# An Approximation Algorithm for Scheduling on Heterogeneous Reconfigurable Resources

Ani Nahapetian[1], Philip Brisk[2], Soheil Ghiasi[3], Majid Sarrafzadeh[1]
[1]Computer Science Department, UCLA, {ani, majid}@cs.ucla.edu
[2]School of Computer and Communication Sciences, EPFL, philip.brisk@epfl.ch
[3]Department of Electrical and Computer Engineering, UC Davis, soheil@ece.ucdavis.edu

## ABSTRACT
Dynamic reconfiguration imposes significant penalties in terms of performance and energy. Scheduling the execution of tasks on a dynamically reconfigurable device is therefore of critical importance. Likewise, other application domains have cost models that are effectively the same as dynamic reconfiguration; examples include: data transmission across multiprocessor systems; dynamic code updating and reprogramming of motes in sensor networks; and module allocation, wherein the sharing of resources effectively eliminates inherent reconfiguration costs.

This paper contributes a fully polynomial time approximation algorithm for the problem of scheduling independent tasks onto a fixed number of heterogeneous reconfigurable resources, where each task has a different hardware and software latency on each device; the reconfiguration latencies can also vary between resources. Assuming that P != NP, an optimal algorithm for this problem would exhaustively enumerate all assignments of tasks onto resources; this search space is exponential in size. The algorithm presented in this paper is a fully polynomial ε-approximation. Using a trimming procedure, a provably polynomial sub-space of possible solutions is explored while ensuring that the solution found is within a factor of at most 1+ε from the optimal solution. This approach can also extended to task graph scheduling in which one task may be dependent on several others.

A general-purpose processor and an FPGA were used to experimentally validate the proposed technique using a pair of encryption algorithms. The latencies of the schedules obtained by the approximation scheme were at most 1.1x longer than the optimal solution, which was found using integer linear programming; this result is better than the theoretical worst-case guarantee of the approximation algorithm, which was 1.999x. The length of the schedules obtained using list scheduling, a well-known polynomial-time heuristic, were at most 2.6x longer than optimal, demonstrating that the proposed approximation algorithm is useful both in theory and practice.

## Keywords
Dynamic Reconfiguration, Heterogeneous Resources, Fully Polynomial Approximation Algorithm.

## 1. INTRODUCTION
Reconfigurable computing has the potential to dynamically adapt a hardware platform to meet the requirements of an application; however, the overhead associated with dynamic reconfiguration has inhibited the widespread use of this promising technology over the past two decades. In many cases, the delay of dynamic reconfiguration is comparable to the runtime of the application; even when dynamic reconfiguration is used, its latency—rather than the latency of the application—will dominate the overall runtime. Consequently, effective scheduling algorithms that can map tasks onto a set of reconfigurable resources—including the scheduling of reconfiguration operations—are required in order to make reconfigurable computing a reality.

Virtually all modern parallel systems employed today are composed of heterogeneous components, each of which has an associated reconfiguration cost. The most obvious component would be a dynamically programmable FPGA, or some other type of coarse-grained reconfigurable device. In a more general sense, the time and energy consumed through the transmission of code and/or data to remote processors across a network can also be modeled as a form of reconfiguration [30][31], even though the underlying hardware platform does not change. One particularly poignant example of this type of reconfiguration is dynamic module uploading in sensor networks.

Another application is the allocation of various library modules to available resources given a fixed area constraint. Resource sharing among IP cores can reduce the area, as well as other costs, and can therefore maximize the number of library modules implemented in hardware. [24] Resource sharing, in this context, can be modeled as the absence of a reconfiguration cost. Sharing resources can reduce the amount of time spent reconfiguring a device at runtime, and can therefore reduce the overall application latency.

All of these problems can be reduced to a straightforward scheduling problem on a set of heterogeneous resources with heterogeneous reconfiguration costs.

## 1.1 Contribution

This paper presents a full polynomial-time ε-approximation algorithm that solves this scheduling problem. An approximation algorithm is a heuristic whose solution quality is guaranteed to be at most a constant factor away from the optimal solution. A fully polynomial-time approximation algorithm is one whose solution quality and runtime can be varied according to a term denoted ε [2, 7].

The ε-approximation algorithm presented in this paper is based on an optimal scheduling algorithm that considers all possible assignments of tasks to resources. A subset of the possible assignments is considered; the specific choice of which assignments to consider yields the ε-approximation.

Two sets of experiments confirm the efficacy of the ε-approximation approximation algorithm. The first set of experiments, on a randomly generated set of input data, compares the ε-approximation algorithm to list scheduling, a well-known and well-understood heuristic that has been used in the past for a wide variety of scheduling problems, and to an optimal integer linear programming (ILP) formulation of the problem. The ε-approximation algorithm found the optimal solution in the vast majority of cases, while list scheduling found optimal solutions rarely.

The second set of experiments focuses on parallel encryption using the Advanced Encryption Standard (AES, a.k.a. Rijndael) algorithm: blocks of data to be encrypted are scheduled on a system containing a general-purpose processor and an FPGA. Once again, the ε-approximation algorithm found optimal and near-optimal solutions in all cases, while list scheduling performed considerably worse. Altogether, these experiments validate the efficacy of the ε-approximation algorithm while confirming that the solutions fall within the theoretically established bounds.

Lastly, we briefly sketch a set of extensions to the ε-approximation algorithm that allow it to be used in the context of task graph scheduling, where tasks may be dependent on one another and where there is a communication costs between dependent tasks that are scheduled on distinct resources.

## 1.2 Paper Organization

The paper is organized as follows. Section 2 begins with a motivating example. Section 3 formalizes the problem statement. Section 4 presents two optimal solutions to the problem: an ILP formulation, which we implemented, and a dynamic programming algorithm, from which the ε-approximation algorithm is derived in Section 5. Section 6 presents the experimental evaluation of the ε-approximation algorithm and compares it to list scheduling and the ILP formulation. Section 7 describes a set of extensions that adapt the ε-approximation algorithm to task graph scheduling. Section 8 proceeds to summarize related work, and lastly, Section 9 concludes the paper.

