in Fig. 3(b). Clearly, data transfer to p_2 must begin as soon as the configuration of p_2 is over, to ensure minimum processing time. For the situation shown in Fig. 3(b), the expressions for finish times of the PUs are

$$\begin{aligned} T_1 &= T_r + \alpha_1(zT_{\rm cm} + wT_{\rm cp}) \qquad (11)\\ T_2 &= 2T_r + \alpha_2(zT_{\rm cm} + wT_{\rm cp}). \qquad (12) \end{aligned}$$

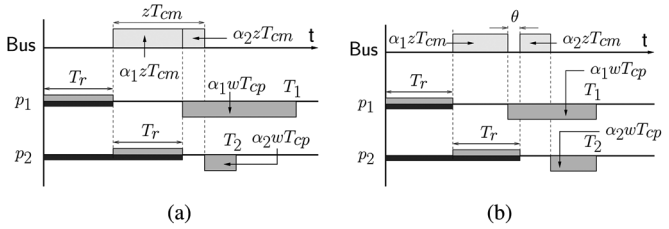

Fig. 3. Different options when reconfiguration time is large [(8) is violated]. Option (a) is suboptimal since some load allocated to p_1 can be transferred to p_2 [as shown in (b)], to achieve smaller processing time $T_f(2) = T_1 = T_2$. (a) Large T_r, suboptimal. (b) Large T_r, optimal.

Using the normalization (2) with $m = 2$ and equating T_1 and T_2, we get the following expressions for the load fractions and optimum processing time

$$\alpha_{1,2} = \frac{1}{2} \pm \left(\frac{1}{1+\sigma}\right)\frac{T_r}{2zT_{\rm cm}} \tag{13}$$

$$T_f(2) = \frac{3T_r}{2} + \left(\frac{1+\sigma}{2}\right) zT_{\rm cm}. \tag{14}$$

As T_r becomes larger, α_1 increases and α_2 decreases. Eventually, when $T_r = (1+\sigma)zT_{\rm cm}$, $\alpha_2 = 0$ and $\alpha_1 = 1$. This essentially means that the entire load can be processed by one PU and the second PU becomes unnecessary. The processing time using a single PU is

$$T_f(1) = T_r + zT_{\rm cm} + wT_{\rm cp} = T_r + (1+\sigma)zT_{\rm cm}. \tag{15}$$

For all values of T_r larger than $(1+\sigma)zT_{\rm cm}$, it is clear that $T_f(1) < 2T_r$. This means that a single PU can finish processing the entire load before p_2 is configured. Therefore, it is not useful to have more than one PU and the optimum number of PUs is one.

The case of two PUs demonstrates that the optimal load distribution scheme can be different for different values of the reconfiguration time T_r. Choice of a particular load distribution as well as the number of PUs must be made depending on the value of T_r. In Section V, we extend the analysis for m PUs, where m is the maximum number of PUs that can be accommodated within the RF.

V. Analysis With m Processing Units

For the system considered in Section II, the analysis is carried out for two cases—case without front-end and the case with front-end. These are now considered.

A. Without Front-End

This case is similar to the one considered in the example in the previous section. Load transfer is not possible to a PU in parallel with configuration or computation. This analysis can be used for architectures that satisfy the following conditions.

1) Either a) the PUs are slaves and the DMA controller cannot directly access the RAM within a PU before configuration of the PU, i.e., the PU contains the interface between the RAM and the data bus, or b) the PUs are bus-masters and,

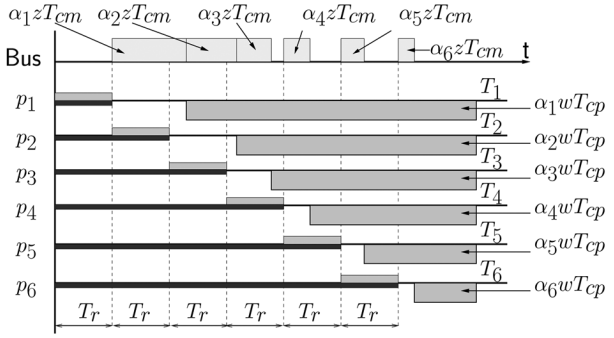

Fig. 4. Timing diagram for case without front-end: $n = 6$ and $q = 3$.

hence, fetch data from memory by themselves. Therefore, data transfer is not possible before configuration of the PU.

2) Either 1) the RAM associated with each PU is single-ported, or 2) the RAM associated with each PU is multiported, but the PUs are designed so that all the ports are occupied during computation. Therefore, data transfer to a PU is not possible while it is busy with computation. Efficient pipelined implementations of data-parallel tasks normally use multiple input and output streams [11], where each data stream corresponds to a dedicated RAM port.

The PUs are configured one after the other, in the order $p_1, \ldots, p_m$. We saw in the previous section that all the available PUs may not contribute towards the optimal solution. Let the number of PUs that participate in computation be n ($n \leq m$). Since the PU speeds are identical, the load fractions decrease monotonically from p_1 to p_n to ensure that all PUs stop computing simultaneously. Depending on the relative values of T_r and the load fractions, it is possible that the reconfiguration time is hidden by the load transfer time for some or all of the PUs (except p_1). Let the reconfiguration time be hidden for the PUs $p_2, \ldots, p_q$ and let the reconfiguration of p_{q+1} be exposed by an idle gap on the data bus after load transfer to p_q. Fig. 4 shows this for the specific case of $n = 6$ and $q = 3$. Since no gap occurs after load transfer to PUs $p_1, \ldots, p_{q-1}$, we have the following relations:

$$\begin{aligned} T_r &\leq \alpha_1 zT_{\rm cm} \\ 2T_r &\leq (\alpha_1 + \alpha_2) zT_{\rm cm} \\ &\vdots \\ (q-1)T_r &\leq \sum_{i=1}^{q-1} \alpha_i zT_{\rm cm}. \end{aligned} \tag{16}$$

Among these, the last equation is the most restrictive, since α_i values monotonically decrease with i. Also, since a gap occurs after load transfer to p_q, we have

$$qT_r > \sum_{i=1}^{q} \alpha_i zT_{\rm cm}. \tag{17}$$

From (16) and (17), we can see that $\alpha_q zT_{\rm cm} < T_r$. Since the load fractions are monotonically decreasing, we also have

$$\alpha_i zT_{\rm cm} < T_r, \qquad i = (q+1), \ldots, n. \tag{18}$$

To ensure minimum possible processing time, load transfer to p_{q+1} must start immediately after configuration. It follows from (18), therefore, that a bus idle gap exists after load transfer to p_{q+1}. Similarly, idle gaps exist after load transfer to each of the PUs $p_{q+2}, \ldots, p_{n-1}$. This is depicted in Fig. 4. From the timing diagram in Fig. 4, the finish times of the PUs can be written as

$$T_i = \begin{cases} T_r + \left(\sum_{j=1}^{i} \alpha_j \right) zT_{\rm cm} + \alpha_i wT_{\rm cp}, & i = 1, \ldots, q \\ iT_r + \alpha_i (zT_{\rm cm} + wT_{\rm cp}), & i = (q+1), \ldots, n. \end{cases} \tag{19}$$

Equating finish time for the first q PUs, we have $T_i = T_{i-1}$ for $i = 2, \ldots, q$, which gives

$$\alpha_{i-1} wT_{\rm cp} = \alpha_i zT_{\rm cm} + \alpha_i wT_{\rm cp}, \quad i = 2, \ldots, q. \tag{20}$$

Using (3) and (20), we get

$$\alpha_i = \left(\frac{\sigma}{1+\sigma} \right) \alpha_{i-1} = \kappa \alpha_{i-1} \tag{21}$$

where $\kappa = wT_{\rm cp}/(wT_{\rm cp} + zT_{\rm cm}) = (\sigma)/(\sigma + 1)$ is the fraction of time spent in computation. We refer to κ as the PU *speed factor*. From (21), we get

$$\alpha_i = \kappa^{i-1} \alpha_1, \qquad i = 2, \ldots, q. \tag{22}$$

Equating the finish times for the remaining PUs, we have $T_i = T_n$ for $i = (q+1), \ldots, (n-1)$, which gives

$$\begin{aligned} \alpha_i &= \alpha_n + (n-i)(1-\kappa) \left(\frac{T_r}{zT_{\rm cm}} \right) \\ i &= (q+1), \ldots, (n-1). \end{aligned} \tag{23}$$

