Reconfiguration and Communication-Aware Task Scheduling for High-Performance Reconfigurable Computing

MIAOQING HUANG, VIKRAM K. NARAYANA, HARALD SIMMLER,
OLIVIER SERRES, and TAREK EL-GHAZAWI
NSF Center for High-Performance Reconfigurable Computing (CHREC),
The George Washington University

High-performance reconfigurable computing involves acceleration of significant portions of an application using reconfigurable hardware. When the hardware tasks of an application cannot simultaneously fit in an FPGA, the task graph needs to be partitioned and scheduled into multiple FPGA configurations, in a way that minimizes the total execution time. This article proposes the Reduced Data Movement Scheduling (RDMS) algorithm that aims to improve the overall performance of hardware tasks by taking into account the reconfiguration time, data dependency between tasks, intertask communication as well as task resource utilization. The proposed algorithm uses the dynamic programming method. A mathematical analysis of the algorithm shows that the execution time would at most exceed the optimal solution by a factor of around 1.6, in the worst-case. Simulations on randomly generated task graphs indicate that RDMS algorithm can reduce interconfiguration communication time by 11% and 44% respectively, compared with two other approaches that consider data dependency and hardware resource utilization only. The practicality, as well as efficiency of the proposed algorithm over other approaches, is demonstrated by simulating a task graph from a real-life application - N-body simulation - along with constraints for bandwidth and FPGA parameters from existing high-performance reconfigurable computers. Experiments on SRC-6 are carried out to validate the approach.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*adaptable architectures*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Hardware task scheduling, reconfigurable computing

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. IIP-0706352.

Authors' address: M. Huang, V. K. Narayana, H. Simmler, O. Serres, and T. El-Ghazawi, Department of Electrical and Computer Engineering, The George Washington University, Ashburn, VA 20147; email: mqhuang@gwmail.gwu.edu; vikram@gwu.edu; harald.simmler@gmail.com; serres@gwmail.gwu.edu; tarek@gwu.edu.

ACM Reference Format:
Huang, M., Narayana, V. K., Simmler, H., Serres, O., and El-Ghazawi, T. 2010. Reconfiguration and communication-aware task scheduling for high-performance reconfigurable computing. ACM Trans. Reconfig. Techn. Syst. 3, 4, Article 20 (November 2010), 25 pages. DOI = 10.1145/1862648.1862650. http://doi.acm.org/10.1145/1862648.1862650.

1. INTRODUCTION

High-performance reconfigurable computers (HPRCs) are traditional HPCs extended with co-processors based on reconfigurable hardware like FPGAs. These enhanced systems are capable of providing significant performance improvement for scientific and engineering applications. Well known HPRC systems such as Cray XD1, the SRC-6, and SGI RC100 use a high-speed interconnect, shown in Figure 1, to connect the general-purpose μP to the co-processor.

Early attempts to improve the efficiency of FPGA co-processors used task placement algorithms in combination with partial run-time reconfiguration or dynamic reconfiguration to reduce the configuration overhead of the FPGA system [Bazargan et al. 2000; Brebner and Diessel 2001; Compton et al. 2002; Diessel et al. 2000; Fekete et al. 2001; Handa and Vemuri 2004; Walder and Platzner 2002; Walder et al. 2003]. Most of these map individual SW functions onto the FPGA to accelerate the whole application; however, the optimization strategies for task placement did not consider data communication or data dependencies among hardware tasks.

We believe that an optimized hardware task scheduling algorithm has to take task dependencies, data communication, task resource utilization and system parameters into account to fully exploit the performance of an HPRC system. This article presents an automated hardware task scheduling algorithm that is able to map simple and advanced directed acyclic graphs (DAG) in an optimized way onto HPRC systems. More precisely, it is able to split a DAG onto multiple FPGA configurations optimizing the overall execution of the hardware part of the user application, after the HW/SW partition has been done. The optimization is achieved by minimizing the number of FPGA configurations and minimizing the interconfiguration communication overhead between the FPGA device and external memory by using a data dependency analysis of the given DAG. Maximum concurrency in hardware processing is ensured by considering pipeline chaining of hardware tasks within the same FPGA configuration.

The remaining text is organized as follows: Section 2 introduces the related work. Section 3 presents the hardware task scheduling algorithm, followed by a mathematical analysis of the algorithm. Section 4 focuses on the results obtained for various graphs by using the proposed algorithm. The limitations of the proposed approach are also outlined. Finally, Section 5 concludes this article.

2. RELATED WORK

Early work on hardware task placement for reconfigurable hardware focused on reducing configuration overhead and therefore improving the efficiency of

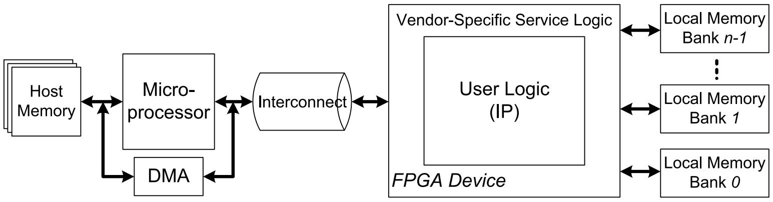


Fig. 1. The general architecture of a high-performance reconfigurable computer.

reconfigurable devices. In Fekete et al. [2001], an offline 3D module placement approach for partially reconfigurable devices was presented. Efficient data structures and algorithms for fast online task placements and simulation experiments for variants of first fit, best fit and bottom left bin-packing algorithms are presented in Bazargan et al. [2000]. In Diessel et al. [2000], the fragmentation problem on partially reconfigurable FPGA devices was addressed. Task rearrangements were performed by techniques denoted as local repacking and ordered compaction. In Walder and Platzner [2002], nonrectangular tasks were explored such that a given fragmentation metric was minimized. Furthermore, a task's shape may be changed in order to facilitate task placement. In Brebner and Diessel [2001], a column-oriented one-dimensional task placement problem was discussed. Task relocations and transformations to reduce fragmentation were proposed in Compton et al. [2002] based on a proposed FPGA architecture that supported efficient row-wise relocation. In Walder et al. [2003], three improved partitioning algorithms based on Bazargan et al. [2000] were presented. In Handa and Vemuri [2004], a fast algorithm for finding empty rectangles in partially reconfigurable FPGA device was presented. However, these algorithms only focused on utilizing the FPGA device efficiently through optimal task placement techniques and did not address the performance optimization among tasks in a given application, which is the main concern of the users of an HPRC system.

More recently, HW/SW co-design algorithms for RC systems have emerged, bringing the ease of use urgently needed in the RC domain. In Wiangtong et al. [2003], a HW/SW co-design model comprised of a single μP and an array of hardware processing elements (PEs implemented on FPGAs) was presented. Small tasks in a user program were dynamically assigned onto PEs. However, even if a single PE could accommodate multiple tasks, the tasks were executed in a sequential way. In Saha [2007], an automatic HW/SW co-design approach on RCs was proposed. The proposed ReCoS algorithm partitioned a program into hardware tasks and software tasks and co-scheduled the tasks on their respective PEs. Although the ReCoS algorithm places multiple hardware tasks into the same configuration, each task is treated as a stand-alone hardware module. No data paths are defined between the tasks even if they reside in the same configuration.

3. REDUCED DATA MOVEMENT SCHEDULING ALGORITHM

Given the hardware task graph of one application and the hardware implementation of each task, the RDMS algorithm schedules the hardware tasks

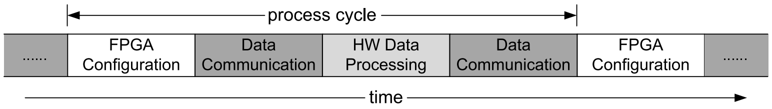


Fig. 2. The basic execution model of hardware tasks on FPGA device.

into a series of FPGA configurations in a way that minimizes the total hardware execution time, by restricting the number of FPGA configurations and the transfer of intermediate data between FPGA and host memory [Huang et al. 2009]. In order to achieve these desired objectives, the RDMS algorithm takes three factors into account during the scheduling process, namely, (a) the task data dependencies, (b) the hardware resource utilization of each task, and (c) the intertask data communication.