**Table 1. Tasks to schedule.**

| Task | Task Type | Execution Time on Processor | Execution Time on FPGA |
|---|---|---|---|
| 1 | 1 | 5 | 2 |
| 2 | 2 | 4 | 3 |

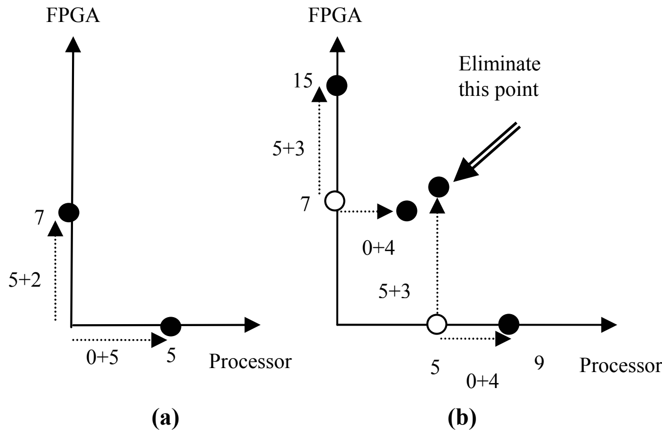



**Figure 1. (a) Assignment of Task 1; (b) Assignment of Task 2.**

# 2. MOTIVATING EXAMPLE

This section presents a motivating example for the scheduling problem that illustrates the key point of the ε-approximation algorithm, which is the primary contribution of this paper. We are given two resources: a processor and an FPGA with respective reconfiguration costs of 0 and 5. Table 1 lists two tasks to be scheduled, including their respective runtimes on the processor an FPGA. The tasks have distinct types, meaning that the FPGA must be reconfigured between the execution of one task and another.

Scheduling is performed in a 2-dimensional space, where the x-axis represents the runtime of tasks on the processor and the y-axis represents the FPGA. Fig. 1(a) considers the potential assignment of the first task to both of the resources. The points (0+5, 0) and (0, 5+2) represent the time taken to schedule the task on the processor and the FPGA, inclusive of reconfiguration costs. Fig. 1(b) proceeds to schedule the second task following the first. This yields 4 data points: (0, 15), (4, 7), (5, 8), and (9, 0).

The runtime of each data point (x, y) is max{x, y}. For example, point (4, 7) represents the parallel execution of Task 1 on the FPGA and Task 2 on the processor, requiring a total time of 7; point (5, 8) represents the parallel execution of Task 1 on the processor and Task 2 on the FPGA, requiring a total time of 8. Point (0, 15), in contrast,

represents the serial execution of both tasks on the FPGA, and point (9, 0) represents the serial execution of both tasks on the processor; the total times are 15 and 9, respectively.

The optimal solution for these two tasks is point (4, 7); however, this example also illustrates something further. Suppose that there were several additional tasks to be scheduled. In this case, the four points in Fig. 1(b) are partial solutions (it should be noted that none of these partial solutions is guaranteed to be part of the optimal solution). Within these partial solutions, however, it is quite clear that point (4, 7) is superior to point (5, 8): if there were more tasks to schedule, then (5, 8) could not possibly be part of an optimal schedule, as such a schedule could be improved via substitution of point (4, 7) instead. Hence, the point (4,7) dominates (5,8), and this point can be eliminated without comprising the optimality of the algorithm. The ability to remove points from consideration is a key feature of the ε-approximation algorithm described in Section 5.

# 3. PROBLEM STATEMENT

The problem addressed in this paper is to schedule *N* independent tasks onto a fixed set of *M* heterogeneous resources with heterogeneous reconfiguration costs; we assume that the scheduling is non-preemptive, meaning that the execution of a task is not interrupted once it has been scheduled. The goal is to minimize the *makespan*, i.e., the completion time of the final task.

Independent tasks means that there is no precedence relation among tasks, i.e., given a sufficient number of resources, all of the tasks can be executed in parallel; the problem, therefore, only becomes challenging when $M < N$. This is a special case of general task graph scheduling where the task graph contains no edges.

Each of the tasks can have different and unrelated execution times on each resource; each resource has an associated reconfiguration cost. We make two simplifying assumptions regarding reconfiguration costs: (1) reconfiguration is preemptive, i.e., no task may execute on a resource *during* reconfiguration; and (2) the reconfiguration cost is known a priori.

By extending the notion of a resource to also be a portion of an FPGA that can be partially reconfigured, partial reconfiguration can also be addressed. The approach presented in this paper is robust enough to handle partial reconfigurations, for cases where the portions of the FPGA to be partially reconfigured are known.

Clearly, this model abstracts away many low-level details. For example, consider a program executing on a processor. Here, we assume that the execution time is constant. In reality, there may be many different execution times, depending, for example, on which optimizations were enabled during compilation. The same is true for execution on an FPGA, as variations in high-level optimizations, technology mapping, floorplanning, placement, and routing can all affect the latency of the task.

Each task has an associated type, whose implication is as follows. Suppose that tasks $t_1$ and $t_2$ are scheduled to execute consecutively on resource r. If their types are different, then r must be reconfigured following the execution of $t_1$ in order to execute $t_2$; if their types are the same, then the reconfiguration is unnecessary. Exploiting task types to eliminate unnecessary reconfigurations is key to achieving low-latency schedules under this model.

An alternative model allows the reconfiguration costs to be associated with each task type rather than the resources. This, for example, can model partial reconfiguration for FPGAs. The advantage is that smaller tasks do not need to spend the time to reconfigure the whole device; however, there are still some restrictive assumptions: (1) This approach models reconfiguration time, but not area, i.e., two small tasks with short reconfiguration times cannot execute simultaneously on the same resource; (2) in practice, the reconfiguration time depends on both the task and the resource, e.g., the reconfiguration time of task t on an Altera Stratix-series FPGA is different from that of an Altera Cylone-series FPGA, and a Xilinx Virtex-series FPGA, etc. These details are abstracted away in our model.

## 3.1 Problem Details and Decomposition

The scheduling problem can be decomposed into two interacting subproblems: (1) the binding of tasks to resources; and (2) computing a schedule for each task on each resource. Due to the independence of tasks, the optimal solution to step (2) is to schedule all tasks of the same type that are bound to the same resource consecutively, as this minimizes the aggregate reconfiguration delay. This is implicit in scheduling heuristics presented in [21][22][28], and we formally prove its optimality in Theorem 1.

**Theorem 1.** Assume that each task is bound to exactly one resource. Then the optimal schedule executes all tasks of the same type consecutively on each resource.