Using $T_1 = T_n$, we can relate the load fractions α_1 and α_n as

$$\alpha_1 = \alpha_n + (n-1)(1-\kappa) \frac{T_r}{zT_{\rm cm}}. \tag{24}$$

Using the normalization (2) for n PUs and substituting for α_i from (22) and (23) and using (24), α_n can be written as

$$\alpha_n = \frac{1 - (1-\kappa) \left[\frac{(n-q-1)(n-q)}{2} + (n-1) \left(\frac{1-\kappa^q}{1-\kappa} \right) \right] \frac{T_r}{zT_{\rm cm}}}{\left(\frac{1-\kappa^q}{1-\kappa} \right) + (n-q)}. \tag{25}$$

Using (24) and (25), the expression for α_1 is

$$\alpha_1 = \frac{1 + (n-q)(n+q-1)(1-\kappa) \frac{T_r}{2zT_{\rm cm}}}{\left(\frac{1-\kappa^q}{1-\kappa} \right) + (n-q)}. \tag{26}$$

The optimum processing time is given by

$$T_f(n) = T_r + \alpha_1(zT_{\rm cm} + wT_{\rm cp}) \tag{27}$$

where α_1 is given by (26). The value of q in (26) can be obtained as follows. The reconfiguration time T_r must satisfy (16) and (17). Therefore, we can combine (16) and (17) to get

$$\frac{1}{q}\sum_{i=1}^{q}\alpha_i zT_{\rm cm} < T_r \le \frac{1}{q-1}\sum_{i=1}^{q-1}\alpha_i zT_{\rm cm}. \tag{28}$$

Let us now consider the inequality

$$T_r > \frac{1}{p}\sum_{i=1}^{p}\alpha_i zT_{\rm cm} \tag{29}$$

where $p \le q$. In the inequality above, α_i is a function of T_r. Substituting for α_i using (22) and using the expression for α_1 from (26), the previous inequality reduces to

$$D_n(p,q)T_r > N_n(p,q)zT_{\rm cm} \tag{30}$$

where $N_n(p,q) = (1-\kappa^p)/(1-\kappa)$ and

$$D_n(p,q) = \left[p\left\{\left(\frac{1-\kappa^q}{1-\kappa}\right) + (n-q)\right\} - \frac{(n-q)(n+q-1)(1-\kappa^p)}{2}\right]. \tag{31}$$

Now substituting $p = q$ in (30) will give the left-hand side of (28). Reversing the inequality and using $p = q-1$, we get the right-hand side of (28). Therefore, (28) can be written as the following two relations:

$$D_n(p,q)T_r > N_n(p,q)zT_{\rm cm}, \qquad p = q \tag{32}$$

$$D_n(p,q)T_r \le N_n(p,q)zT_{\rm cm}, \qquad p = q-1. \tag{33}$$

T_r must satisfy both these conditions for $q \in \{2,\ldots,n-1\}$. For the case $q = n$ (no gaps), the lower limit on T_r implied by (32) is not necessary, whereas for $q = 1$ (gaps after each PU load transfer), the upper limit on T_r indicated by (33) is not necessary. We can, therefore, write down the conditions that T_r must satisfy for different values of q

$$\begin{cases} D_n(n-1,n)T_r \le N_n(n-1,n)zT_{\rm cm}, & q = n \\ (32)\text{ and }(33), & q = 2,\ldots,n-1 \\ D_n(1,1)T_r > N_n(1,1)zT_{\rm cm}, & q = 1. \end{cases} \tag{34}$$

We must choose the value of q such that T_r satisfies the appropriate condition for the selected q, as given by (34). Once q has been determined, we can compute the load fractions α_i using (22), (23), (25), and (26). The optimum processing time can then be computed using (27). It may be noted that the intervals of T_r, implied by (34) for different values of q, abut each other and, therefore, span a contiguous range of possible values of T_r. This is clear from the fact that

$$D_n(q-1,q) = D_n(q-1,q-1), \qquad q = 2,\ldots,n$$
$$N_n(q-1,q) = N_n(q-1,q-1), \qquad q = 2,\ldots,n$$

1: Compute $T_f(1)$ using (15).
2: **if** $T_f(1) \le 2T_r$ **then**
3: {Single PU finishes before p_2 is configured. Therefore $n^* = 1$. }
4: $n \leftarrow 1$, $\alpha_1 \leftarrow 1$, $q \leftarrow 1$. Exit ;
5: **end if**
6: {Since $T_f(1) > 2T_r$, $n^* > 1$. Search for n^*}
7: **for** $n = 2$ to m **do**
8: { For each n, perform a search for the value of q }
9: **for** $q = 1$ to n **do**
10: **if** T_r satisfies (34) for current value of q **then**
11: break ;
12: **end if**
13: **end for**
14: Compute α_i $(i = 1,\ldots,n)$ using (22), (23), (25) and (26).
15: Compute the processing time $T_f(n)$ using (27).
16: **if** $T_f(n) \le (n+1)T_r$ **then**
17: { Load is processed before p_{n+1} is configured. Therefore, current n is n^*}
18: Exit ;
19: **end if**
20: **end for**

Fig. 5. Without front-end: algorithm to determine maximum number of PUs n that can take part in computation, out of the m available PUs. The corresponding value of q, load distribution, and processing time are also obtained.

With a nonzero reconfiguration time T_r, it is possible that all the m available PUs are not used for computation. In fact, if the processing time using $n < m$ PUs is less than or equal to the time instant p_{n+1} becomes ready for computation, we can be sure that p_{n+1} (and the remaining PUs) cannot contribute towards reducing the processing time. This can be used to determine the maximum number of PUs that are useful. The procedure is given in Fig. 5. The algorithm performs two searches—for q and for n^*. In the worst case, $q = n$ $\forall\ n = 1,\ldots,m$ and $n^* = m$, in which case the algorithm runtime complexity is $O(m^2)$. The algorithm is run offline, before the start of application execution, and the value of n^* and load fractions are determined beforehand.

We now define two quantities, the normalized processing time ϕ and the normalized reconfiguration time ρ

$$\phi = \frac{T_f(n)}{zT_{\rm cm}}, \qquad \rho = \frac{T_r}{zT_{\rm cm}}. \tag{35}$$

In Fig. 6(a), the plot of the normalized processing time ϕ with respect to the number of PUs utilized, is shown for $\kappa = 0.94$. This is the value of κ for one of the examples described in Section VII. From Fig. 6(a), we can see that the processing time reduces with an increase in the number of PUs. For a given ρ, there exists a maximum number of PUs n^*, beyond which it is not possible to get a further reduction in the processing time. Therefore, for minimum processing time, one must use n^* PUs in the system. Also, it can be seen that for a fixed PU speed, the

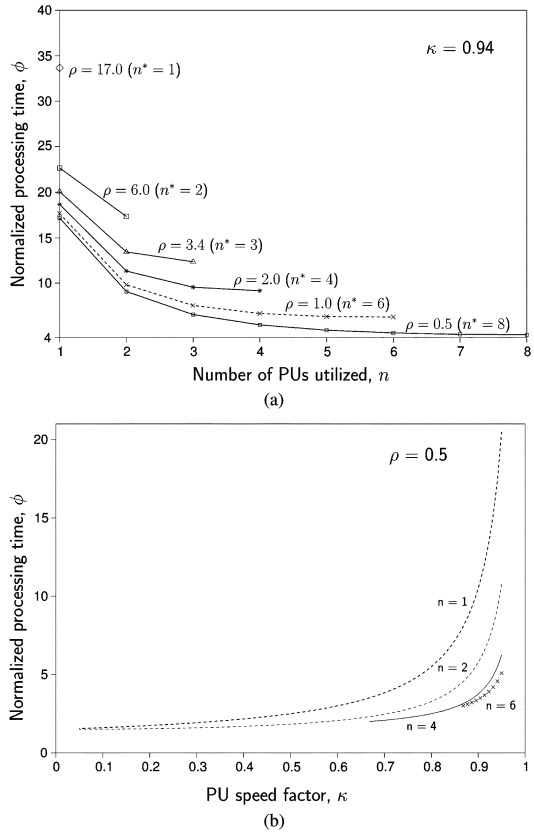

Fig. 6. Plot of the normalized processing time ϕ for the case without front-end. (a) Shows the variation with different number of PUs used, with each curve for a different value of reconfiguration time. (b) Shows the variation with the PU speed factor κ. Each curve (for a particular n) is plotted only for those values of κ for which a solution exists, i.e., $n \leq n^*$. (a) ϕ versus n, $\kappa = 0.94$, (b) ϕ versus κ, $\rho = 0.5$.

processing time increases with ρ, which is as expected. Also, as expected, the number of useful PUs increases as ρ decreases. In the limit when $T_r \rightarrow 0$, the theory is identical to the conventional DLT and T_f keeps reducing monotonically with the number of PUs, with no limit on the maximum number of PUs.