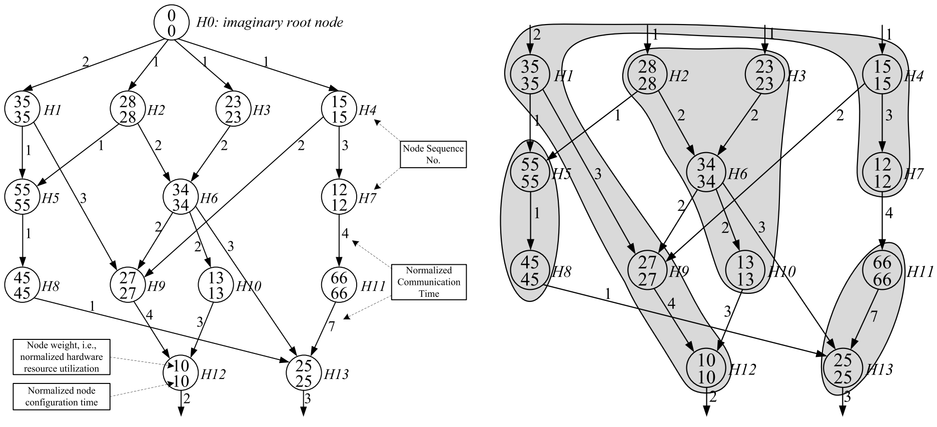
RDMS assumes the execution model shown in Figure 2, for hardware tasks running on the FPGA co-processor. After the FPGA device is configured and before hardware tasks start their computation, raw data are transferred from the μP or external memory to the FPGA device. After hardware tasks finish their computation, the processed data are transferred back from the FPGA device to external memory, before the next FPGA configuration is loaded into the device.[1] The data from external memory are then again transferred to the FPGA, in order to feed tasks that have a dependency on other tasks that finished computation in the preceding configuration(s).

In addition to the assumption of the execution model shown in Figure 2, the RDMS algorithm uses the following assumptions about the underlying system:

(1) Implementation of all tasks are pipelined, allowing concurrent execution of tasks within a configuration. (If the tasks are not inherently pipelined, FIFO buffers at the input and output of the tasks can be used, to create a similar effect.)
(2) The data processed by each FPGA configuration is large, allowing the initial pipeline latency to be neglected.
(3) FPGA supports only full configuration (partial reconfiguration is not allowed).

The processing time of each task is the ratio of input data volume for the task and data consumption throughput of the task. From Assumptions 1 and 2, it follows that the processing time of a particular FPGA configuration simply equals the processing time of the slowest task in the configuration. The time taken for data transfer between the host memory and FPGA is computed based on the interconnect bandwidth and the data volume transfer to/from the task edges. The execution time T_{hwe} for the entire computation consisting of n configurations is, therefore, the sum of (1) $n \cdot T_R$, in which T_R is the configuration time of the whole FPGA device, (2) the processing time of slowest task in each configuration, and (3) host memory data transfer time for each configuration.

[1] In most cases the data communication and data processing are overlapped due to the high throughput of the hardware design. We intentionally distinguish these two times in this work for the sake of simplicity.


H0: imaginary root node
H1
H2
H3
H4
H5
H6
H7
H8
H9
H10
H11
H12
H13
Node Sequence No.
Normalized Communication Time
Node weight, i.e., normalized hardware resource utilization
Normalized node configuration time

(a) An Example DAG Consisting of 13 Nodes

(b) Scheduling Result of the Example DAG Using RDMS

Fig. 3. The example graph.

As mentioned earlier, configuration time and data communication time are pure overheads for using RCs and therefore both should be reduced to a minimum. Two strategies are applied to achieve this objective.

(1) Execute as many tasks as possible in one FPGA configuration to minimize the amount of configurations. This will reduce the configuration overhead as well as the communication overhead between tasks.
(2) Group communicating tasks into the same configuration to minimize the data communication time and to maximize the performance achieved through pipelining and concurrent task execution.

Given the DAG of an application, the overall amount of communication volume among the tasks is fixed. When the tasks of the DAG are scheduled into multiple FPGA configurations, the communication can be categorized into two types - the internal communication inside FPGA configurations and the interconfiguration communication. Because the sum of these two categories of communication is fixed, we expect that minimization of the interconfiguration communication is equivalent to maximizing the internal communication inside FPGA configurations.

We identified the required task scheduling as a dependent knapsack problem, which is a constrained case of the knapsack problem [Kellerer et al. 2004]. For the single-objective independent knapsack problem, there exists an exact solution that can be computed using a dynamic programming approach [Kleinberg and Tardos 2005]. In the traditional knapsack problem, each object is associated with two parameters, the weight and the profit. The weight can be seen as a penalty that has to be paid for selecting an object and the profit is a reward. The goal of the knapsack problem is to maximize the profit subject to a maximum total weight.

In our task scheduling case, we have a dependent knapsack problem where we have to take care of the task dependencies and make sure that all parent tasks have been scheduled before scheduling the corresponding child task. This ensures that the input data of every task is either obtained from host memory as a result of a previously executed task, or through data forwarding from another task within the same configuration.

The objects in the task scheduling knapsack problem are the hardware tasks given in the DAG. The weight of each object is the hardware resource utilization of the corresponding task. Because the objective of the knapsack problem is to maximize the profit, we have to define the profit according to our two goals: maximize the amount of tasks and minimize the interconfiguration communication between tasks.[2] Both profits are transformed into a common unit, *time*, so that a single unified metric for the profit can be used.

The number of tasks assigned to each configuration is dependent on the allocated resources for each task. Therefore, we define one part of the profit to be the configuration time of each task. This "configuration time" is the fraction of the full configuration time of the FPGA device and it is proportional to the amount of used resources divided by the total available resource. Note that we allow only full configuration of the FPGA, so the "task configuration time" is merely a mathematical construct used for defining the profit. The "task configuration time" part of the profit, P_{cfg}, is calculated for each task using (1).

$$P_{cfg} = \frac{S_{req}}{S_{full}} \times T_R, \tag{1}$$

where S_{req} is the resource utilization of the task, S_{full} is the full resource of the target FPGA device, and T_R is the configuration time of the whole FPGA device. Note that although Eq. 1 may be identified to be equivalent to the weight, it is a necessary component of the profit, because we are attempting to maximize the resource utilization of the FPGA. This is similar to the subset-sum problem, which has a profit entirely identical to the weight.

The second part of the profit is defined as the time corresponding to the intertask communication that is saved due to the fact that communicating tasks are executed in the same configuration. This time, the saved intertask communication time, P_{com} is obtained as a ratio of intertask communication data amount and the available I/O bandwidth, multiplied by 2. The formula is given in (2). We count the data amount twice because we save the time to transfer data out of the FPGA and back into the FPGA device again.

$$P_{com} = \frac{V_{com}}{B_{IO}} \times 2, \tag{2}$$

where V_{com} is the intertask communication volume, and B_{IO} is the I/O bandwidth between FPGA and host memory or μP.

[2] Hardware processing time is not considered in the objective function for the algorithm, since the communications and reconfigurable cost are greater than that time in most cases.

The example task graph in Figure 3(a) shows the resource utilization of each task normalized to 100 and the corresponding "task configuration time" calculated with respect to the full configuration time of 100 unitless time. The intertask communication time is shown next to the task graph edges. Note that the intertask communication time shown is actually platform dependent, since it depends on the I/O bandwidth; yet, the communication time (instead of communication volume) is shown associated with the edges of the graph, for ease of understanding. Similar comments hold for the task resource utilization shown in the figure.

The scheduling result of the example graph is illustrated in Figure 3(b). The resulting schedule shows a saved intertask communication time among task H2, H3, H6, and H10 of 12.

3.1 Mathematical Description of the RDMS Algorithm

The RDMS algorithm repeatedly applies the dependent knapsack algorithm to schedule the tasks into independent configurations. The objective of the dependent knapsack problem is to find a subset of tasks that has a maximum combined task configuration time and a maximum saved intertask communication time.

The dependent knapsack problem is described formally as follows. There is a DAG of n nodes of positive weight and one root node of no weight. The nodes in the DAG are numbered in an increasing order from top to bottom and from left to right at the same level. Each node is associated with a weight w_i and a profit p_i, $0 \leq i \leq n$. The w_i values correspond to the percentage of resource utilization of each task, whereas the profit values p_i associated with each task are computed using (1). The profit derived from savings in intertask communication is captured by a matrix E[3]; this matrix describes the links among the nodes and $E(i, j)$ is the profit associated with the edge between node i and j, computed using (2).

To solve the dependent knapsack problem, it is required to select a subset of nodes $S \subseteq \{1, \ldots, n\}$ so that $\sum_{i \in S} w_i \leq W$ (the given upper bound) and, subject to two restrictions, namely:

—The profit of the selected subset S, given by $\sum_{i \in S} p_i + interc(S)$, is as large as possible, where $interc(S)$ is the profit due to the saved intertask communication, as described in Eq. (3):[4]

$$interc(S) = \sum_{i,j \in S; i,j \neq 0} E(i, j) \tag{3}$$

—For all nodes $i \in S$, their parent nodes also belong to S, so that the concurrent execution of all tasks and the internal data forwarding inside a single FPGA configuration becomes possible.