**Proof.** Let $S = (t_1, t_2, \ldots, t_n)$ be an ordering of tasks that are bound to a resource, R. Now, let us decompose S into five sub-orderings, $S = S_1S_2S_3S_4S_5$, such that $S_1 = (t_1, t_2, \ldots t_a)$, $S_2 = (t_{a+1}, t_{a+2}, \ldots, t_b)$, $S_3 = (t_{b+1}, t_{b+2}, \ldots, t_c)$, $S_4 = (t_{c+1}, t_{c+2}, \ldots, t_d)$, $S_5 = (t_{d+1}, t_{d+2}, \ldots, t_n)$, $S_2$ and $S_4$ are non-empty and only contain tasks of type i, $S_3$ is non-empty, and $t_a$, $t_{b+1}$, $t_c$, and $t_{d+1}$ all have types other than i. We prove by contradiction that S is sub-optimal.

Assume to the contrary that S is an optimal ordering, i.e., that L(S), the latency of executing the tasks in order S, including reconfiguration latencies, is minimal among all orderings.

Let $\Sigma_k$ be the latency of executing all tasks in $S_k$, including reconfiguration latencies.

For $S_2$ and $S_4$, all tasks have type i. Let $\rho_i$ be the latency of reconfiguring R to support tasks of type i, and let $L_2$ and $L_4$ be the respective latencies executing the tasks in $S_2$ and $S_4$, without the reconfiguration latencies. Therefore, $\Sigma_2 = \rho_i + L_2$ and $\Sigma_4 = \rho_i + L_4$.

Then:

$$
\begin{aligned}
L(S) &= \Sigma_1 + \Sigma_2 + \Sigma_3 + \Sigma_4 + \Sigma_5 \\
&= \Sigma_1 + \rho_i + L_2 + \Sigma_3 + \rho_i + L_4 + \Sigma_5 \\
&= 2\rho_i + \Sigma_1 + L_2 + \Sigma_3 + L_4 + \Sigma_5 \\
&= 2\rho_i + X, \text{ where } X = \Sigma_1 + L_2 + \Sigma_3 + L_4 + \Sigma_5
\end{aligned}
$$

Now, consider a different task ordering S' = $S_1S_3S_2S_4S_5$. Since all tasks in $S_2$ and $S_4$ have type i, there is no need to reconfigure the device between $S_2$ and $S_4$. Therefore, the latency of executing the tasks in $S_2S_4$ is $\rho_i + L_2 + L_4$. Therefore, the total latency of S', denoted L(S') is:

$$
\begin{aligned}
L(S') &= \Sigma_1 + \Sigma_3 + \rho_i + L_2 + L_4 + \Sigma_5 \\
&= \rho_i + X \le 2\rho_i + X = L(S)
\end{aligned}
$$

If $\rho_i > 0$, then $L(S') < L(S)$, contradicting the assumption that S is optimal. It follows that all tasks of the same type must occur consecutively in an optimal ordering. □

In conclusion, Theorem 1 proves that obtaining an optimal schedule is trivial once tasks have been bound to resources; therefore, the most important problem is to determine precisely the binding of tasks to resources. This task, nonetheless, is NP-complete.

# 4. OPTIMAL ALGORITHMS

This section presents two optimal algorithms that solve the scheduling problem that was characterized in the preceding section. Subsection 4.1 presents an ILP formulation, which is used for comparison against the ε-approximation algorithm in Section 6. Subsection 4.2 presents an optimal dynamic programming algorithm from which the ε-approximation algorithm is derived.

## 4.1 Integer Linear Program

Fig. 2 shows the ILP formulation of the scheduling problem outlined in Section 3. The objective function is to minimize the makespan; since the goal is to minimize the makespan, it follows that the makespan is equal to the maximum finishing time among all resources. Constraint (1) ensures that each task is bound to exactly one resource; Constraint (2) ensures that the reconfiguration cost is calculated once for each task type that is bound to each resource, in accordance with Theorem 1. Constraint (3) ensures that the maximum finishing time of all of the tasks bound to each resource does not exceed the makespan.

## 4.2 Dynamic Programming Algorithm

The optimal algorithm is formulated as a dynamic programming problem. An *M*-dimensional space is created where *M* is the number of resources (e.g., *M = 2* in Fig. 1). Points are plotted onto the graph, according to their execution times, or costs. For example the point (3, 4, 0) represents 3 units of time on the first resource, 4 units of time on the second resource, and 0 units of time on the third.

Pseudocode for the optimal algorithm is shown in Fig. 3.

| Constants |
|---|
| N – number of tasks |
| M – number of resources |
| R – number of task types |
| $r_j$ – reconfiguration cost of resource j |
| $p_{i,j}$ – runtime of task i on resource j |
| $t_{i,k} = \begin{cases} 1 & \text{if task i is of type k} \\ 0 & \text{otherwise} \end{cases}$ |

| Variables |
|---|
| T - Makespan |
| $x_{i,j} = \begin{cases} 1 & \text{if task i is assigned to resource j} \\ 0 & \text{otherwise} \end{cases}$ |
| $y_{j,k} = \begin{cases} 1 & \text{if a task of type k is assigned to resource j} \\ 0 & \text{otherwise} \end{cases}$ |

| Integer Linear Program | |
|---|---|
| Minimize: T | |
| Subject to the following constraints: | |
| $\sum_{j=1}^{M} x_{i,j} = 1, \quad 1 \le i \le N$ | (1) |
| $y_{j,k} - x_{i,j} t_{i,k} \ge 0, \quad 1 \le i \le N, \quad 1 \le j \le M, \quad 1 \le k \le R$ | (2) |
| $\sum_{k=1}^{R} r_j y_{j,k} + \sum_{i=1}^{N} p_{i,j} x_{i,j} \le T, \quad 1 \le j \le M$ | (3) |

**Figure 2. Integer linear program formulation of the reconfigurable scheduling problem for independent tasks with heterogeneous resources.**

The algorithm proceeds as follows. First, the tasks are sorted according to their type, to allow for concise record keeping of reconfigurations. Next, the first *M* points are placed in the space. The reconfiguration cost, if applicable, is summed with the execution time to plot the points.

Each remaining task creates *M* new points for each existing point in the graph. The cost of each new point is the sum of the cost of the old point and the cost of the current task on the current resource. The reconfiguration cost is added, only if it has not yet been incurred for the current task type.