A plot of the variation of the normalized processing time with the PU speed factor κ is given in Fig. 6(b). $\kappa \rightarrow 1$ as $wT_{\rm cp} \rightarrow \infty$ and $\kappa = 0$ when $wT_{\rm cp} = 0$. We can see that the processing time increases with κ, as expected. Each curve in the plot is for a particular value of number of PUs n used. The curves (each for fixed n) are plotted only for those values of κ which give a valid solution, i.e., $n \leq n^*$. We can see from Fig. 6(b) that solution with more PUs exists only for slower PUs, i.e., for large κ. This is as expected.

B. With Front-End

Here, we consider the case when data transfer to a PU is possible while it is being configured or while it is performing computation. This analysis can be used for architectures that satisfy the following conditions.

1) The PUs are slaves and the DMA controller has access to the RAM associated with a PU even before the PU is configured. This is possible if a fixed interface is provided between the RAM and the data bus and the RAM is external to the PU.

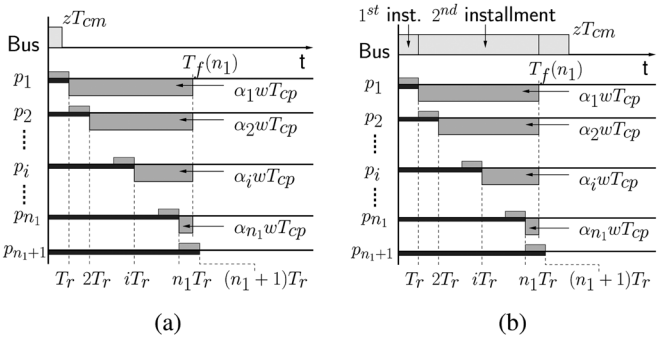

Fig. 7. Timing diagram for computation of first installment, for the two cases of value of T_r relative to $zT_{\rm cm}$. $n_1 \leq m$ PUs participate in computation. The subscript 1 in n_1 indicates that it is the first installment. (a)$zT_{\rm cm} \leq T_r$, (b) $zT_{\rm cm} > T_r$.

2) The RAM associated with each PU has a minimum of two ports. If the RAM is dual-ported, one port can be utilized for data input/output during PU computation, while the other port can be used by the DMA controller to transfer data to the RAM in parallel. If the RAM is multi-ported, the PUs are designed so that during computation, one RAM port is left free to allow for load transfer.

The situation considered here is a special case of the general situation of processors with arbitrary release times on a bus network considered in [22]. The release times correspond to the time instants when the PUs are ready to start computation, i.e., after the PUs are configured. All the different cases that need to be considered have been treated exhaustively in [22]. We have made some improvements to the solution presented in [22], which results in a slightly different scheduling algorithm from the one proposed in [22]. For the sake of completeness, we present the complete analysis. Our contributions are pointed out wherever applicable.

Using the notation in [22], the release time of PU p_i is denoted as t_i. In our case, the release times t_i of the PUs correspond to the time they are ready for computation, after configuration. If the PUs are each configured successively, the release times are

$$t_i = iT_r, \qquad i = 1, \ldots, m. \tag{36}$$

Depending on the value of T_r, there are two cases to be considered.

1) Case 1 ($zT_{\rm cm} \leq T_r$): All the load is transferred before the first PU is configured. The entire load is processed in a single installment. Let n be the number of PUs that participate in computation, to give a minimum finish time. As derived in [22], the load fractions and optimum processing time are given by

$$\alpha_i = \frac{1}{wT_{\rm cp}} \left[\frac{1}{n} \left\{ wT_{\rm cp} + \sum_{j=1}^{n} t_j \right\} - t_i \right], \quad i = 1, \ldots, n, \quad n \leq m \tag{37}$$

$$T_f(n) = t_1 + \alpha_1 wT_{\rm cp} \tag{38}$$

where t_i is given by (36). The timing diagram is shown in Fig. 7(a). The number of PUs that participate in computation is determined based on the fact that the load fraction values should be positive quantities. Knowing that the load fractions

```
1: for n = m to 1 do
2:    Compute α_n using (37) (Case 1) or (39) (Case 2).
3:    if α_n > 0 then
4:       Exit ;
5:    end if
6: end for
```

Fig. 8. Algorithm to determine n, the number of PUs that can participate in computation, for case 1 as well as case 2.

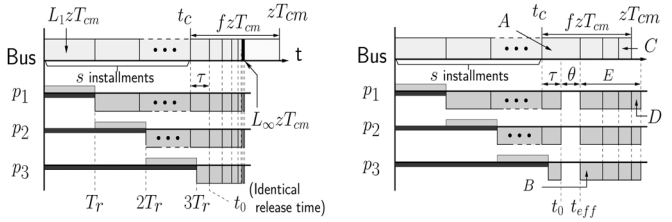

Fig. 9. *Special case* for $m = 3$ as shown in (a) it is not possible for the PUs to consume all the load. (b) Shows the proposed solution for $k_0 = 4$, where A, B, C, D, and E are, respectively, $L_{s+1} zT_{\rm cm}$, $L_{s+1} wT_{\rm cp}/n$, $L_{s+k_0} zT_{\rm cm}$, $L_{s+k_0} wT_{\rm cp}/n$, and $fwT_{\rm cp}/n$. (a) The *special case*. (b) The proposed solution.

decrease monotonically, it is enough to check for $\alpha_n > 0$. The iterative procedure to determine n is described in [25], presented here in Fig. 8. As before, the value of n obtained will satisfy $T_f(n) < (n+1)T_r$.

2) Case 2 ($zT_{\rm cm} > T_r$): In this case, the load is delivered in multiple installments and the load distribution strategy is as follows. First, as much load as possible is transferred to the PUs within duration T_r. This forms the first installment. The load fractions and finish time for the first installment, as derived in [22], are

$$\alpha_i = \frac{1}{wT_{\rm cp}} \left[\frac{1}{n} \left\{ \frac{t_1 wT_{\rm cp}}{zT_{\rm cm}} + \sum_{j=1}^{n} t_j \right\} - t_i \right], \quad i = 1, \ldots, n, \quad n \le m \tag{39}$$

$$T_f(n) = t_1 + \alpha_1 wT_{\rm cp} \tag{40}$$

where t_i is given by (36). The number of PUs n that participate in processing the first installment is obtained using the algorithm in Fig. 8. For the purpose of discussion, let us denote the number of PUs utilized in the ith installment as n_i. All the n_1 PUs finish computation of the first installment at time $T_f(n_1)$. During computation, the second installment is loaded in the RAM for a duration equal to $\min(T_f(n_1), zT_{\rm cm}) - T_r$. For the second installment, the release times of the participating PUs is given by

$$t_i = \begin{cases} T_f(n_1), & i = 1, \ldots, n_1 \\ iT_r, & i = (n_1+1), \ldots, n_2. \end{cases} \tag{41}$$

If $T_f(n_1) \ge zT_{\rm cm}$, only two installments are sufficient to process the entire load. The optimum processing time is $T_f(n_2)$. The situation is similar to Case 1 and the same procedure is used to obtain the load fractions and finish time. On the other hand, if $T_f(n_1) < zT_{\rm cm}$, more than two installments are needed to process the entire load. The load fractions and finish time for the second installment are obtained in the same manner as the first installment of Case 2. The process is continued using as many installments as required, till all the load is consumed.