[3] The communication times between imaginary root node 0 and other nodes are not considered during the calculation since the corresponding data are input data to the whole graph and have to be transferred anyway. For the same reason, the final output of the DAG is not considered in the calculation.

[4] If there is no edge from i to j, $E(i, j)$=0.

Algorithm 1: Algorithm for Dependent Knapsack Problem

```
Input: Empty array $P_{OPT}[0..n, 0..W]$, $S[0..n, 0..W]$ and a DAG of
       corresponding $n$ nodes plus an imaginary root node 0. The nodes in
       the DAG are numbered in an increasing order from top to bottom
       and from left to right at the same level. Each node is associated
       with one weight $w_i$ and one profit $p_i$. Matrix $E[0..n, 0..n]$ describes
       the link among the nodes and the associated profit of each link in
       the DAG.
Output: $P_{OPT}(n, W)$ is the optimal profit combination and $S(n, W)$
        contains the corresponding nodes.
Initialize $P_{OPT}[0, w] = S[0, w] = 0$ for $0 \le w \le W$, and
$P_{OPT}[i, 0] = S[i, 0] = 0$ for $0 \le i \le n$;
for $i = 1$ to $n$ do
    for $w = 1$ to $W$ do
        Use the recurrence Equations (4)–(5) to compute $P_{OPT}(i, w)$ and fill
        in $S(i, w)$;
```

We use the dynamic programming approach to solve the dependent knapsack problem. Dynamic programming relies on solving smaller subproblems in order to eventually solve the original problem. Recursive relations are used in order to achieve the desired objective. Recursion is defined on the task subset S based on variable index i and the weight (resource) constraint w; $S(i, w)$ denotes the optimal subset of tasks chosen from $\{1, \ldots, i\}$ under the weight constraint w. The objective of the dependent knapsack problem is to determine $S(n, W)$, where W is the full FPGA resource constraint. Corresponding to $S(i, w)$, we also have the maximal profit values $P_{OPT}(i, w)$. Since the choice of $S(i, w)$ is essentially based on the tasks that result in maximal profit, we first define recursions on $P_{OPT}(i, w)$. Using the notations w_i and p_i, the weight and profit of task i as defined in the beginning of Section 3.1, the recurrence relation for the maximal profit value P_{OPT} is:

$$P_{OPT}(i, w) = \begin{cases} P_{OPT}(i-1, w) & \text{if } w_i > w, \\ \max(P_{OPT}(i-1, w),\ [P_{OPT}(i-1, w') + p_i + \mathit{inter}(i)]) & \text{Otherwise.} \end{cases} \tag{4}$$

where $\mathit{inter}(i)$ is a measure of the savings in intertask communication based on the inclusion of task i to the subset $S(i-1, w')$, to be quantified later. The variable w' will also be formally defined later, but for this initial discussion let us consider it to be $w' = (w - w_i)$. The principle behind Eq. (4) is to obtain P_{OPT} (and therefore S) for tasks $\{1, \ldots, i\}$, based on the result for tasks $\{1, \ldots, i-1\}$. The basic idea is that if task i has a weight w_i smaller than the weight constraint w, then its inclusion in S implies that the other tasks chosen from $\{1, \ldots, i-1\}$ comprise an optimal subset under the weight constraint $w' = (w - w_i)$.

Corresponding to recurrence (4), the recurrence relation for task subset S may be easily written down. From the first line of (4), if $w_i > w$ it follows that $S(i, w) = S(i-1, w)$, which means that task i is not included in $S(i, w)$. This could also happen for line 2 of (4), when the profit $P_{OPT}(i-1, w)$ is the larger term. The only case when task i is included in $S(i, w)$ is when the overall profit due to its inclusion will be larger than $P_{OPT}(i-1, w)$. These points are captured succinctly as follows:

$$S(i, w) = \begin{cases} S(i-1, w) & \text{if } w_i > w \text{ or } P_{OPT}(i-1, w) > [P_{OPT}(i-1, w') + p_i + inter(i)]\ , \\ \{S(i-1, w'), i\} & \text{Otherwise.} \end{cases} \tag{5}$$

The expression to compute $inter(i)$, described earlier as the savings in intertask communication due to the inclusion of task i, is as follows:

$$inter(i) = \sum_{j \in S(i-1, w'), j \neq 0} E(j, i) \tag{6}$$

We have referred to w' earlier as the weight constraint on the remaining tasks chosen from $\{1, \ldots, i-1\}$, provided task i is included in $S(i, w)$. Since task i is being considered for inclusion, the weight constraint on the remaining tasks would be $w - w_i$, which is accurate in the absence of precedence constraints among tasks. For our case, however, it might turn out that $S(i-1, w - w_i)$ does not contain all of the parents of node i. In that case, starting from $w - w_i$, the constraint w' must be successively reduced until $S(i-1, w')$ contains all the parents nodes of task i. That is,

$$w' = max\{x | x \leq w - w_i \text{ and all of node } i\text{'s parent nodes belong to } S(i-1, x)\} \tag{7}$$

Note that the weight is a continuous variable, so it is discretized in steps of 1% of maximum FPGA resource constraint W, in order to determine w' using (7). Based on equations (4)-(7), the overall dynamic programming solution for the dependent knapsack problem is given in Algorithm 1. As mentioned earlier, the output of the algorithm is the required subset of tasks $S(n, W)$ with the corresponding maximal profit $P_{OPT}(n, W)$. Algorithm 1 is used within the RDMS algorithm, which is described next.

The RDMS algorithm begins with Algorithm 1 to schedule a subset of tasks into the first configuration. Once this is done, the scheduled tasks (or nodes) are removed from the DAG. Edges that connect the existing nodes to the deleted nodes are rearranged so that they are connected to the root node instead. Algorithm 1 is then applied again to determine the second configuration. The process is repeated until all tasks in the DAG are scheduled. These steps are formally described in Algorithm 2.

The example DAG shown in Figure 3(a) is scheduled using RDMS, and the resulting configurations are shown in Figure 3(b). The scheduled result consists of four FPGA configurations, {H2,H3,H6,H10}, {H1,H9,H12,H4,H7}, {H5,H8} and {H11,H13}. In other words, the FPGA will be reconfigured

Algorithm 2: RDMS Algorithm

```
Input: A DAG of $n + 1$ nodes, representing $n$ hardware tasks and an
       imaginary root node 0. Each node $i$ has a weight $w_i$ and a profit $p_i$.
       Matrix $E[0..n, 0..n]$ describes the link among the nodes and the
       associated profit of each link in the DAG.
Output: A sequence of disjoint subsets $\{S_1, \ldots, S_j\}$ satisfying
        $\sum_{i \in S_k} w_i \leq W$, $k = 1, \ldots, j$.
Let $\mathcal{O}$ denote the set of current remaining items and initialize $\mathcal{O} = \{1, \ldots, n\}$, let $k$=1;
while $\mathcal{O}$ is not empty do
    Apply Algorithm 1 on $\mathcal{O}$ and DAG to find the subset $S_k$;
    Remove the items in the subset $S_k$ from $\mathcal{O}$ and corresponding nodes
    from the DAG;
    Connect those nodes whose parent nodes have been taken to the root
    node directly;
    $k = k + 1$;
```

four times. Each configuration performs part of the computation on one piece of data.

By taking two system parameters, the configuration time of hardware tasks and the intertask communication time into account, the RDMS algorithm can address the characteristics of different platforms automatically. On an RC system with a long configuration time, the configuration overhead of hardware tasks will play a more important role in the scheduling process. On a different RC system that can switch from one configuration to the other instantly, but instead has a comparatively slow interconnect, the algorithm will favor packing of tasks into the same configuration if their intertask data transfer is large.

3.2 Mathematical Analysis of the RDMS Algorithm

We carry out a formal analysis of the proposed RDMS algorithm, in order to determine its time complexity, space requirements, as well as the worst-case performance bounds.