**Optimal Algorithm: Minimize Makespan**

```
1: Sort tasks to group together according to their type
2: for each task t
3:      ►Keep track of what set has been assigned
4:      if currentTaskType <> t.type
5:           currentTaskType = t.type
6:           for each old point o OldPointsList
7:                for each resource r
8:                     o.reconfiged[r] = 0
9:                end for
10:          end for
11:     endif
12:     ►Add M new points to the graph for each old point
13:     for each old point o in list of old points
14:          for each resource r
15:               Create a new point n
16:               ►Initialize new point to old point
17:               for each resource r1
18:                    n.Cost[r1] = o.Cost[r1]
19:                    n.reconfiged[r1] = o.reconfiged[r1]
20:               endfor
21:               ►Add new costs to the point
22:               n.Cost[r] = n.Cost[r] + t.Cost[r]
23:               if n.reconfiged[r] == 0
24:                    n.Cost[r] += resource[r].ReconfigCost
25:                    n.Cost[r] = 1
26:               endif
27:               add n to list of NewPointsList
28:          endfor
29:     endfor
30:     RemoveEliminatablePoints(NewPointsList)
31:     OldPointsList = NewPointsList
32: endfor
33: return assignment with the minimum makespan
```

**Figure 3. Optimal dynamic programming algorithm for the reconfigurable scheduling problem for independent tasks with heterogeneous resources.**

In the pseudocode, the position of the point in the space is represented by the array named *Cost*, which stores the cost incurred on each resource.

During each iteration after the new points are added, the old points are discarded. After all the tasks have been graphed, the optimal solution is the one with the smallest makespan. The makespan of all the points on the graph is calculated as the maximum of the finish times on each of the resources.

While plotting points, certain cases arise where a point will obviously not lead to an optimal solution, thus these points can be eliminated from consideration. This occurred, for example, with point (5, 8) in the example shown in Fig. 1. Fig. 4 shows pseudocode for the elimination procedure.

**RemoveEliminatablePoints(NewPointsList)**

```
1: for each point n in NewPointsList
2:      for every other point p in NewPointsList
3:           if n.Cost[r] <= p.Cost[r] for all resources r
           and (n.reconfiged[r] >= p.reconfiged[r] for all
           resources r or currentTaskType <>
           nextTaskType)
4:                remove n from newPointsList
5:           endif
6:      endfor
7: endfor
```

**Figure 4. Pseudocode to remove provably sub-optimal points from consideration during scheduling.**

Now, consider the case where all points of one type are plotted, but before any points of the next type have been plotted. The points that are larger in all dimensions can be eliminated. These points will not be a part of the optimal solution because a better solution exists up to that point. Eliminating these points will not diminish the quality of the solution due to the optimal substructure of the problem.

A larger cost between points of the same type is not enough to eliminate a point. Points with a larger cost are still potentially optimal, because they have already incurred the reconfiguration cost. If a point is larger than another point that has not incurred the reconfiguration cost, it is too early to say if the point will be larger than the other point after all the tasks of the same resource have been plotted.

Points, however, can be eliminated if they have equal or worse reconfiguration histories. The reconfiguration history of each point is maintained by the array *reconfiged*, which stores whether the point's assignment, so far, has involved a reconfiguration of the resource, *r*, on the current task type.

Until now, we have only considered the case where the reconfiguration cost is associated with the resource. The reconfiguration, in actuality, may be associated with the task; for example, to transmit data to a remote processor in a multiprocessor system, the reconfiguration cost is proportional to the size of the data. The only modification

to the algorithm is to store the reconfiguration cost for each task type rather than each resource.

**Theorem 2:** Assume we are given two points, p and q, representing the intermediate solutions of the dynamic programming algorithm. If p has a greater cost on all resources and has completed the same or less reconfigurations, then point p is dominated by the point q and hence will never be a part of the optimal solution.

**Proof:** Assume the contrary, that there is a problem instance where point p is part of the optimal solution. As point p and point q represent the same set of scheduled tasks, if point p was replaced with point q, the completion time on all the resources would decrease, and hence the makespan would decrease. Thus, the optimal solution could be improved, which is a contradiction. Hence point p will never be a part of the optimal solution. □

**Theorem 3:** The dynamic programming algorithm in Figs. 3 and 4 minimizes the makespan.

**Proof:** The algorithm carries out an exhaustive search of all possible task assignments. It only eliminates points from its consideration, when, provably, they will not lead to the optimal solution. □

The dynamic programming algorithm has an exponential worst-case time and space complexity of $O(M^N)$, where $M$ is the number of resources and $N$ is the number of tasks. Though the algorithm's space and time complexity make is prohibitory, it provides a solid foundation for the polynomial-time ε-approximation algorithm introduced in the next section.

## 5. APPROXIMATION ALGORITHM

The scheduling problem, as formulated in Section 2, is NP-complete, proven by a reduction from the well-known set-sub problem; unless it is somehow proven that P=NP, an optimal polynomial time algorithm for this scheduling problem does not exist [5].

The optimal dynamic programming algorithm described in the preceding section has an exponential worst-case time complexity. Using a trimming procedure, however, this algorithm can be converted into a fully polynomial-time ε -approximation algorithm. The approximation algorithm can find solutions that are a factor of at most 1+ε away from the optimal solution.

The underlying idea is that if two points in the M-dimensional space are sufficiently close, then it suffices to examine just one of them; although this sacrifices optimality, the tradeoff between runtime and solution quality can be tuned by the user's choice of ε.

A point, $o$, in the original space is not transferred to a new space if:

$$\frac{makespan_a}{(\delta+1)} \le makespan_o \le makespan_a, \qquad (4)$$

where $a$ is the point in the new space and δ the factor by which points are trimmed. It should be noted that δ is not the same as ε; δ and ε will be related to one another after some further discussion.

Three considerations are necessary when approximating points. First, when approximating between the assignment of tasks of the same type, the point in the new space must have a larger or equal approximation history; this is based on the same reasoning applied to the elimination of points, e.g., Fig. 1. Second, the makespan of the point that is eliminated must be approximately equal to the point that replaces it. Third, the cost along all axes must be approximately the same or greater for the point to be approximated.