Consideration of a special case: Let us consider a situation when the number of participating PUs n becomes equal to m (maximum possible number of PUs) after $s-1$ installments. Then, after s installments, the PUs will have identical release times for all the remaining installments. In this case, using (39) and (40) with identical t_i's, it turns out that the load is distributed equally among all the $n(=m)$ PUs. If L_j is the load fraction distributed in the jth installment $(j > s)$, the execution time for the installment is $L_j wT_{\rm cp}/n$. As before, the next installment $(L_{j+1} zT_{\rm cm})$ is distributed during this duration. Therefore, we have $L_{j+1} zT_{\rm cm} = L_j wT_{\rm cp}/n$ which can be written as

$$L_{j+1} = L_j \gamma_n, \qquad j > s \tag{42}$$

where $\gamma_n = wT_{\rm cp}/(nzT_{\rm cm})$. If $\gamma_n < 1$, the successive load fractions keep reducing. In this case, there is one difficulty. This is when the load fractions reduce to an infinitesimally small value before all the load is consumed. This scenario is depicted in Fig. 9(a) for $m = 3$. In the figure, t_c corresponds to the time duration for the first s installments. After processing s installments, the PUs have an identical release time t_0. The load remaining after distributing s installments is $fzT_{\rm cm}$, $f < 1$. In the situation depicted, it is not possible to consume all the remaining load $fzT_{\rm cm}$. This is mathematically captured as

$$t_0 + [\text{Execution time for load } fzT_{\rm cm} \text{ using } n \text{ PUs}] \le zT_{\rm cm} \tag{43}$$

which can be written as

$$t_0 + f\frac{wT_{\rm cp}}{n} \le zT_{\rm cm}. \tag{44}$$

Denoting $\tau = (t_0 - t_c)$ [Fig. 9(a)], the previous equation can be rewritten as

$$\tau \le fzT_{\rm cm}(1 - \gamma_n). \tag{45}$$

We shall refer to the situation when (45) holds as the *special case*. In [22], a heuristic solution for the *special case* is presented, wherein the processor execution is delayed by a duration $\delta = zT_{\rm cm} - \tau(1+\gamma_n)$ so that the entire load can be processed in two installments. When this heuristic is used, the processing time does not always decrease monotonically with increase in the number of PUs utilized. An example when this occurs is for $\kappa = 0.8$ and $\rho = 0.1$, shown in Fig. 10 (dashed line). This type of behavior of processing time is undesirable.

We present an improved solution for the *special case*, based on a multi-installment strategy. This is depicted in Fig. 9(b). The basic idea of delaying PU execution is the same as in [22]. Let the idle time of the PUs be θ. Then the effective release time is $t_{\rm eff} = t_0 + \theta$. From Fig. 9(b), the total load fraction delivered in the $(s+1)^{th}$ installment to all PUs is

$$L_{s+1} = \frac{\tau_{\rm eff}}{zT_{\rm cm}} \tag{46}$$

where

$$\tau_{\rm eff} = \tau + \theta = t_{\rm eff} - t_c. \tag{47}$$

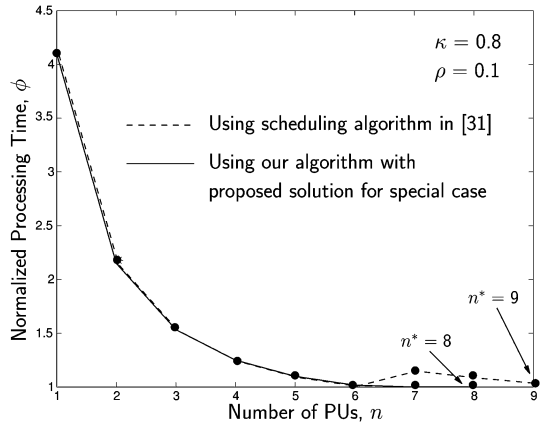

Fig. 10. Comparison of the plot of ϕ versus n obtained using the scheduling algorithm in [22] and by using our proposed algorithm. The plots are for $\kappa = 0.8$ and $\rho = 0.1$.

1: $t_c = 0, \quad \sigma = wT_{cp}/(zT_{cm})$, $t_i = iT_r$ for $i = 1, \ldots, m$
2: Choose $k_0 \gg 1$.
3: **while** $t_1 < zT_{cm}$ **do**
4: Obtain n using algorithm in Fig. 8 (Case 2)
5: **if** $t_1 = \cdots = t_n$ **then**
6: $\tau \leftarrow t_1 - t_c, \quad f \leftarrow 1 - t_c/(zT_{cm}), \quad \gamma_n \leftarrow \sigma/n$
7: Check for *special case* using (45).
8: **if** *special case* **then**
9: Use the proposed solution for *special case* with k_0 installments. Load distribution and $T_f(n)$ are given by (46)-(51). Exit ;
10: **end if**
11: **end if**
12: Obtain load distribution and $T_f(n)$ for a single installment using Case 2.
13: $t_c \leftarrow t_c +$ Duration of load installment
14: $t_i = T_f(n), \quad i = 1, \ldots, n$
15: **end while**
16: Obtain n using algorithm in Fig. 8 (Case 1)
17: Obtain load distribution and $T_f(n)$ for last load installment using Case 1. Exit ;

Fig. 11. With front-end: complete algorithm for determining the load distribution and processing time.

As before, the execution time for jth installment is $L_j w T_{\rm cp}/n$ for $j > s$. We fix the total number of installments to $s + k_0$ and then choose $\tau_{\rm eff}$ such that only the last installment computation occurs after the time instant $zT_{\rm cm}$. That is

$$t_{\rm eff} + \sum_{j=s+1}^{s+k_0-1} \frac{L_j}{n} w T_{\rm cp} = zT_{\rm cm}. \tag{48}$$

The load installments are still related by (42), which gives

$$L_j = L_{s+1}\gamma_n^{j-(s+1)}, \qquad j = (s+1), \ldots, (s+k_0). \tag{49}$$

Using (46)–(49), we get

$$\tau_{\rm eff} = \frac{f z T_{\rm cm}}{\left(1 + \sum_{j=1}^{k_0-1} \gamma_n^j\right)}. \tag{50}$$

The PU idle time θ is then $\tau_{\rm eff} - \tau$. The execution time for processing the load $fzT_{\rm cm}$ is $fwT_{\rm cp}/n$. The finish time is, therefore, given by

$$T_f(n) = t_c + \tau_{\rm eff} + \frac{f w T_{\rm cp}}{n}. \tag{51}$$

Fig. 9(b) shows the proposed solution for $m = 3$ and $k_0 = 4$. As described earlier, the *special case* occurs when $\gamma_n < 1$. Therefore, (50) and (51) indicate that $T_f(n) \rightarrow zT_{\rm cm}$ when $k_0 \rightarrow \infty$. In other words, one can achieve a finish time as close to $zT_{\rm cm}$ as desired, by choosing an appropriately large value of k_0. For an infinitely large number of installments, the proposed solution is optimum, since the finish time cannot possibly be reduced below $zT_{\rm cm}$ in any load distribution scheme.

The *special case* can also occur when $n < m$. This is possible if computation times are small and the load fractions tend to zero even before p_{n+1} is configured. However, as long as the condition for special case (45) holds, we need not consider any additional PU, since it is possible to get a finish time close to $zT_{\rm cm}$ with our multi-installment strategy. If n is small, $\tau_{\rm eff}$ and k_o can be adjusted to get a finish time as close to $zT_{\rm cm}$ as desired.

The complete solution procedure is given in Fig. 11. The algorithm runtime complexity depends on the number of load installments, which in turn depends on the values of T_r, $zT_{\rm cm}$, $wT_{\rm cp}$, and m. In the algorithm in Fig. 11, we have not considered the case when $T_r = 0$, since zero reconfiguration time does not occur in practice. For the *special case*, it was observed that a value of $k_0 = 20$ is generally sufficient to get good results. This is depicted in Fig. 10 (solid line) for the case $\kappa = 0.8$ and $\rho = 0.1$. As desired, the processing time decreases monotonically with the number of PUs utilized.