3.2.1 *Time Complexity.* In order to compute each $P_{OPT}(i, w)$ in the Algorithm for Dependent Knapsack Problem, it traces back at most $w - 1$ steps, see Eq. (7). (The number of steps is based on the fact that the weight is expressed as a percentage of resource utilization, and because it is discretized in steps of 1% of maximum resource constraint, w is effectively an integer between 1 and 100) Furthermore, it may take $n - 1$ steps to check the solvability of each node under the worst-case scenario. Therefore, it takes $O(nw)$ time to compute each $P_{OPT}(i, w)$. The time to compute whole row is $O(n+n\times 2+\cdots+n\times(W-1)+n\times W)$, which is $O(nW^2)$. The time complexity of the whole algorithm, i.e., n rows, is $O(n^2W^2)$.

In order to compute the time complexity of RDMS Algorithm, consider the worst-case scenario in which only one node is taken every iteration. Because the time complexity of Algorithm 1 is $O(n^2W^2)$ given n items and an upper bound W, the time for calculating the sequence of subsets is $O(n^2W^2 + (n-1)^2W^2 + \cdots + 2^2W^2 + W^2)$, which is $O(n^3W^2)$. Since a breadth-first search technique is adopted to reorganize the directed rooted graph (Line 5 of Algorithm 2), it takes $O(n + E)$ time given n nodes and E edges among them. The maximum possible value of E is n^2; hence, the time complexity of the reorganization is $O(n + n^2)$, which is $O(n^2)$. The time complexity of the reorganization for the whole process is $O(n^2 + (n-1)^2 + \cdots + 2^2 + 1)$, which is $O(n^3)$. Altogether, the time complexity of Algorithm 2 is $O(n^3W^2)$ under the worst-case scenario.

Pisinger [1999] introduced an algorithm that is able to reduce the time complexity of the independent knapsack problem by lowering W to $\overline{w}$, where $\overline{w}$ is the largest weight of an item in the instance. Further research is required to study the feasibility of similar optimizations for the RDMS algorithm.

3.2.2 *Space Requirement.* Since the RDMS algorithm runs the Algorithm for the Dependent Knapsack Problem multiple times, the space requirement of the former one is same as for the latter one.

For the Algorithm for the Dependent Knapsack Problem, the space requirement to save the information of n items will be $O(n^2)$ at the most. Because the computation of both $P_{OPT}(i, w)$ and $S(i, w)$ relies only on the variables from the previous row, only two rows of each of them are required during the processing. Therefore, the P_{OPT} array will require $O(W)$ space because every $P_{OPT}(i, w)$ is a scalar. Contrastingly, every $S_{OPT}(i, w)$ is a vector that is required to store at most i scalars due to the fact that Algorithm 1 is considering item 1 through item i at any given moment. Therefore, row i of S_{OPT} array will take at most $O(iW)$ space. The two largest rows will take $O((n-1)W + nW)$ space, which is $O(nW)$. Putting all things together, both Algorithms 1 & 2 require $O(n(n + W))$ space at the maximum.

3.2.3 *Worst-Case Performance Bound.* Although results to be presented later in Section 4 show that RDMS performs well, it is desirable to know the worst case performance bound of the algorithm. The performance bound of a heuristic algorithm is generally given in the form of a ratio of the worst case performance to the optimal performance, R^∞. In our case, the performance metric is total execution time, T_{hwe}. We therefore have to obtain

$$R^\infty = \frac{T_{hwe}^{RDMS}}{T_{hwe}^{opt}} \text{ (worst case)} \tag{8}$$

where T_{hwe}^{opt} is the total execution time for the optimal solution. In Operations Research literature, a range of values is generally given for the worst case performance ratio, that is, $R^\infty \in [R_L, R_U]$. The lower limit, R_L, is generally obtained by constructing an extreme example (or a family of examples) for which

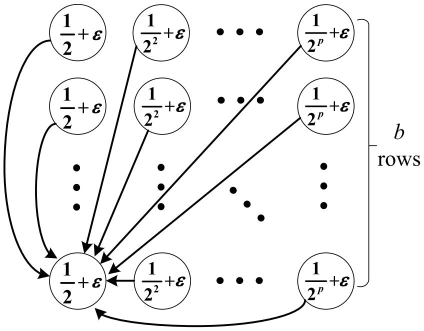

$\frac{1}{2}+\varepsilon$
$\frac{1}{2^2}+\varepsilon$
$\frac{1}{2^p}+\varepsilon$
b
rows

Fig. 4. Task graph used for obtaining lower limit on worst-case performance ratio.

the optimal solution is known and the given algorithm performs very badly. We will follow this approach to obtain the lower limit. However, derivation of the upper limit R_U is nontrivial and requires advanced mathematics. Currently, we provide some arguments to conjecture on the value of the upper limit, as described later.

One extreme example of a task graph is shown in Figure 4. For convenience, the task nodes are shown arranged as a rectangular array. The value within each task is the resource consumption specified as a fraction of the total FPGA area. There are p sets of tasks corresponding to the p columns in Figure 4; each set consists of b tasks of size $\left(\frac{1}{2^k}+\varepsilon\right)$ each. The value of k varies from $1,\ldots,p$, corresponding to the p sets, and ε is a small value that satisfies $\varepsilon \le \frac{1}{2^{2p}}$. The value of b is chosen to be an integer multiple of (2^k-1) for $k=1,\ldots,p$, that is, $b = a \times \prod_{k=1}^{p}(2^k-1)$ for some positive integer a. The reason for this choice will become apparent in the explanation that follows. The example considered here is based on a numerical example used by Caprara and Pferschy [2004].

All tasks shown in Figure 4 perform computations followed by data transfer to the task located in the lower left corner. We assume that all tasks have the same processing time, T_{task}. The data transfer rate is also assumed to be the same for all edges. T_{comm} is the data transfer time for an edge, if an off-chip data transfer is required for that edge. Also, the reconfiguration time for the entire FPGA is T_R.

The graph has to be partitioned into multiple configurations since all tasks clearly cannot fit in the FPGA simultaneously. For a given k, (2^k-1) tasks of size $\left(\frac{1}{2^k}+\varepsilon\right)$ can fit in a single configuration. If tasks from the same set (fixed k) are chosen for each configuration, the choice of value of b specified earlier allows an integer number of configurations per set.

Based on the assumptions listed in Section 3, the execution time of each configuration will be T_{task}, irrespective of the number of tasks in the configuration. Since the data volume over each edge is the same, an optimal solution is obtained if the number of configurations is minimized. This corresponds to the one-dimensional bin-packing problem (BPP) found in the literature [Coffman, Jr. et al. 1996]. The optimal solution is to pack p tasks, one from each set, into one bin (or configuration). The p tasks will fit in the configuration due to the fact that the definition of ε guarantees $\sum_{k=1}^{p}\left(\frac{1}{2^k}+\varepsilon\right) \leq 1$. The optimal solution therefore consists of b configurations, with the tasks in each row in Figure 4 constituting a configuration. The bottom row should naturally be configured last, due to the data dependency. The total execution time is the sum of bT_R, bT_{task} and the total data transfer time. Since the graph has $(bp-1)$ edges, and off-chip data transfer is not required for the $(p-1)$ edges present in the last configuration, the total data transfer time is $2T_{comm}[(bp-1)-(p-1)]$. The factor 2 is, as explained earlier, due to data transfer occurring twice for each edge, once from FPGA to external memory and then again from external memory back to the FPGA. The optimal execution time is therefore,

$$T_{hwe}^{opt} = b\,[T_R + T_{task}] + 2[p(b-1)]T_{comm}. \tag{9}$$

When RDMS schedules the same graph, it will pack the tasks differently. RDMS attempts to maximize the FPGA resource utilization for the first configuration, before proceeding with the second configuration, and so on. It is clear that RDMS will therefore choose tasks from the last set ($k = p$) for the first few configurations, before proceeding with the other sets, in decreasing order of the value of k. In other words, RDMS starts with the tasks in the last column on the right side, and proceeds toward the left in Figure 4. As mentioned earlier, only (2^k-1) tasks from each set can fit within a configuration. Since there are b tasks in each set, RDMS will give $\frac{b}{2^k-1}$ configurations for each set. The total number of configurations is therefore $N = \sum_{k=1}^{p}\frac{b}{2^k-1}$. The final configuration consists of a single task, the task present in the lower left corner of Figure 4. The total execution time is the sum of NT_R, NT_{task} and the total data transfer time. Since the graph has $(bp-1)$ edges and off-chip data transfer is required for all the edges, the total data transfer time is $2T_{comm}(bp-1)$. The total execution time using RDMS algorithm is therefore,

$$T_{hwe}^{RDMS} = b\left(\sum_{k=1}^{p}\frac{1}{2^k-1}\right)[T_R + T_{task}] + 2(pb-1)T_{comm}. \tag{10}$$

The ratio of the execution time from (10) and (9) is

$$R = \frac{T_{hwe}^{RDMS}}{T_{hwe}^{opt}} = \frac{A+C}{B+D},$$

where $A = N[T_R + T_{task}]$, $C = 2(pb-1)T_{comm}$, $B = b\,[T_R + T_{task}]$ and $D = 2[p(b-1)]T_{comm}$. Since Figure 4 depicts a family of bad examples obtained by varying p from 1 to ∞, the ratio R differs for different instances of the family. It can be shown that for all the instances, A/B increases with p and asymptotically reaches the value 1.606695... (for $p \to \infty$). Also, C/D decreases for $p = 2, 3\ldots$ and can be shown to have an upper limit of 1.25 (corresponding to $p = 2$). Since $A/B \in [1, 1.6067)$ and $C/D \in [1, 1.25]$, it follows that $R \in [1, 1.6067)$, since R is the mediant of A/B and C/D. R_L is the largest possible value of R, which is,

$$R_L = 1.6067$$

It is interesting to note that the largest value of R is obtained when $T_{comm} = 0$. This is as expected, since RDMS will perform better in the presence of data transfer ($T_{comm} \neq 0$), because it is designed to minimize off-chip data transfer overheads.