The psuedocode for the approximation algorithm is the same as for the optimal algorithm, but with one additional modification. After removing all points that can be eliminated, the remaining points are approximated, using the procedure in Fig. 5. A call to this approximation procedure is placed between lines 30 and 31 in Fig. 3.

```
Approximate(NewPointsList)
1:  Sort points in NewPointsList in non-decreasing
order of makespan
2:  Place the first point in ApproxPointList
3:  for each point n in NewPointsList
4:       for each point a in ApproxPointsList
5:            if (n.reconfiged[r] > a.reconfiged[r] for
              any resource r and currentTaskType =
              nextTaskType)  or n.makespan >
              a.makespan*(δ+1) or (n.Cost[r] > a.Cost[r]
              and n.Cost[r] > a.Cost[r]*(δ+1) for all
              resources r)
6:                 add a to ApproxPointsList
7:            endif
8:       endfor
9:  endfor
10: return ApproxPointList
```

**Figure 5. Pseudocode for the approximation algorithm.**

The approximation algorithm starts off by adding the first point in *NewPointsList* to the newly formed list, named *ApproxPointList*. The remaining points are added to *ApproxPointList*, if a point does not already exist in *ApproxPointList* that approximates it.

The input to the scheduling algorithm is a collection of $N$ tasks, $M$ resources, and an approximation parameter ε,

where 0 < ε < 1. The algorithm returns a schedule whose makespan is worst than optimal by a factor of at most 1+ ε.

We set ε to be related δ by the following equation: δ = ε/2N, so that the following proof will hold.

**Theorem 4:** The approximation procedure presented in section 5.1 returns a schedule for the input tasks on the given resources, where the makespan of the schedule is 1+ε factor of the makespan of the optimal schedule.

**Proof:** The approximation algorithm introduces no error except in the trimming of points from the space. Thus we consider only that part of the algorithm.

Let $makespan_o$ be an optimal solution and $makespan_a$ be an approximate solution. We use an asterisk to denote the solution after the final iteration, e.g., $makespan_0*$.

To prove the approximation bound, we must formally porve that $makespan_a* \leq (1 + \varepsilon)makespan_0*$. Our approach uses inductions on the number of iterations. Specifically, we prove that for every point *o* in the original space at iteration *i*, there is a point *a* in the new space such that:

$$\frac{makespan_a}{\left(\frac{\varepsilon}{2N}+1\right)^i} \leq makespan_o \leq makespan_a \, . \tag{5}$$

This equality must hold for the final iteration as well, i.e.:

$$\frac{makespan_a *}{\left(\frac{\varepsilon}{2N}+1\right)^N} \leq makespan_o * \leq makespan_a * \, . \tag{6}$$

It follows that:

$$\frac{makespan_a *}{makespan_o *} \leq \left(\frac{\varepsilon}{2N}+1\right)^N . \tag{7}$$

Now, observe that:

$$\lim_{N\to\infty}\left(\frac{\varepsilon}{2N}+1\right)^N = e^{\varepsilon/2} \leq 1+\frac{\varepsilon}{2}+\left(\frac{\varepsilon}{2}\right)^2 , \tag{8}$$

Since 0 < ε < 1, it follows that:

$$1+\frac{\varepsilon}{2}+\left(\frac{\varepsilon}{2}\right)^2 \leq 1+\varepsilon \tag{9}$$

From which we can conclude:

$$\frac{makespan_a *}{makespan_o *} \leq 1+\varepsilon \tag{10}$$

This proves that the algorithm is an ε-approximation. □

Next, we prove that the algorithm runs in polynomial time.

**Theorem 5:** The approximation algorithm runs in polynomial worst-case time.

**Proof:** First, we need to bound the number of points in the space; this is accomplished by using a maximum value for the makespan, denoted by *T* (similar to Fig. 2). *T*, for example, could be the makespan of all of the tasks scheduled onto one single resource; or, alternatively, it could be the makespan computed by an efficient heuristic, such as list scheduling, that makes no guarantees of solution quality; the optimal makespan cannot exceed *T*.

Based on the approximation scheme, each point in the space must differ from the other points by at least $1 + \varepsilon/2N$ along all *M* axes in the space. Thus, the maximum number of points in the space, *P*, is given by:

$$P = \left(\lfloor log_{1+\varepsilon/2N} T \rfloor + 1\right)^M \times M! \tag{11}$$

$$= \left(\frac{lnT}{ln\left(1+\varepsilon/2N\right)}+1\right)^M \times M! \tag{12}$$

Since x/(1+x) ≤ ln(1+x), it follows that:

$$\leq \left(\frac{2N\left(1+\varepsilon/2N\right)lnT}{\varepsilon}+1\right)^M \times M! \tag{13}$$

$$\leq \left(\frac{4NlnT}{\varepsilon}+1\right)^M \times M! \tag{14}$$

Here, we assume that *M*, the number of resources, is a constant value. Therefore, *M!* is also constant, and the number of points in the space is proportional to 1/ε raised to a constant power. Since the complexity of the algorithm is polynomial in terms of the number of points in the space, and ε, the ε-approximation is fully polynomial-time. □

It is important to note that the runtime on the algorithm is heavily dependent on the number of resources, *M*, which is assumed to be a constant. If the approximation algorithm trimmed points according to their distance from the origin instead of makespan results in an ε-√2 approximation, but whose running time is less dependent on *M*.

## 6. EXPERIMENTAL RESULTS

This section evaluates the quality of the ε-approximation algorithm presented in this paper; it is compared against the optimal ILP formulation, as well as list scheduling, a well-understood greedy heuristic. The list scheduling heuristic processes tasks in the sorted order, and schedules the current task on the resource with the earliest finish time.

Subsection 6.1 applies the algorithm to scheduling parallel encryption tasks. Subsection 6.2 summarizes a theoretical study on a set of randomly generated data.

## 6.1 Application to Encryption

The first set of experiments schedules a parallelized encryption algorithm on a system with heterogeneous resources. The encryption algorithm used is the Advanced Encryption Standard (AES, a.k.a. Rijndael), which is a block cipher. The target system contains an Intel Pentium III general-purpose processor and a Virtex-E 1000e FPGA.

The system initially splits the data inputs, which are to be scheduled, into heterogeneous blocks with different encryption levels, i.e., three different key sizes: 128, 196, and 256 bits. The input data is also split into blocks of 128, 196, and 256 bits for encryption.

This yields nine distinct task types. The reconfiguration time of the FPGA for each task type was determined a-priori by implementing different versions of the AES and synthesizing it on the FPGA. The runtime on both the Pentium III and FPGA was determined by encrypting randomly-generated input data on each system.