The variation of the normalized processing time (ϕ) with n and κ is similar to that for the case without front-end Fig. 6. There exists an optimal number of PUs n^*, which increases as ρ decreases. Also, the processing time increases with κ and more PUs can be used for larger values of κ.

VI. Discussions

Our analysis gives us the optimum load fractions as well as the maximum number of useful PUs, for a given reconfiguration delay and computation speed relative to the bus speed (ρ and σ). This data can be used in two ways. Given an area constraint for the DRL, it is possible to know the maximum number of PUs m that can be accommodated in the DRL. If $m \geq n^*$, our analysis shows that we need to use only n^* PUs and some of the DRL area will remain unused. If $m < n^*$, it is possible to get the finish time using our analysis, but it will not be the best possible speedup that can be obtained for the given values of ρ and σ. Alternatively, if we want a certain finish time, it is possible to use this analysis to find the minimum area required to get the required finish time. For example, if we want $8 < \phi < 10$ with $\rho = 0.5$ and $\kappa = 0.94$ in a "no front-end" architecture, it can be seen from Fig. 6(b) that we need not use more than two PUs. This information can be used within any task-based scheduler to get an optimized schedule.

The analysis presented in the previous section assumes that after processing, the PU output result data remains within the local memory. Since the local memory is part of the overall memory address space, the output data can either be used by the

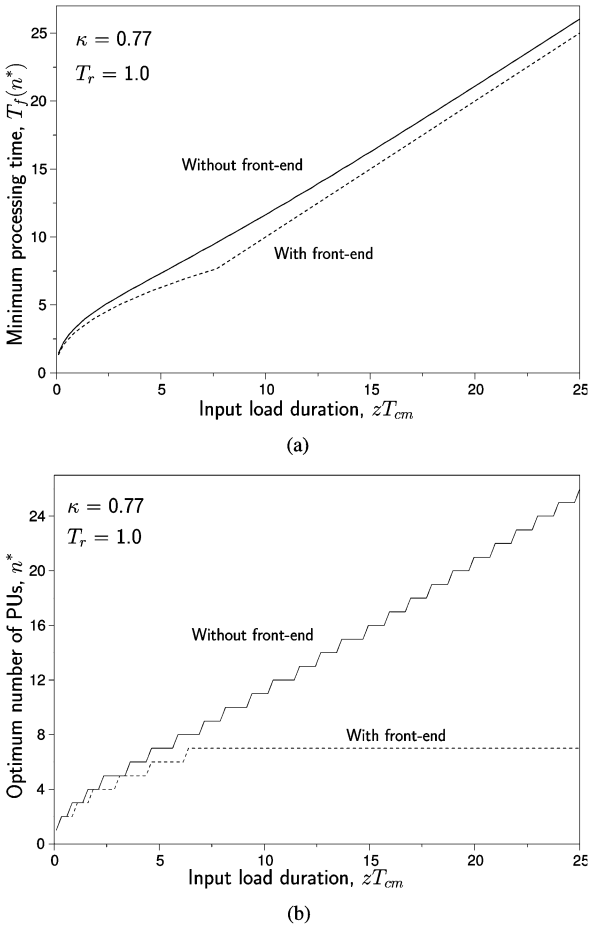

Fig. 12. Variation of minimum processing time and the optimum number of PUs, with input load duration. (a) $T_f(n^*)$ versus $zT_{\rm cm}$, (b) n^* versus $zT_{\rm cm}$.

GPP or by a subsequent task to be performed within the DRL or by any other peripheral. In this case, there is no need to transfer the data to external memory. However, the analysis will not be valid if the DRL has to be completely reconfigured, immediately after the PUs finish execution, to perform another task that requires the local memory. In this case, we need to transfer data from the local memory to external memory. For such a situation, the analysis presented in this paper will not give the optimum finish time. The result transfer time must then be taken into consideration along with a bus bandwidth constraint. The details of this analysis are given in [26] and [27].

Fig. 12 shows the variation of the minimum processing time as well as the corresponding value of n^*, with change in the load transfer duration $zT_{\rm cm}$. The figure shows it for both cases, i.e., with and without front-end. For the front-end case, we have set $k_0 = 20$. The chosen value of $\kappa = 0.77$ is for the FIR filter example to be discussed in Section VII. The load duration can increase either as a result of increase in the input data size or due to a reduction in the data bus bandwidth. As the load duration increases, it is possible to use more PUs in the case without front-end, to get an optimum processing time. This is, however, not the case with front-end, since beyond a certain point, the *special case* comes into play, which eliminates the need for additional PUs to reduce the processing time.

From Fig. 12, we observe that n^* for the front-end case is always less than or equal to that for the case without front-end. This means that a lower area is occupied in the DRL for the front-end case. In addition, processing time is smaller for the front-end case. The front-end architecture for the DRL, therefore, seems to be better. However, as mentioned earlier, one RAM port must be left free during computation in the case with front-end, to allow for load transfer from the data bus. For example, if we have dual-ported RAM within each PU, the case without front-end can use both the ports during computation. On the other hand, in the front-end architecture, the PU computation unit has access to only one RAM port. This can result in a reduction in computation speed. In other words, the apparent advantage of the front-end architecture could be offset by a degradation in computation speed. Choice of the appropriate architecture can be made only after quantifying the speed degradation, which is application dependent.

One important aspect of the problem considered in this paper is that RTR is used because all parts of an application cannot be simultaneously mapped to the RF. During the course of execution of an application, tasks are sequentially configured on the RF, whenever they are encountered. Our work aims to obtain the minimum possible processing time, whenever the RF needs to be configured to accommodate a new task. This work is orthogonal to the use of reconfiguration for achieving larger functional density, reported elsewhere [28].

VII. Example Applications

We have applied the theory developed in the previous sections to two examples—namely, 1-D discrete wavelet transform (DWT) and FIR filter. Hardware simulation results are presented for both examples. In addition, experimental proof-of-concept on actual FPGA hardware is presented for the FIR filter example.

A. Simulation Details and Results

For each example application, we have designed dedicated PUs to perform the required function. The hardware description for a single PU was targeted onto a Xilinx FPGA of the Virtex family, which supports partial reconfiguration with one column or frame being the basic unit for reconfiguration. Partial reconfiguration of Xilinx FPGAs is done by using partial bitstreams. In order to obtain partial bitstreams for each of the PUs, we have used the module-based partial reconfiguration flow described in [3], with each PU corresponding to a module. Xilinx ISE 6.3 (Service Pack 3) software was used for generating the required partial bitstreams. For configuration clock frequency less than 50 MHz, the number of configuration clock cycles for reconfiguration using the SelectMAP interface directly corresponds to the number of bytes in the partial bitstream [29]. The configuration clock can be different from the system clock used by the PUs during computation. The value of ρ is then calculated as

$$\rho = \frac{\text{Number of system clock cycles for configuration}}{\text{Number of system clock cycles for total load transfer}}. \quad (52)$$

TABLE I
RESULTS FOR 1-D DWT: $\kappa = 0.94$, $T_r = 1.7 \times 10^5$ clks AND $zT_{\rm cm} = 5 \times 10^4$ clks. "CALCULATED" VALUES ARE FROM THE DERIVED EQUATIONS, WHEREAS THE "ACTUAL" VALUES ARE MEASURED FROM HARDWARE SIMULATION

Using proposed load schedule for no front-end					Using equal loads		Front-end case
n	q	Load fractions	T_f (clock cycles)		T_f (clock cycles)		Estimated T_f
		$\{\alpha_1, \ldots, \alpha_n\}$	Calculated	Actual	Calculated	Actual	(clock cycles)
1	1	{1.0}	1×10^6	9.64×10^5	1×10^6	9.64×10^5	9.53×10^5
2	1	{0.6, 0.4}	6.72×10^5	6.77×10^5	7.57×10^5	7.37×10^5	6.47×10^5
3	1	{0.54, 0.33, 0.13}	6.18×10^5	6.13×10^5	7.88×10^5	7.74×10^5	6.01×10^5
4	–	No solution	–	–	8.88×10^5	–	No solution

Similarly, the value of σ may be computed using

$$\sigma = \frac{\text{Number of system clock cycles for computation of given load}}{\text{Number of system clock cycles required to transfer same load}}. \quad (53)$$

The details for each individual example are presented as follows. The examples correspond to implementations of the case without front-end. Implementation of the front-end case requires a preconfigured data interface within each PU, and hence, it was not attempted. Instead, estimates of the processing time for the front-end case are provided, using the values of ρ and σ computed for the case without front-end.