Derivation of the upper limit R_U for the worst-case ratio is difficult. However, as we have observed for the extreme example, the worst-case performance is observed when the communication volume is zero. It is reasonable to extend this argument for all graphs, that is, RDMS will in general have the worst performance when no data communication is present in the task graph. When the task graph has no data communication, the problem to be solved is a BPP, and RDMS in this case behaves identically to the subset-sum algorithm. Using detailed mathematics, it has been shown in Caprara and Pferschy [2004] that the subset-sum algorithm has an upper limit worst-case performance of 1.6210... We therefore conjecture that for RDMS, $R_U = 1.6210$. The worst-case performance ratio R^{∞} for the RDMS algorithm would therefore satisfy

$$R^{\infty} \in (1.6067, 1.6210) \tag{11}$$

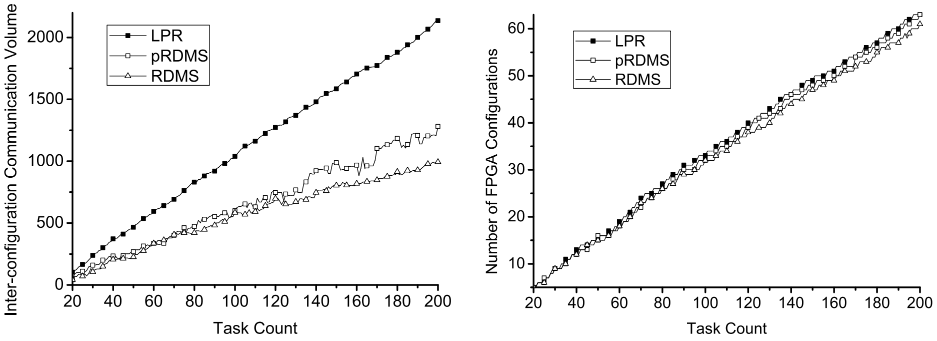
In general, the RDMS algorithm will give a solution close to the optimum, and (11) is only a bound on the worst-case value.

4. EXPERIMENTAL RESULTS

In order to demonstrate the advantage of the proposed RDMS algorithm, we have compared it with two other solutions that only consider hardware task resource utilization and data dependency. One algorithm is the previous version of the RDMS algorithm, which was presented in Huang et al. [2008] and referred to as pRDMS hereafter. The other algorithm adopts a Linear Programming Relaxation [Kellerer et al. 2004] approach, referred to as LPR. Three different comparisons are made among the three algorithms. First, a direct comparison is carried out using the example graph in Figure 3(a). Second, randomly generated data flow graphs are used to cover a comprehensive scope of different applications. Third, the task graph from a real-life application, an astrophysics N-body simulation, is scheduled using RDMS and pRDMS, by applying constraints from real HPRCs, the SRC-6 and Cray XD1. Finally, based on the scheduling result obtained from RDMS, the N-body application is emulated on SRC-6 hardware platform and the measured results compared against the theoretical expectations.

4.1 Scheduling Comparison on Example Graph

The pRDMS algorithm is similar to the RDMS algorithm; however, it does not consider the intertask communication during the scheduling process. The pRDMS algorithm schedules the 13 nodes of the example graph in Figure 3(a) also into four FPGA configuration, that is, {H2,H3,H4,H6}, {H1,H7,H9,H10,H12},{H5,H8} and {H11,H13}. A direct comparison with result from RDMS shows that RDMS reduces the inter-configuration communication time by 21.1%.



(a) Inter-configuration Communication (b) Number of FPGA Configurations

Fig. 5. Scheduling efficiency comparison among three approaches when intertask communication time is much smaller than task configuration time.

Table I. Scheduling Efficiency Improvement on Randomly Generated Graphs Using RDMS Against Other Two Algorithms

	Inter-configuration Communication			Number of Configurations		
	Sim 1*	Sim 2*	Sim 3*	Sim 1	Sim 2	Sim 3
LPR	49.1%	39.7%	42.7%	4.3%	3.9%	4.4%
pRDMS	13.0%	7.0%	13.1%	1.8%	1.4%	1.9%

*Sim 1: Simulation 1; Sim 2: Simulation 2; Sim 3: Simulation 3.

In the LPR approach to schedule hardware tasks represented by a DAG, nodes are scheduled level by level and the nodes in the next level are not considered until all nodes in the current level are scheduled. The typical LPR approach is to sort nodes in the same level into a decreasing order based on profit per weight ratio. Since the profit of each node is defined as configuration time and is linear to its weight, all nodes have the same profit per weight ratio in our case. Therefore, all nodes in the same level are sorted in a increasing order based on weight and then scheduled in a sequence. Once one FPGA configuration has no spare space to accommodate the node under consideration, it starts a new configuration. The LPR algorithm schedules the 13 nodes of the example graph into six FPGA configurations, that is, {H2,H3,H4}, {H1,H6,H7}, {H5,H9,H10}, {H8}, {H11,H12}, {H13}. Correspondingly, RDMS reduces the interconfiguration communication time by 63.4% compared with LPR in this case.

4.2 Scheduling Comparison on Randomly Generated Synthetic Graphs

In order to compare the scheduling efficiency among the three algorithms, that is, LPR, pRDMS and RDMS, these algorithms have been implemented in C++. Randomly generated task graphs of node count from 20 to 200 were applied and the number of configurations and interconfiguration communication time were recorded. For each task graph, there are ten nodes in each level. Every node was randomly connected to one to three parent nodes; the weight and the configuration time of each node were the same and they were randomly assigned between 1 and 50. Three different simulations were carried out, representing three different types of systems.

(1) *Simulation* 1. The intertask communication time is randomly assigned between 1 and 10. In other words, the intertask communication time is much smaller than task configuration time. This simulation represents those RC systems that have a long configuration time.
(2) *Simulation* 2. The intertask communication time is randomly assigned between 1 and 50. In other words, the intertask communication time is comparably the same as the task configuration time. This simulation represents those RC systems that have a medium configuration time.
(3) *Simulation* 3. The intertask communication time is randomly assigned between 1 and 100. In other words, the intertask communication time is much larger than the task configuration time. This simulation represents those RC systems that have a very short configuration time.

By observing the scheduling results shown in Figure 5 and Table I, it can be seen that the RDMS algorithm is capable of reducing the interconfiguration communication time by 11% and 44% on an average, compared with pRDMS and LPR algorithms, respectively. In terms of the number of FPGA configurations, the RDMS algorithm generates on average, 2% and 4% less configurations than pRDMS and LPR, respectively.

4.3 Astrophysics N-Body Simulation

4.3.1 *Application Description.* The target application we intend to implement on a reconfigurable computer is part of an astrophysical N-Body simulations where gas-dynamical effects are treated by a smoothed particle hydrodynamics (SPH) method [Lienhart et al. 2002; Lucy 1977; Monaghan and Lattanzio 1985]. The principle of this method is that the gaseous matter is represented by particles, which have a position, velocity and mass. In order to form a continuous distribution of gas from these particles, they are smoothed by folding their discrete mass distribution with a smoothing kernel W. This folding means that the point masses of the particles become smeared so that they form a density distribution. At a given position the density is calculated by summing the smoothed densities of the surrounding particles. Mathematically, the summation can be written as

$$\rho_i = \sum_{j=1}^{N} m_j W\left(\vec{r}_i - \vec{r}_j, h\right), \tag{12}$$

where h is the smoothing length specifying the radius of the summation space.