We generated a suite of 1000 test cases with randomly generated input bitstreams to encrypt. We then scheduled each test case 3 times: using the optimal ILP formulation, using the approximation algorithm, and using list scheduling. For the approximation algorithm, we set the value of ε to 0.999, which means that the makespan of the approximate solution should be 1.999x greater than the makespan of the optimal solution in the worst case.

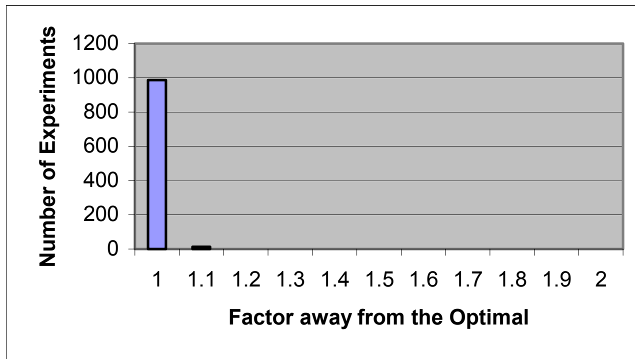
find an optimal solution, the makespan of the approximate solution was 1.1x worse than the optimal makespan. These results are far closer to the optimal solution than to the theoretical worst case.

Figures 6(b) shows that list scheduling performs much worse; in no case did list scheduling find an optimal solution. The makespans obtained from list scheduling range from 1.6x to 2.6x worse than optimal.

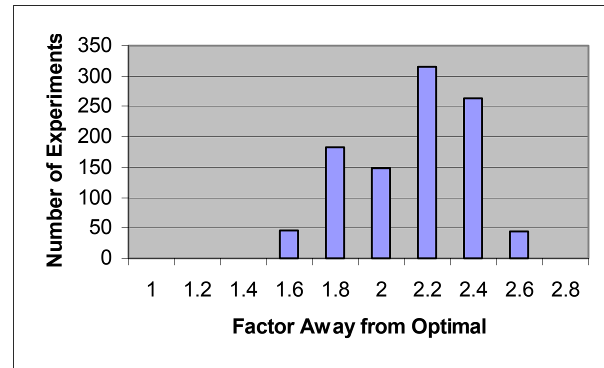
## 6.2 Random Input Data

This subsection presents an experimental study in which we vary the number of tasks and task types; unlike the preceding subsection, the set of tasks to be scheduled are generated randomly. First, we randomly generate a number of task types, and the number of tasks per type (which is assumed to be the same for all types); from this we can derive the total number of tasks to be scheduled; in each case, we randomly generate 1000 instances of the problem to solve.

Fig. 7 compares the ILP, ε-approximation algorithm, and list scheduling heuristic on these benchmarks with ε set to 0.999. Since ε must be between the values zero and one, a very large value for ε was chosen to give the list scheduling heuristic the best chance to be competitive.




(a)




(b)

**Figure 6. AES scheduling error distribution: (a) ε-approximation versus ILP; (b) list scheduling heuristic vs. ILP.**

Figs. 6(a) and (b) show the results of these experiments. Fig. 6(a) compares the approximation algorithm to the ILP, while Fig. 6(b) compares the list scheduling heuristic to the ILP. In both figures, the y-axis shows the number of experiments out of 1000, and the x-axis represents the maximum factor by which the makespan is different from the optimal value.

Fig. 6(a) shows that out of the 1000 test cases, the approximation algorithm found the optimal solution in the vast majority of cases; in the few cases where it could not

Fig. 7 shows that in the vast majority of cases, the ε-approximation algorithm found optimal solutions, while the list scheduling heuristic did not; furthermore, in a fair number of cases, the list scheduling heuristic found solutions that were worse than the theoretical guarantee of 1.999x of the ε-approximation algorithm. List scheduling faired best when there was only one type of task: in this case, reconfiguration occurs at most once per resource: an overly simplistic instance of the problem.

In Fig. 8, the value of ε is varied, and there are 3 task types with 3 tasks per type. For each choice of epsilon, the results are compared to the ILP. Also, we verify than no experiment results in a makespan (1+ε)x longer than optimal. The list scheduling heuristic, once again, produces many solutions whose makespan are more than (1+ε)x longer than the optimal solution.

In Fig. 9, we examine the effect of varying the number of resources. This work assumes a finite and limited number of resources, as the algorithm's complexity increases exponentially with respect to the number of resources. Fig. 9 experimentally demonstrates an increase in the number of resources improves the quality of the resulting schedules both for our algorithm as for the list scheduling heuristic, as would be expected by spreading out the tasks across resources. However, even with 100 resources, the list scheduling algorithm averages over 18% of the makespans as more the worst case theoretical bound imposed on the approximation algorithm.

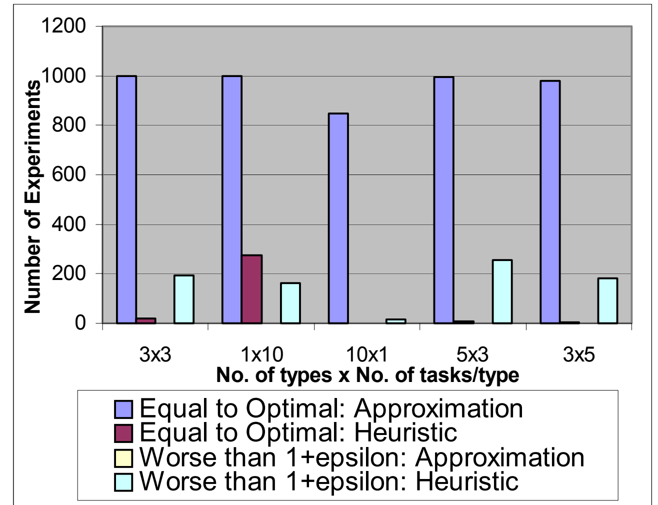



**Figure 7. Varying the number of tasks to be scheduled.**

## 7. EXTENSION TO TASK GRAPH SCHEDULING WITH COMMUNICATION

Until now, we have considered independent tasks. Our work, however, can be extended to handle the case where there are dependencies between tasks. This is in the case where we are given as input a data flow graph, where each node is a task and each edge represents a data dependency between the parent node and the child node. The graph, of course, is a precedence directed acyclic graph (DAG).

To extend out work, the basic idea is to levelize the graph and then run our algorithm on each level of the graph. There are a few additional issues that need to be considered. First, to handle the case when certain resources take longer to execute than others at a certain level, the execution time of the resources should be given as input to the next level's iteration, so the appropriate axis will be scheduled from a new starting point instead of from zero, or the origin.