1) 1-D DWT: We have chosen the (9,7) wavelet filter kernel for implementing a single-level 1-D DWT. We have designed the 1-D DWT unit based on the basic design presented in [30]. The designed PU performs in-place computation on 16-bit data samples. The data bus is taken to be 32 bits wide, whereas the configuration bus is 8 bits. The frequency of the configuration clock, system clock, and data bus are taken to be identical. From a sample simulation, the value of σ was determined using (53). The PU speed factor $\kappa = \sigma/(\sigma + 1)$ was then calculated to be 0.94. It was verified in simulation that $\kappa = 0.94$ is almost constant with different amounts of load fed to the PU.

The input data size was taken as 100 000 samples, which corresponds to $zT_{\rm cm} = 5 \times 10^4$ system clock cycles. A single DWT PU was then targeted to the Xilinx Virtex-II Pro FPGA XC2VP30. Based on the partial bitstream size, the normalized reconfiguration time ρ was then computed using (52). The computed value is $\rho = 3.4$, which gives $n^* = 3$ [see Fig. 6(b)]. The system was then simulated for $n = 1, 2, 3$. The required load fractions were computed using the analysis in Section V-A. For simulation, each PU was provided access to sufficient amount of RAM to hold its input data. The data bus was modeled as simple READ and WRITE. The hardware simulation results are presented in Table I. From the table, we observe that the values of T_f observed in simulation are close to that computed from the derived equations, and minimum T_f occurs for $n = 3$ as expected.

When equal loads are provided to the PUs, i.e., $\alpha_i = zT_{\rm cm}/n$ for $i = 1, \ldots, n$, the finish time corresponds to the time instant when p_n finishes computation. For this case, the expression for the finish time can be obtained as

$$T_f = \begin{cases} nT_r + \frac{zT_{\rm cm}}{n}\left(\frac{1}{1-\kappa}\right), & \frac{zT_{\rm cm}}{n} < T_r \\ T_r + zT_{\rm cm}\left[1 + \frac{1}{n}\left(\frac{\kappa}{1-\kappa}\right)\right], & \frac{zT_{\rm cm}}{n} \geq T_r. \end{cases} \quad (54)$$

Table I shows the simulation results when the PUs are provided with equal loads, as well as the values computed using (54). From the table, we can see that the proposed load schedule gives a lower processing time compared to equally dividing the load among the PUs. Table I also gives some estimates for the finish time for the front-end case, using $\rho = 3.4$ and $\kappa = 0.94$. As expected, the finish times are smaller for the front-end case. However, possible increase in κ due to usage of RAM port during load transfer (Section VI) has not been accounted for.

2) FIR Filter: We have used Xilinx CoreGenerator to obtain a 16th order (17-tap) low-pass FIR filter core. The filter core is based on distributed arithmetic and accepts 8-bit input data every 8 clock cycles. Each PU is designed with the FIR core and surrounding control logic for input and output data transfer. Input and output data are taken as 8-bits wide. The data bus as well as the configuration bus are taken to be 8-bits wide. The data bus is actually an interface to SRAM, and is designed to transfer each byte every three clock cycles. Our theory is applicable here since a single SRAM port is equivalent to the constraint of using a shared bus.

The input data size is again 100 000 samples, which gives $zT_{\rm cm} = 3 \times 10^5$ clock cycles. The PU is targeted to a Xilinx Virtex XCV300 FPGA, with the resulting partial bitstream size being approximately 6×10^4 bytes. The configuration clock frequency is taken to be half the system clock frequency, therefore, $T_r = 1.2 \times 10^5$ system clock cycles, which gives $\rho = 0.4$. We also have $\kappa = 0.77$, computed from sample simulations of a single PU. Using these values of ρ and κ, our analysis gives $n^* = 5$. Hardware simulations were carried out for $n = 1, \ldots, 5$, with each PU having access to as much local RAM as necessary. Hardware simulation results for the FIR filter are given in Table II. As before, the simulated values are close to the computed values. Also, the proposed load distribution is better than distributing equal load to all PUs. Again, estimated values of the finish times for the front-end case are smaller than those for the no-front-end case, assuming same values of ρ and κ.

B. Experimental Results

We now describe the experiment carried out on actual FPGA hardware. The hardware platform is the XSV-300 board from XESS Corporation [31]. The board components and connections pertinent to our experiment is depicted in Fig. 13. Access to all components on the board from the desktop personal computer (PC) is through the complex programmable logic device (CPLD). For, e.g., to transfer data between the PC and onboard

TABLE II
RESULTS FOR FIR FILTER: $\kappa = 0.77$, $T_r = 1.2 \times 10^5$ clks AND $zT_{\rm cm} = 3 \times 10^5$ clks. "CALCULATED" VALUES ARE FROM THE DERIVED EQUATIONS, WHEREAS THE "ACTUAL" VALUES ARE MEASURED FROM HARDWARE SIMULATION

Using proposed load schedule for no front-end					Using equal loads		Front-end case
n	q	Load fractions	T_f (clock cycles)		T_f (clock cycles)		Estimated T_f
		$\{ \alpha_1, \ldots, \alpha_n \}$	Calculated	Actual	Calculated	Actual	(clock cycles)
1	1	{ 1.0 }	1.42×10^6	1.43×10^6	1.42×10^6	1.43×10^6	1.12×10^6
2	2	{ 0.565, 0.435 }	8.57×10^5	8.61×10^5	9.22×10^5	9.26×10^5	6.82×10^5
3	2	{ 0.427, 0.329, 0.244 }	6.78×10^5	6.81×10^5	7.95×10^5	7.95×10^5	5.75×10^5
4	1	{ 0.388, 0.296, 0.204, 0.112 }	6.26×10^5	6.27×10^5	8.06×10^5	8.07×10^5	5.51×10^5
5	1	{ 0.384, 0.292, 0.200, 0.108, 0.016 }	6.21×10^5	6.22×10^5	8.61×10^5	8.62×10^5	No solution
6	–	No solution	–	–	9.37×10^5	–	No solution

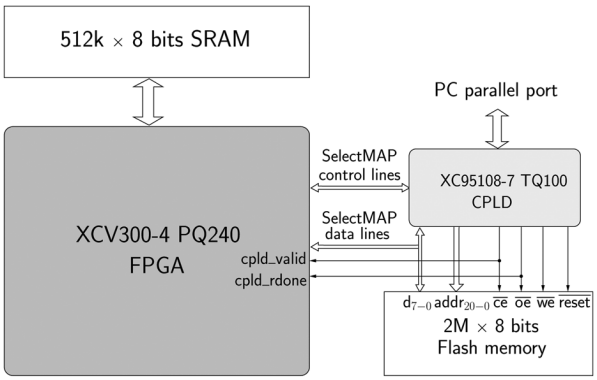

Fig. 13. XSV-300 board components and connections relevant to our experiment.

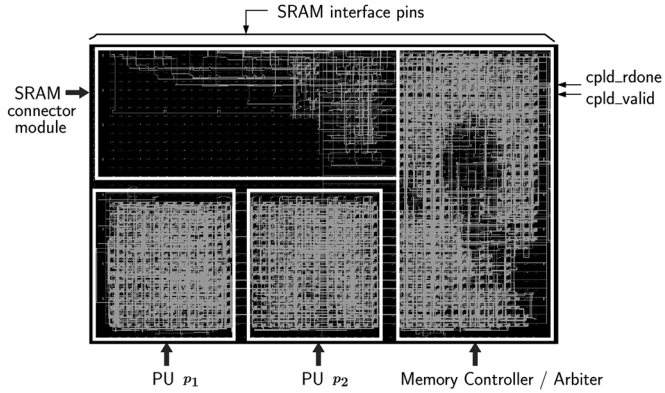

Fig. 14. Layout of FIR filter example implemented on XCV300, as seen in the FPGA_Editor Xilinx software.