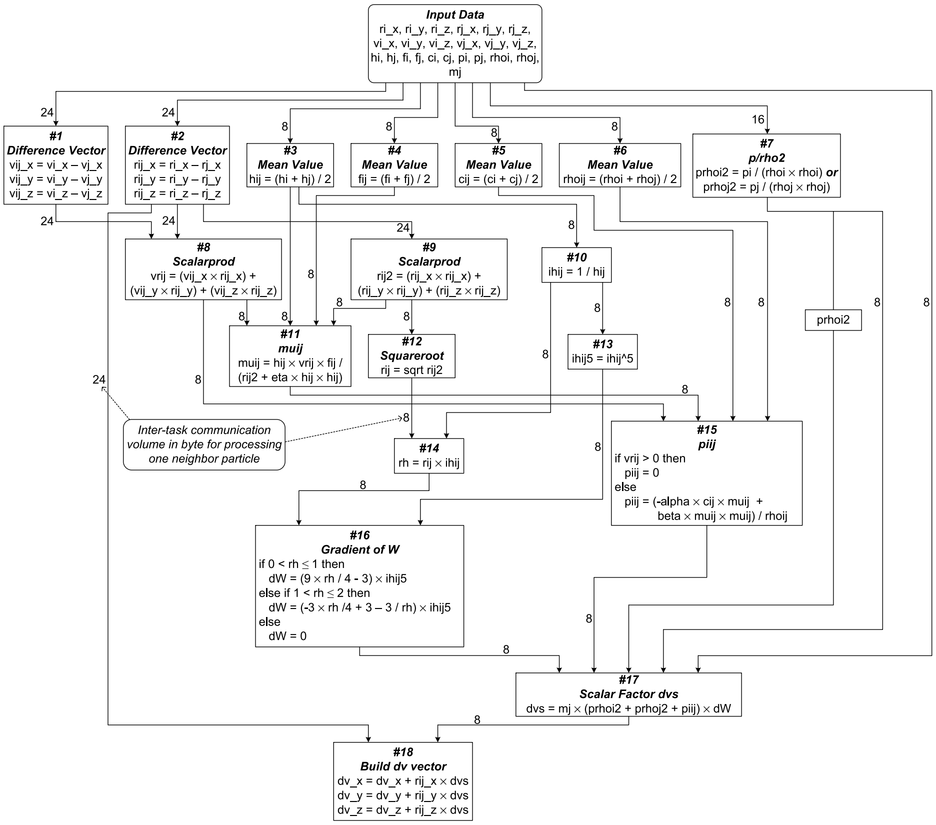


Fig. 6. Data flow graph of SPH pressure force calculation (with assigned node number in each box).

Commonly used smoothing kernels are strongly peaked functions around zero and are non-zero only in a limited area. A natural choice for the kernel is the spherically symmetric spline kernel proposed by Monaghan and Lattanzio [1985], defined by

$$W\left(\vec{r}_i - \vec{r}_j, h\right) = \frac{1}{h^3} B\left(x = \frac{\left|\vec{r}_i - \vec{r}_j\right|}{h}\right)$$
$$\text{with } B(x) = \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & 0 < x \le 1 \\ \frac{1}{4}(2-x)^3 & 1 < x \le 2 \\ 0 & x > 2. \end{cases} \tag{13}$$

The important point of SPH is that any gas-dynamical variables and even their derivatives can be calculated by a simple summation over the particle data multiplied by the smoothing kernel or its derivative. The motion of the SPH particles is determined by the gas-dynamical force calculated via the smoothing method, and the particles move as Newtonian point masses under this force. Equation (14) is the physical formulation of the velocity derivative given by the pressure force and the artificial viscosity.

$$\frac{d\vec{v}_i}{dt} = -\frac{1}{\rho_i}\nabla P_i + \vec{a}_i^{visc}. \tag{14}$$

The SPH method transforms Eq. (14) to the formulation in Eq. (15), which is only one of several possibilities.

$$\begin{aligned}
&\frac{d\vec{v}_i}{dt} = -\sum_{j=1}^{N} m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} + \Pi_{ij} \right) \nabla_i W\left(\vec{r}_{ij}, h_{ij}\right) \\
&\Pi_{ij} = \begin{cases} \frac{-\alpha c_{ij}\mu_{ij} + \beta \mu_{ij}^2}{\rho_{ij}} & \vec{v}_{ij}\vec{r}_{ij} \leq 0 \\ 0 & \vec{v}_{ij}\vec{r}_{ij} > 0 \end{cases} \\
&\vec{r}_{ij} = \vec{r}_i - \vec{r}_j, \vec{v}_{ij} = \vec{v}_i - \vec{v}_j, \rho_{ij} = \frac{\rho_i + \rho_j}{2} \\
&f_{ij} = \frac{f_i + f_j}{2}, c_{ij} = \frac{c_i + c_j}{2}, h_{ij} = \frac{h_i + h_j}{2} \\
&\mu_{ij} = \frac{h_{ij}\vec{v}_{ij}\vec{r}_{ij}}{\vec{r}_{ij}^{\,2} + \eta^2 h_{ij}^2} f_{ij}.
\end{aligned} \tag{15}$$

The gradient of W in Eq. 15 consists of three components in Cartesian coordinates (x, y, z). The x-component of the gradient of the smoothing kernel is shown in Eq. 16. The calculation of y and z-components is the same.

$$\frac{\partial_i W}{\partial x} = \begin{cases} \left(\frac{9|\vec{r}_{ij}|}{4h_{ij}^6} - \frac{3}{h_{ij}^5} \right)\left(r_{ix} - r_{jx}\right) & 0 < \frac{|\vec{r}_{ij}|}{h_{ij}} \leq 1 \\ \left(-\frac{3|\vec{r}_{ij}|}{4h_{ij}^6} + \frac{3}{h_{ij}^5} - \frac{3}{h_{ij}^4|\vec{r}_{ij}|} \right)\left(r_{ix} - r_{jx}\right) & 1 < \frac{|\vec{r}_{ij}|}{h_{ij}} \leq 2. \\ 0 & \frac{|\vec{r}_{ij}|}{h_{ij}} > 2 \end{cases} \tag{16}$$

The diagram in Figure 6 shows the data flow to compute Eq. (15), which consists of a total of 18 tasks. The communication volume of the edges in Figure 6 is based on the task requirements, obtained using the equations describing each task. For example, task #1 uses six double precision variables as inputs; three of these variables, viz., vi_x, vi_y, and vi_z, need to be transferred only once when calculating the SPH pressure force for particle i. Therefore, only three variables, corresponding to the neighbor particles, are shown as inputs to task #1. The input edge to task #1 is therefore shown to carry 24 bytes.

4.3.2 *Testbeds.* To obtain realistic performance estimates for the SPH pressure force calculation pipeline, we have used the system parameters from two existing HPRC platforms, the SRC-6 and Cray XD1. The whole FPGA configuration time is 130 ms on SRC-6 and 1,824 ms on Cray XD1, respectively. On both platforms, the bandwidth of interconnect is 1.4×10^9 B/s.

Table II. Resource Utilization of Pipelined Double-precision (64-bit) Floating-point Operators

	+/−	×	÷	√
Slices	1,640*	2,085*	4,173*	2,700*

*+/−: [Govindu et al. 2005], ×: [Zhuo and Prasanna 2007],
÷: [Hemmert and Underwood 2006], √: [Thakkar and Ejnioui 2006]

Table III. Resource Utilization of Hardware Tasks

Node No.	Operator* Combination	Slices	Percentage of Device Utilization†	
			XC2V6000	XC2VP50
1,2	3A	4,920	17.13	24.51
3,4,5,6	1A	1,640	5.71	8.17
7	1M,1D	6,258	21.79	31.17
8,9	2A,3M	9,535	33.20	47.50
10	1D	4,173	14.53	20.79
11,15	1A,4M,1D	14,153	49.27	70.50
12	1S	2,700	9.40	13.45
13	4M	8,340	29.04	41.55
14	1M	2,085	7.26	10.39
16	3A,4M,1D	17,433	60.69	86.84
17	2A,2M	7,450	25.94	37.11
18	3A,3M	11,175	38.91	55.67
Overall	24A,29M,5D,1S	123,390	429.59	614.68

*A: adder/subtractor, M: multiplier, D: divider, S: square root.
†Assume 15% of slices in device are reserved for vendor service logic.