Second, the reconfiguration status of the resources should be given as input to the next level, so that the reconfigurations can be accounted for properly. To ensure that tasks, whose parents have not completed their execution not be executed, tasks type orderings can be reshuffled to ensure that this constraint is not violated. In the worse case, some idle time will be scheduled.

There exist a cost and even a delay for moving data from one resource to another. These parameters can easily be considered with our algorithm. The cost of scheduling children tasks onto different resources can be added to the reconfiguration cost for that task, on all resources other than the one the parent task is scheduled on. Similarly, the delay can be accounted for by adding the delay to the execution time of the task.

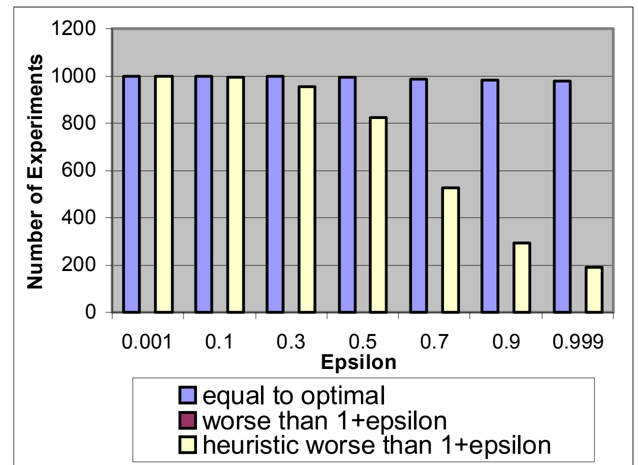



**Figure 8. Varying the value of ε.**

When working with precedence DAGs, communication between tasks can also be a constraint on the schedule. Fortunatly, communication cost between tasks can be incorporated into our algorithm, by adding the cost of communication to the task execution cost, for schedules which split the tasks onto different resources. We do not have a proof for whether this heuristic is an approximation algorithm for the problem of DAG scheduling. In the future, we hope to extend our work to address this more general model of tasks, and provide not just a heuristic for this problem but an approximation algorithm.

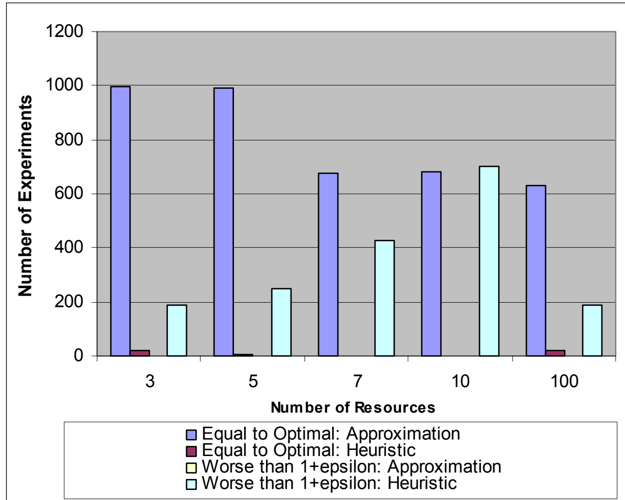



**Figure 9. Varying the number of resources to be scheduled.**

# 8. RELATED WORK

There exists extensive literature in the classic problem of scheduling independent tasks onto heterogeneous resources so as to minimize the makespan. Note this literature does not consider reconfiguration cost, the main thrust of this work. A linear programming approach is taken in both [11] and [13]. The problem is formulated as an integer linear programming problem. An LP-relaxation is used to assign tasks to resources. Up to *m*-1 tasks can be split among different resources, where *m* is the number of resources. To obtain a valid non-preemptive solution, each split task must be reassigned to one resource. This is done with complete enumeration as shown in [13]. Also, the values can be rounded as shown in [14]. This paper goes on to give a 2-approximation algorithm for the case of an unbounded number of resources

More recently, [11] has improved on the previous work, by using linear programming to schedule short tasks and dynamic programming to schedule long tasks. The determination of long versus short tasks is carried out according to the level of approximation as determined by epsilon.

In the related work where reconfiguration cost is considered several perspectives exists. [19] first introduced prefetching of a configuration profile to overlap the reconfiguration with execution in partially reconfigurable systems. [22] and [23] present heuristics that utilizes prefetching for reconfiguration aware scheduling. These approaches are orthogonal to our approach, as we do not consider the impact of prefeching in our work.

Reconfiguration cost in task execution is considered in the realm of regenerative energy sources in [5][20][21][26]. These works present heuristics for schedule task execution given the variability of energy availability.

Work on scheduling algorithms for reconfigurable resources that consider area and execution time together include [27]. Our approach is robust enough to be applied to this problem area..

An approximation algorithm is presented in [25] for problems with a delay-cost model, as with the problem presented in this paper. Unfortunately, the addition of reconfiguration cost to the model can only be done by mapping reconfiguration cost to the execution delay, which results in schedules with gaps in execution and potentially unnecessary additional reconfigurations. Similarly, [29] considers pareto optimal solutions for the case where there is a delay-cost tradeoff. The same limitation applies to this work as well.

In [14], a mixed integer linear programming based heuristic is proposed for our problem. The algorithm carries out an exponential branch and bound search of all possible solutions, but halts the search after a certain amount of time, regardless of whether the search has completed. In [18], preemption of reconfiguration is examined. In [28], and optimal algorithm is given for the related problem of minimizing reconfiguration delay. In [16], several heuristics are presented, which are adaptations of the pseudopolynomial solutions for the famous parallel machines, single server problem. In [17], several simple heuristics are presented for this and related problems.

This paper describes algorithms that do consider reconfiguration cost when scheduling independent tasks onto heterogeneous resources. The heuristic presented is an approximation algorithm whose solution quality can be bounded. Also, all the algorithms utilize only computational techniques, as opposed to linear programming techniques. The work presented in this paper is based on the preliminary work presented in [12].

# 9. CONCLUSION

In this paper, we have presented a fully polynomial approximation algorithm for determining the optimal scheduling of independent tasks onto heterogeneous resources with heterogeneous reconfiguration costs. The approximation is derived form the optimal algorithm, by trimming the number of points in the space.