SRAM, the CPLD and FPGA must be programmed with the required interfaces and control logic. Similarly, programming the Flash memory requires the appropriate logic to be programmed in the CPLD. The XSTOOLS software package is used for programming the CPLD. The XSTOOLS software is also used for programming the FPGA whenever the SRAM needs to be accessed. We have developed "C" programs to READ/WRITE SRAM and Flash memory from the PC, through the PC parallel port. The FPGA logic for accessing SRAM is based on the "PC to SRAM interface" design in [32], whereas the CPLD designs are based on examples available on the XESS website [31].

For our experiment, the configuration data required for FPGA reconfiguration is stored in the Flash memory. The configuration data constitutes of the following: 1) initial power-up configuration of the FPGA, which has the fixed controller modules as well as placeholders for the PUs and 2) partial bitstream for each PU. A state machine programmed in the CPLD carries out the required reconfiguration. Configuration is initiated as soon as appropriate control signals are received from the PC parallel port. The CPLD then configures the FPGA with the initial configuration 1). After that, the PUs are sequentially configured. Reconfiguration is done through the SelectMAP port of the FPGA. The data lines of the SelectMAP port are directly connected to the data lines of the Flash memory. The CPLD controls the SelectMAP control signals, while simultaneously issuing the appropriate address and read signals to the Flash. After configuration of every PU, the CPLD signals a pulse on the *cpld_rdone* pin of the FPGA, while asserting a logic high on *cpld_valid*.

The FIR filter example presented in Section VII-A.2 was targeted onto the XCV300 FPGA. Two PUs were implemented on the FPGA, as shown in the layout in Fig. 14. The different modules marked on the layout are explained as follows.

1) PU p_1, p_2: The FIR filter PUs.
2) Memory controller/Arbiter: This module accepts requests from the PUs for reading/writing data to SRAM, and issues the appropriate control/data signals to the SRAM. Each PU requests for data as soon as it is configured, so some arbitration is required to ensure that load transfer occurs in the required order.
3) SRAM connector module: On the XSV-300 board, the SRAM chip has its interface pins connected to almost the entire top portion of the FPGA, as indicated in Fig. 14. The SRAM connector module is required for providing access to SRAM pins that are not directly attached to the Memory controller module.

Connection between the PUs and Memory controller module, as well as between the SRAM connector and Memory controller, is through fixed, unidirectional routing lines called bus-macros [3]. In particular, connection between p_1 and the memory controller is through long bus-macros that run "over" p_2. The long bus-macros were created using the methodology outlined in [33]. These lines provide reliable connection even while p_2 is undergoing reconfiguration.

Xilinx modular design flow [34] was used for implementing all the required modules. However, for generating the partial

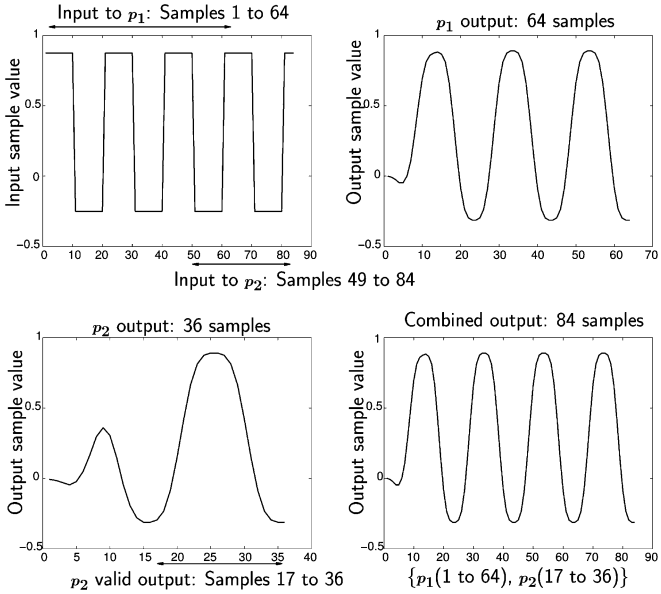

Fig. 15. Plots of input square wave and low-pass filtered output samples obtained from the experiment carried out on the XSV-300 hardware board.

bitstreams, the difference-based flow was used [3]. The difference-based flow ensures that the fixed part (in particular, the SRAM connector module) remains the same during reconfiguration of the PUs. During reconfiguration of each PU, the entire FPGA height spanning the width of the PU is reconfigured. However, since the SRAM connector module remains the same, the reconfiguration of the portion of the SRAM connector that lies above the PU occurs in a glitchless manner, so it is possible for the SRAM interface to be active even during PU reconfiguration.

The local RAM for each PU was implemented using Xilinx lookup tables (LUTs) within each PU. The maximum capacity of the local RAM turned out to be 64 bytes per PU. This presented a serious problem for testing our theory, since such a small load (at normal values of σ) gives $n^* = 1$. The execution time of each PU was artificially stretched by inserting a delay of 4096 clock cycles between processing of successive input samples. This resulted in $\sigma = 1370$, consequently increasing n^* to 3.

The number of input samples was taken as 100 bytes ($zT_{\rm cm} = 300$). In order to ensure that there is no interdependency in the computations carried out by the PUs, the last sixteen input samples fed to p_1 must also be input to p_2. Consequently, the effective input size is 84 samples. This overlap of data is indicated in Fig. 15, for the square wave input. With the input data stored in SRAM, the runtime partial reconfiguration experiment was carried out. As soon as a pulse on *cpld_rdone* signal is observed (when *cpld_valid* $= 1$), the memory controller issues a start signal to the PU that is configured. The rest of the process of load transfer and computation occurs as outlined earlier. After computation, each PU requests the memory controller/arbiter to transfer result data back to SRAM. Contents of SRAM are later read back into the PC. The output samples obtained are shown in Fig. 15. The outputs from each PU are then combined as indicated, to get the required low-pass filtered output signal.

TABLE III
EXPERIMENTAL RESULTS FOR FIR FILTER: $\sigma = 1370 \Rightarrow \kappa = 0.99927$. CONFIGURATION CLOCK FREQUENCY IS HALF THAT OF THE SYSTEM CLOCK. BASED ON BITSTREAM SIZE, T_r IS TAKEN AS 1.2 $\times 10^5$ clks FOR COMPUTING α_i. $zT_{\rm cm} = 300$ clks. CALCULATED VALUE OF T_f IS 3.86 $\times 10^5$ clks

PU	Calculated load fraction	# samples input given	Bitstream size (bytes)	Measured time ($\times 10^5$ clock cycles)	
				Start	Finish
p_1	0.65	64	59396	1.19	3.82
p_2	0.35	36	58488	2.36	3.84

Table III gives the time information recorded from the experiments. The time is recorded within the memory controller using a counter that increments every 1024 clock cycles. The counter values are written back to SRAM, which are then READ into the PC along with the output data. We observe that the start times of p_i (which is the time instant p_i is ready after configuration) is almost the same as iT_r ($i = 1, 2$), as expected. We can also see that the measured finish times are almost equal, and are very close to that obtained from theory (3.86 $\times 10^5$ clks). It may be noted that $n = 3$ cannot be implemented due to limited FPGA area. Further, for $n = 3$, $\alpha_3 = 0.04$ which corresponds to 4 samples input to p_3; this cannot be implemented since p_3 must be given at least 16 input samples, corresponding to overlapped data as mentioned earlier.

VIII. CONCLUSIONS

In this paper, we have described a methodology for mapping data-parallel applications onto reconfigurable hybrid processor architectures. We have modified the framework of DLT in order to account for reconfiguration overhead of PUs. When the reconfiguration overhead is absent, the processing time reduces with the inclusion of every additional PU. In contrast, when there is a reconfiguration overhead, we have demonstrated that there exists an upper limit on the number of PUs that can be used in the RF, beyond which an improvement in processing time cannot be obtained. We have shown this for two cases—the case when load cannot be transferred to the DRL in parallel with reconfiguration/computation and the case when parallel load transfer is possible. Algorithms for obtaining the optimum number of PUs and analytical expressions for the corresponding optimum load fractions, load transfer schedule, and processing time were derived.