RDMS and pRDMS algorithm are used for scheduling the graph given in Figure 6. LPR results are not considered here, since the previous subsection clearly demonstrates that RDMS and pRDMS outperform LPR by a large margin. In order to get desired performance and computation accuracy for the graph in Figure 6, fully pipelined double-precision (64-bit) floating point arithmetic units are needed for implementation on hardware. Many researchers have reported various floating-point arithmetic designs on FPGA devices [Govindu et al. 2005; Hemmert and Underwood 2006; Thakkar and Ejnioui 2006; Zhuo and Prasanna 2007]. The resource utilization of pipelined double-precision (64-bit) floating-point operators based on available literature is listed in Table II. These primitive operators are used to construct the functionality of nodes in Figure 6. In general, multiple primitive operators are used to build a pipelined hardware node so that all operations in one FPGA configuration can be executed in parallel to maximize the throughput. For instance, node #11 needs 1 adder, 4 multipliers and 1 divider, which is denoted as “1A,4M,1D” in Table III. The amount of slices occupied by each node is simply the summation of the slices of the primitive operators. We list the percentage of slice utilization of each node on the FPGA devices of the two representative reconfigurable computers in Table III, that is, Virtex-II6000 with SRC-6 and Virtex-IIP50 with Cray XD1. It is evident that multiple FPGA configurations are required to implement the dataflow graph in Figure 6 on both platforms.

Table IV. Hardware Module "Configuration Time" and Intertask CommunicationTime (unit: ms)

Node	"Configuration Time"*		Intertask Communication Time					
No.	SRC-6	Cray XD1	S. N.†	D. N.†	Time	S. N.	D. N.	Time
1,2	22.27	447.05	0‡	1	27.43	6	15	9.14
3	7.42	149.02	0‡	2	27.43	7	17	9.14
4	7.42	149.02	0‡	3	9.14	8	15	9.14
5	7.42	149.02	0‡	4	9.14	8	11	9.14
6	7.42	149.02	0‡	5	9.14	9	11	9.14
7	28.32	568.63	0‡	6	9.14	9	12	9.14
8,9	43.16	866.39	0‡	7	18.29	10	14	9.14
10	18.89	379.17	0‡	17	9.14	10	13	9.14
11	64.06	1286.00	1	8	27.43	11	15	9.14
12	12.22	245.33	2	18	27.43	12	14	9.14
13	37.75	757.80	2	8	27.43	13	16	9.14
14	9.44	189.45	2	9	27.43	14	16	9.14
15	64.06	1286.00	3	11	9.14	15	17	9.14
16	78.90	1584.03	3	10	9.14	16	17	9.14
17	33.72	676.94	4	11	9.14	17	18	9.14
18	50.58	1015.40	5	15	9.14			

*The "task configuration time" is a hypothetical value, as explained in Section 3.

†S. N.: Source Node; D. N.: Destination Node.

‡Although the communication between node #0 and other nodes is listed for the sake of completeness, it is not included in the calculation of the interconfiguration communication.

4.3.3 *Result.* As shown in Figure 6, the data of every particle consists of 13 scalar variables, that is, 104 bytes.[5] If we assume that the number of particles to be considered is $\mathcal{N}$ and the number of neighbors of each particle is $\mathcal{K}$, then the initial storage requirement is $104\mathcal{N}$ bytes and the pipeline in Figure 6 needs to perform $\mathcal{N} \times \mathcal{K}$ iterations. For the experiment carried out using RDMS and pRDMS, the number of particles is set to $\mathcal{N}$=16,000 and the number of neighbors of each particle is set to $\mathcal{K}$=100. As mentioned before, constraints from the two platforms, the SRC-6 and Cray XD1, are used during the scheduling process. That is, both pRDMS and RDMS use the hardware task resource utilization and "task configuration time" listed in Tables III and IV. RDMS additionally uses the intertask communication time given in Table IV, which is computed based on the numbers next to the edges shown in Figure 6. To elaborate, if b is the number of bytes indicated in Figure 6 as the requirement for processing one neighbor particle, the corresponding communication volume during the whole simulation is $\mathcal{N} \times \mathcal{K} \times b$ bytes. The time taken for transferring these data is easily computed based on the interconnect bandwidth. For example, the total communication volume for intertask communication between task #2 and #8 is $16,000 \times 100 \times 24 = 3.84 \times 10^7$ bytes, which will need 27.43 ms for transfer over an I/O with bandwidth 1.4×10^9 B/s.

[5] Although there are six input variables in task #1, 3 variables, that is, vi_x, vi_y and vi_z, are needed to be transferred only once when calculating the SPH pressure force of particle i. The three double-precision inputs in Figure 6 are for transferring the neighbor particles. The same fact applies to task #2, #3, #4, #5, #6 as well.

Table V. SPH Pressure Force Scheduling Efficiency Comparison between pRDMS and RDMS

		pRDMS	RDMS
Inter-configuration Communication Time (s)	SRC-6	0.347	0.329
	Cray XD1	0.512	0.384
Number of FPGA Configurations	SRC-6	5*	5†
	Cray XD1	7‡	7§

*{1,2,6,7,8}, {3,4,5,9,10,13}, {11,15}, {12,14,16}, {17,18}.
†{1,2,6,7,8}, {3,4,5,9,10,12,14}, {11,15}, {13,16}, {17,18}, see Table VI for detail.
‡{1,2,3,4,7}, {8,9}, {6,11,12}, {5,10,13,14}, {16}, {15}, {17,18}.
§{1,2,8}, {5,6,7,9}, {3,4,10,12,13}, {14,16}, {11}, {15}, {17,18}.

The number of FPGA configurations and the inter-configuration communication time, obtained from the scheduling algorithms for both platforms, are listed in Table V. The inter-configuration communication time is obtained by summing up the I/O time for all configurations. For example, the fourth configuration obtained using RDMS for SRC-6 consists of the tasks #13 and #16. From Figure 6, this configuration would require two inputs (8 bytes each) and one output (8 bytes), or collectively 24 bytes per neighbor particle. These 24 bytes corresponds to a total I/O transfer time of 27.43 ms using a 1.4 GB/s interconnect, as shown earlier. Similar computations are carried out for all configurations. To illustrate the actual numbers obtained for one case (RDMS for SRC-6), the values for all the inter-configuration communications are listed in Table VI under the column labeled 'Estimated (using 1.4GB/s)'. The procedure outlined here is carried out for RDMS as well as pRDMS, using the parameters for the two platforms.

Based on the scheduling results, the following observations can be made. On the SRC-6 platform, RDMS reduces the communication time by 5% compared to pRDMS. An analysis of both scheduling sequences has shown that pRDMS has eliminated some high volume data communication simply because they belong to the first tasks that were combined. For example, RDMS has scheduled task #9, #10, #12 and #14 into the same configuration whereas pRDMS has scheduled them into two separate configurations, which increases the overall interconfiguration communication. On Cray XD1, the communication reduction is increased to 25% since RDMS successfully schedules those tasks between which there exists heavy communications into the same configuration, for example, #1, #2 and #8. The communication reduction is higher because Cray XD1 uses smaller FPGAs and therefore it is less likely that high volume communication transfers are covered by pRDMS. To summarize, RDMS algorithm is effective in reducing the communication overhead for a real-life application like the N-body simulation, in addition to reducing the number of configurations.

4.3.4 *Experiments on Hardware and Discussion of the Limitations.* In order to demonstrate the applicability of the proposed algorithm, and also to understand its limitations, we have emulated the execution of the N-body application on the SRC-6 platform. The task graph for N-body simulation is first scheduled using RDMS, to determine the constituent configurations as described in the previous subsection. We have observed that the practically

Table VI. The RDMS Implementation of SPH Pressure Force on the SRC-6

FPGA Configuration	Tasks	Interconfiguration Communication Time (ms)					
		Estimated (using 1.4 GB/s)		Estimated (using 800 MB/s)		Measured	
		Input	Output	Input	Output	Input	Output
1	1,2,6,7,8	0	91.42	0	159.99	0	163.55
2	3,4,5,9,10,12,14	27.43	54.84	48.00	95.97	48.13	98.35
3	11,15	63.98	9.14	111.97	16.00	112.30	16.31
4	13,16	18.28	9.14	31.99	16.00	32.10	16.30
5	17,18	54.85	0	95.99	0	96.26	0
Total		164.54	164.54	287.95	287.95	288.79	294.51

achieved interconnect bandwidth on the SRC-6 is 800 MB/s as against the theoretically achievable 1.4 GB/s, when the amount of data per transaction is comparably small, say 4 MB. This practical value is used in obtaining the scheduling result. It turns out that the tasks in each configuration, with this reduced interconnect bandwidth, is the same as that listed in the footnote of Table V.