The algorithm is shown both theoretically and experimentally to be a $(1+\varepsilon)$-approximation. Extensive verification of the algorithm utilizing random values was conducted. Additionally, a real world application to schedule the parallel encryption of tasks onto an FPGA and a general purpose processor was used to demonstrate that the approximation is capable of scheduling the encryption of heterogeneous inputs onto heterogeneous resources dramatically more effectively than list scheduling. Its solution quality also compares well with the optimal, though exponential, algorithm.

## 10. REFERENCES


[1] B. Schneier, *Applied Cryptography*. John Wiley and Sons, Inc, New York, 1996.

[2] T. H., Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, Cambridge, MA, 2001.

[3] J. Daeman, and V. Rijmen, AES Proposal: Rijndael. In *the Proceedings of First Advanced Encryption Standard Conference*, 1998.

[4] M. R. Garey, and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[5] I. Follcarelli, A. Acquaviva, A. Susu, T. Kluter, G. De Micheli, An Opportunistic Reconfiguration Strategy for Environmentally Powered Devices. In *The Proceedings of the 3rd ACM International Conference on Computing Frontiers* (CF '06), 2006.

[6] P. M. Heysters, G. J. M. Smit, E. Molenkamp, Energy-Efficiency of the MONTIUM Reconfigurable Tile Processor. In *The Proceedings of Engineering of Reconfigurable Systems and Algorithms* (2004).

[7] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., Boston, MA, 1997.

[8] E. Horowitz and S. Sahni, Exact and Approximate Algorithms for Scheduling Nonidentical Processors. *Journal of the Association of Computing Machinery* 23, 2 (April 1976), 317-327.

[9] Implementation of AES (Rijndael) in C/C++. http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm

[10] K. Jansen, and L. Porkolab, Improved Approximation Schemes for Scheduling Unrelated Parallel Machines. In *The Proceedings 31st ACM Symposium on Theory of Computing (STOC '99)*, 1999.

[11] J. K. Lenstra, D. B. Shmoys, and E. Tardos, Approximation Algorithms for Scheduling Unrelated Parallel Machines. *Mathematical Programming* 46, 1990.

[12] A. Nahapetian, S. Ghiasi,, and M. Sarrafzadeh, Task Scheduling on Heterogeneous Resources with Heterogeneous Reconfiguration Costs. In *The Proceedings of Parallel and Distributed Computing and Systems (PDCS): Special Session on Synthesis of Programmable Systems*, 2003.

[13] C. N. Potts, Analysis of a Linear Programming Heuristic for scheduling Unrelated Parallel Machines. *Discrete Applied Mathematics 10* , 1985.

[14] U. N. Shenoy, P. Banerjee, and A. Choudhary, A System-level Synthesis Algorithm with Guaranteed Solution Quality. In *The Proceedings of Design Automation and Test in Europe*, 2003.

[15] I. Verbauwhede, P. Schaumont, and H. Kuo, Design and Performance Testing of a 2.29-GB/s Rijndael Processor. *IEEE Journal of Solid-State Circuits* 38, 3 (March 2003), 569-572.

[16] J. Angermeier and J. Teich, Heuristics for Scheduling Reconfigurable Devices while Respecting Reconfiguration Overheads. In *The Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium (ISDPS '08)*, 2008.

[17] K. Bondalapati and V. Prasanna, Reconfigurable Computing Systems. In *The Proceedings of the IEEE 90* (7) (2002) pp.1201-1217.

[18] F. Dittmann and S. Frank, Hard Real-Time Reconfiguration Port Scheduling. In *The Proceedings of Design Automation Conference ( DAC '07)*, 2007.

[19] Z. Li and S. Hauck, Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. In *The Proceedings of International Symposium on FPGA*, 2002.

[20] C. Moser, d. Brunelli, L. thiele, and L. Benini, Real-Time Scheduling with Regenerative Energy. In *The Proceedings of 18th Euromicro Conference on Rea-Time Systems (ECRTS '06),* 2006.

[21] A. Nahapetian, P. Lombardo, A. Acquaviva, L. Benini, M. Sarrafzadeh, Dynamic Reconfiguration in Sensor Networks with Regenerative Energy Sources. In *The Proceedings of Design Automation and Test Europe (DATE '07)*, 2007.

[22] J. Resano, D. Mozos, D. Verkest, F. Catthour, and S. Vernalde, Specific Scheduling Support to Minimize the Reconfiguration Overhead of Dynamically Reconfigurable Hardware. In *The Proceedings of Design Automation Conference (DAC '04)*, 2004.

[23] J. Resano et al, Run-time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems. In *The Proceedings of FPL'03*, 2003.

[24] N. Moreano, E. Borin, C. C. de Souza, G. Araujo: Efficient datapath merging for partially reconfigurable architectures. *IEEE Trans. on CAD of Integrated Circuits and Systems* 24(7): 969-980 (2005)

[25] S. Roy, K. Belkhale, and P. Banerjee, An α-Approximate Algorithm for Delay-Constraint Technology Mapping. In *Proceedings of Design Automation Conference (DAC '99)*, 1999.

[26] C. Rusu, R. Melhem, and D. Mosse, Multi-Version Scheduling in Rechargeable Energy-Aware Real-Time Systems. In *Proceedings of Euromicro Conference on Rea-Time Systems (ECRTS '03),* 2003.

[27] K. Bazargan, R. Kastner, and M. Sarrafzadeh, Fast Template Placement for Reconfigurable Computing Systems. In *IEEE Design and Test – Special Issue on Reconfigurable Computing*, 17, 1 (Jan. 2000), 68-83.

[28] Soheil Ghiasi, Ani Nahapetian, Majid Sarrafzadeh. An Optimal Algorithm for Minimizing Runtime Reconfiguration Delay. *ACM Transactions on Embedded Computing Systems (TECS) Vol. 3, No 2, pp. 237-256, May 2004*.

[29] P. Yang and F. Catthoor, "Pareto-Optimization Based Run-Time Task Scheduling for Embedded Systems. In *Proceedings of International Symposium on Software Synthesis ( ISSS '03)*, 2003.

[30] S. Kogekar, S. Neema, and X. Koutsoukos, Dynamic Software Reconfiguration in Sensor Networks. In *Proceedings of the 2005 Systems Communications* (August 14 - 17, 2005).

[31] S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, A. Ledeczi, and M. Maroti, Constraint-guided dynamic reconfiguration in sensor networks. In *Proceedings of the Third international Symposium on information Processing in Sensor Networks (IPSN '04),* 2004.