Hardware simulations of two examples, viz., 1-D DWT and FIR filter, targeted to Xilinx FPGAs, were presented. The theory developed was used to obtain the optimum number of PUs (n^*) to be used in the FPGA, as well as the load fractions and data transfer schedule, based on the estimated value of reconfiguration time. Hardware simulations were performed for all values of $n \leq n^*$, to show that optimum processing time is achieved for $n = n^*$. Simulations also showed that the proposed load distribution results in smaller processing time, compared to a simple strategy of equally distributing the load to all PUs. Implementation of a hardware prototype on an XSV-300 FPGA board was then presented. It was shown that the finish time obtained on the hardware prototype was close to that obtained from theory. The practical applicability of the theory developed was, thus, demonstrated.

ACKNOWLEDGMENT

The authors would like to thank Xilinx's University Program for providing the Xilinx ISE software used in this paper. The authors would also like to thank the reviewers for their detailed comments that have helped to improve the quality of this paper.

REFERENCES

[1] A. DeHon and J. Wawrzynek, "Reconfigurable computing: What, why, and implications for design automation," in *Proc. 36th ACM/IEEE Des. Autom. Conf.*, 1999, pp. 610–615.

[2] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002.

[3] Xilinx Inc., San Jose, CA, "Two flows for partial reconfiguration: module based or difference based," Tech. Rep. XAPP290, 2004.

[4] Atmel Corp., San Jose, CA, "FPSLIC on-chip partial reconfiguration of the embedded AT40K FPGA," , 2002.

[5] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, "An optimal algorithm for minimizing run-time reconfiguration delay," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 2, pp. 237–256, May 2004.

[6] R. Maestre, F. J. Kurdahl, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh, "A framework for reconfigurable computing: Task scheduling and context management," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 9, no. 12, pp. 858–873, Dec. 2001.

[7] J. Resano, D. Verkest, D. Mozos, S. Vernalde, and F. Catthoor, "A hybrid design-time/run-time scheduling flow to minimise the reconfiguration overhead of FPGAs," *Microprocess. Microsyst.*, vol. 28, no. 5–6, pp. 291–301, Aug. 2004.

[8] J. Noguera and R. M. Badia, "Hw/sw codesign techniques for dynamically reconfigurable architectures," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 10, no. 8, pp. 399–415, Aug. 2002.

[9] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[10] K. Bondalapati and V. K. Prasanna, "Reconfigurable computing systems," *Proc. IEEE*, vol. 90, no. 7, pp. 1201–1217, Jul. 2002.

[11] C.-T. King, W.-H. Chou, and L. M. Ni, "Pipelined data-parallel algorithms: Part I—Concept and modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 4, pp. 470–485, Oct. 1990.

[12] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Considering run-time reconfiguration overhead in task graph transformations for dynamically reconfigurable architectures," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2005, pp. 273–274.

[13] ——, "PARLGRAN: Parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures," in *Proc. ACM/IEEE Asia-South Pacific Des. Autom. Conf.*, 2006, pp. 491–496.

[14] V. Bharadwaj, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Comput.*, vol. 6, no. 1, pp. 7–17, Jan. 2003.

[15] D. Benitez, "Performance of reconfigurable architectures for image-processing applications," *J. Syst. Arch.*, vol. 49, no. 4–6, pp. 193–210, Sep. 2003.

[16] K. N. Vikram and V. Vasudevan, "Hardware-software co-simulation of bus-based reconfigurable systems," *Microprocess. Microsyst.*, vol. 29, no. 4, pp. 133–144, May 2005.

[17] C. Lee, Y.-F. Wang, and T. Yang, "Global optimization for mapping parallel image processing tasks on distributed memory machines," *J. Parallel Distrib. Comput.*, vol. 45, no. 1, pp. 29–45, Aug. 1997.

[18] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "Increasing the flexibility in FPGA-based reconfigurable platforms: The Erlangen slot machine," in *Proc. IEEE Conf. Field-Program. Technol. (FPT)*, 2005, pp. 37–42.

[19] D. Robinson and P. Lysaght, "Modelling and synthesis of configuration controllers for dynamically reconfigurable logic systems using the DCS CAD framework," in *Proc. 9th Int. Conf. Field Program. Logic Appl. (FPL), Lecture Notes Comput. Sci. (LNCS)*, 1999, pp. 41–50.

[20] Y.-C. Cheng and T. G. Robertazzi, "Distributed computation with communication delay," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 24, no. 6, pp. 700–712, Nov. 1988.

[21] V. Bharadwaj, X. Li, and C. C. Ko, "Efficient partitioning and scheduling of computer vision and image processing data on bus networks using divisible load analysis," *Image Vision Comput.*, vol. 18, no. 11, pp. 919–938, Aug. 2000.

[22] V. Bharadwaj, H. Li, and T. Radhakrishnan, "Scheduling divisible loads in bus networks with arbitrary processor release times," *Comput. Math. Appl.*, vol. 32, no. 7, pp. 57–77, Oct. 1996.

[23] J. Sohn and T. G. Robertazzi, "Optimal divisible job load sharing for bus networks," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 32, no. 1, pp. 34–40, Jan. 1996.

[24] T. G. Robertazzi, "Ten reasons to use divisible load theory," *IEEE Comput.*, vol. 36, no. 5, pp. 63–68, May 2003.

[25] V. Bharadwaj and G. Barlas, "Scheduling divisible loads with processor release times and finite size buffer capacity constraints in bus networks," *Cluster Comput.*, vol. 6, no. 1, pp. 63–74, Jan. 2003.

[26] G. D. Barlas, "Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 5, pp. 429–441, May 1998.

[27] K. N. Vikram and V. Vasudevan, "Scheduling divisible loads on partially reconfigurable hardware," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2006.

[28] M. J. Wirthlin and B. L. Hutchings, "Improving functional density through run-time constant propagation," in *Proc. 5th Int. Symp. FPGAs*, 1997, pp. 86–92.

[29] *"Virtex-II Pro and Virtex-II Pro X FPGA User Guide,"* Xilinx Inc., San Jose, CA, 2005.

[30] D. S. Taubman and M. W. Marcellin*, JPEG2000: Image Compression Fundamentals, Standards and Practice*. Dordrecht, The Netherlands: Kluwer, 2002, ch. 17.

[31] Xess Corp., Apex, NC, "X engineering software systems (XESS) Corp.," 2006.

[32] Univ. Queensland, Brisbane, Australia, "VHDL XSV board interface projects," 2006.

[33] J. Thorvinger, "Dynamic partial reconfiguration of an FPGA for computational hardware support," M.S. thesis, Dept. Electrosci., Lund Inst. Technol., Lund, Sweden, 2004.

[34] *"Xilinx Development System Reference Guide,"* Xilinx Inc., San Jose, CA, 2003.

Krishna N. Vikram (S'97–M' 06) received the B.E. degree (with distinction) in electronics and communication engineering from Sri Jayachamarajendra College of Engineering, Mysore, India, in 2000. He has submitted the Ph.D. dissertation in electrical engineering at the Indian Institute of Technology Madras, Chennai, India.

He is currently a Member of the Technical Staff in the Embedded Systems Group at Siemens Corporate Technology, Bangalore, India. His research interests include reconfigurable computing, computer architecture, and image compression.

Mr. Vikram is a member of the IEEE Computer Society.

Vinita Vasudevan (M' 96) received the B.Tech. degree in engineering physics and the Ph.D. in electrical engineering from Indian Institute of Technology Bombay, Mumbai, India, in 1986 and 1993, respectively, and the M.S. degree in electrical engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1988.

She is currently a Professor in the Electrical Engineering Department of Indian Institute of Technology Madras, Chennai, India. Her research interests include design and computer-aided design (CAD) of digital and mixed signal circuits.