After obtaining the schedule, the execution of the application is emulated on hardware. To be specific, in the absence of double-precision floating point implementations for the constituent tasks of N-body simulation, we have emulated the execution of each task by simple delays. Rather than checking the functionality of the application, the purpose of this experiment is to verify the proposed approach based on actual data transfers and in the presence of reconfiguration of the FPGA.

Table VI shows the measured results from the experiments on SRC-6, compared against the theoretically expected results. The results show that the theoretical expectations are very close to the experimentally measured results. This clearly demonstrates the applicability of the proposed algorithm.

The scheduling results and experiments reveal some of the limitations of the RDMS algorithm. For instance, the RDMS algorithm does not try to reuse shared edges in a task graph. As an example, consider tasks #11 and #15 in Figure 6, which comprise the third configuration scheduled by RDMS. Both of these tasks use the same output from task #8, yet the same data is transferred separately for the two tasks even though they reside in the same configuration. This is reflected in the task communication time (measured and expected values) listed in Table VI.

The RDMS algorithm also does not consider limitations of bandwidth of locally attached memory. While this assumption is accurate when internal block RAM of the FPGA is used, there could be some limitations in the presence of external local memory. As an example, consider again the configuration *3*, which consists of tasks #11 and #15. From the numbered edges in Figure 6, it can be deduced that during execution, this configuration requires seven input ports and one output port to access local memory, with each port having a width of eight bytes (or 64-bits). This requirement cannot be directly satisfied in the SRC-6, since it features only six local SRAM modules attached to one FPGA. The FPGA in the SRC-6 therefore has access to only six ports of 64-bits each. Sharing of this limited bandwidth for the tasks would result in a degradation of the task execution time. Detailed execution of the tasks is currently not modeled in our emulation experiments.

It is also worth mentioning that RDMS currently works only for the cases when the data processed is very large, thereby allowing us to ignore pipeline latencies. Tasks within a configuration are therefore considered to execute in parallel. Further research is needed to extend RDMS for the cases when smaller data is processed, as well as when task execution is not carried out in a pipelined fashion (i.e., tasks within a configuration execute one after the other). Additional considerations include limitations in the size of local memory attached to the FPGA. One possible solution to this problem is to use multiple installments of input and output data transfer, interleaved with execution. The other alternative is to reconfigure the FPGA several times for the same sequence of configurations, with each sequence operating only on a part of the data (so I/O transfer occurs only for the first and last configuration of one sequence of configurations). There will definitely be tradeoffs between choosing multiple configuration sequences (with reduced data transfers) versus multiple data installments (and fewer configuration sequences); further study is required to analyze these tradeoffs.

5. CONCLUSIONS

In this article, we have proposed the Reduced Data Movement Scheduling (RDMS) algorithm, for scheduling tasks assigned to FPGA co-processors on RC systems. RDMS reduces the inter-configuration communication overhead and FPGA configuration overhead by taking data dependency, hardware task resource utilization and intertask communication into account during the scheduling process. Mathematical analysis of the algorithm shows that the time complexity is $O(n^3W^2)$ in the worst-case, and the maximum space requirement is $O(n(n+W))$, for a graph with n nodes. These are reasonable requirements for a static task scheduling algorithm. The worst-case performance ratio of RDMS is expected to have a value between 1.6067 and 1.6210, which is good for a heuristic scheduler that takes into account many constraints. Simulation results show that the RDMS algorithm is able to reduce inter-configuration communication time by 11% and 44% respectively and also generate fewer FPGA configurations, compared to pRDMS and LPR algorithms, which only consider data dependency and task resource utilization during scheduling. The scheduling results obtained for a real-life application, N-body simulation, for SRC-6 and Cray XD1 platforms, verifies the efficiency of RDMS algorithm. Experiments on SRC-6 further validate the proposed approach.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their comments that greatly helped in improving the quality of the paper. The authors are grateful to Dr. Mohamed Bakhouya for helpful discussions. The authors would like to thank Guillermo Marcus from Institute for Computer Engineering of the University of Heidelberg in Germany for helpful discussions about N-body simulation using FPGAs. The authors also would like to acknowledge

Gerhard Lienhart, Andreas Kugel and Reinhard Männer from University of Mannheim in Germany for providing the data flow graph of SPH pressure force calculation.

REFERENCES

BAZARGAN, K., KASTNER, R., AND SARRAFZADEH, M. 2000. Fast template placement for reconfigurable computing systems. *IEEE Des. Test Comput. 17*, 1, 68–83.

BREBNER, G. AND DIESSEL, O. 2001. Chip-based reconfigurable task management. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'01)*. 182–191.

CAPRARA, A. AND PFERSCHY, U. 2004. Worst-case analysis of the subset sum algorithm for bin packing. *Oper. Res. Lett. 32*, 20, 159–166.

COFFMAN, JR., E. G., GAREY, M. R., AND JOHNSON, D. S. 1996. Approximation algorithms for bin packing: a survey. In *Approximation Algorithms for NP-Hard Problems*. D. Hochbaum Ed., PWS Publishing, Boston. 46–93.

COMPTON, K., LI, Z., COOLEY, J., KNOL, S., AND HAUCK, S. 2002. Configuration relocation and defragmentation for run-time reconfigurable computing. *IEEE Trans. VLSI Syst. 10*, 3, 209–220.

DIESSEL, O., ELGINDY, H., MIDDENDORF, M., SCHMECK, H., AND SCHMIDT, B. 2000. Dynamic scheduling of tasks on partially reconfigurable FPGAs. *IEE Proc. Comput. Digital Techniq.* (Special Issue on Reconfigurable Systems) *147*, 3, 181–188.

FEKETE, S. P., KÖHLER, E., AND TEICH, J. 2001. Optimal FPGA module placement with temporal precedence constraints. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'01)*. 658–665.

GOVINDU, G., SCROFANO, R., AND PRASANNA, V. K. 2005. A library of parameterizable floating-point cores for FPGAs and their application to scientific computing. In *Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'05)*. 137–145.

HANDA, M. AND VEMURI, R. 2004. A fast algorithm for finding maximal empty rectangles for dynamic FPGA placement. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*. Vol. 1. 744–745.

HEMMERT, K. S. AND UNDERWOOD, K. D. 2006. Open source high performance floating-point modules. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*. 349–350.

HUANG, M., SIMMLER, H., SAHA, P., AND EL-GHAZAWI, T. 2008. Hardware task scheduling optimizations for reconfigurable computing. In *Proceedings of the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA'08)*.

HUANG, M., SIMMLER, H., SERRES, O., AND EL-GHAZAWI, T. 2009. RDMS: A hardware task scheduling algorithm for reconfigurable computing. In *Proceedings of the 16th Reconfigurable Architectures Workshop (RAW'09)*.

KELLERER, H., PFERSCHY, U., AND PISINGER, D. 2004. *Knapsack Problems*. Springer, Berlin.

KLEINBERG, J. AND TARDOS, É. 2005. *Algorithm Design*. Pearson/Addison-Wesley, Boston, MA.

LIENHART, G., KUGEL, A., AND MÄNNER, R. 2002. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*. 182–191.

LUCY, L. B. 1977. A numerical approach to the testing of the fission hypothesis. *Astronom. J. 82*, 12, 1013–1024.

MONAGHAN, J. J. AND LATTANZIO, J. C. 1985. A refined particle method for astrophysical problems. *Astron. Astrophys. 149*, 135–143.

PISINGER, D. 1999. Linear time algorithms for knapsack problems with bounded weights. *J. Algor. 33*, 1, 1–14.

SAHA, P. 2007. Automatic software hardware co-design for reconfigurable computing systems. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'07)*. 507–508.

THAKKAR, A. J. AND EJNIOUI, A. 2006. Design and implementation of double precision floating point division and square root on FPGAs. In *Proceedings of the IEEE Aerospace Conference*.

WALDER, H. AND PLATZNER, M. 2002. Non-preemptive multitasking on fpga: Task placement and footprint transform. In *Proceedings of the 2nd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*. 24–30.

WALDER, H., STEIGER, C., AND PLATZNER, M. 2003. Fast online task placement on FPGAs: free space partitioning and 2D-hashing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*. 178–185.

WIANGTONG, T., CHEUNG, P., AND LUK, W. 2003. Multitasking in hardware-software codesign for reconfigurable computer. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'03)*. Vol. 5. 621–624.

ZHUO, L. AND PRASANNA, V. K. 2007. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Trans. Para. Distrib. Syst. 18*, 4, 433–448.

Received March 2009; revised July 2009; accepted August 2009