Scheduling on Heterogeneous Resources with Heterogeneous Reconfiguration Costs

Ani Nahapetian Soheil Ghiasi Majid Sarrafzadeh

Computer Science Department
University of California, Los Angeles
Los Angeles, California

{ani, soheil, majid}@cs.ucla.edu

ABSTRACT

In this paper, we provide an optimal algorithm and a fully polynomial time approximation algorithm for the problem of scheduling independent tasks onto a fixed number of heterogeneous unrelated resources with heterogeneous reconfiguration costs. The notion of solution dominance is used to consider all or approximately all possible assignments of tasks to resources. To demonstrate the utility of the approximation algorithm, it is used to schedule blocks of data that are to be encrypted using the Rijndael encryption algorithm on a general-purpose processor and an FPGA. The results confirm the theoretical conclusions.

1. INTRODUCTION

Today, many parallel systems are composed of heterogeneous components, where the resources may have a reconfiguration cost associated with them. The reconfiguration cost can be the traditional cost associated with dynamically reconfiguring Field Programmable Gate Arrays (FPGAs). Also, it can represent the time cost of transmitting data and code to remote processors. Thus with these reconfigurable systems arise new scheduling problems, with a new reconfiguration cost constraint.

In this paper we present an optimal algorithm and a fully polynomial-time approximation algorithm for the problem of scheduling independent tasks onto heterogeneous resources with reconfiguration costs.

The algorithm is utilized to schedule blocks of data that are to be encrypted using the Rijndael encryption algorithm, where two heterogeneous resources, a processor and an FPGA, are available.

The rest of the paper is organized as follows. In section 2, the problem statement is formalized. This is followed an overview of independent task scheduling algorithms, in section 3. In sections 4 and 5, respectively, the optimal and approximation algorithm are presented and analyzed. Section 6 presents the experimental framework and the results. Finally section 7 concludes the paper.

2. PROBLEM STATEMENT

In this paper, we examine the question of assigning and scheduling independent tasks to a fixed number of heterogeneous unrelated resources with heterogeneous reconfiguration costs, in a non-preemptive manner, so as to minimize the final task completion time, referred to as the makespan. The tasks are independent in the sense that there exist no precedence relations; all the tasks can be executed in parallel. Each of the tasks can have different and unrelated execution times on each of the heterogeneous resources. Finally, there is a reconfiguration cost associated with each resource. The reconfiguration cost is incurred on a resource each time there is a switch between task types.

A task type is defined as follows. Tasks that incur no reconfiguration cost when executed after each other on a resource are of the same type. Tasks of different types would incur a reconfiguration cost between executions.

The algorithms presented in this paper are robust enough to also handle the case where the reconfiguration costs are associated with each task type instead of each resource. This is the model used for partial reconfigurations.

When examining the problem, the real issue is assigning tasks to resources. The optimal scheduling of the tasks, which are already assigned to the resources, is simply scheduling of the tasks of the same type one after each other. This schedule incurs the minimum amount of reconfiguration delay. Thus obtaining the optimal schedule is trivial after the assignment has been carried. The real issue is the assigning of the tasks to the available resources, assuming the optimal scheduling will be used.

3. RELATED WORK

The problem of scheduling independent tasks onto heterogeneous unrelated resources so as to minimize the makespan has been examined before. A linear programming approach is taken in both [7] and [8]. The problem is easily formulated as an integer linear programming problem. The binary constraint is relaxed, which results in a linear programming relaxation. After the tasks are assigned to the resources, up to m-1 tasks can be split among different resources, where m is the number of resources. There exist multiple resources for one task, since the relaxation allows tasks to be executed on multiple resources. To obtain a valid binary solution, the split tasks must be reassigned to one resource. This is done with complete enumeration as shown in [8]. Also, the values can be rounded as shown in [7]. This paper goes on to give a 2-approximation algorithm for the case of an unbounded number of resources

More recently, [6] 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.

The problem of independent task scheduling has been examined many times before. However, the emphasis has not been on resources or tasks with reconfiguration costs. None of the referenced methods can be applied to resources with either full or partial reconfiguration costs, nor can they be applied to systems that incur a delay when transmitting code and/or data to remote resources. The remainder of the paper describes algorithms that do consider reconfiguration cost when scheduling independent tasks onto heterogeneous resources.

4. OPTIMAL EXPONENTIAL SOLUTION

4.1 Example

Consider the following example. Given two resources, a general-purpose processor and an FPGA, with reconfiguration costs of 0 and of 10, respectively, schedule the tasks listed in Table 1.

Table 1. Tasks to Be Scheduled

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

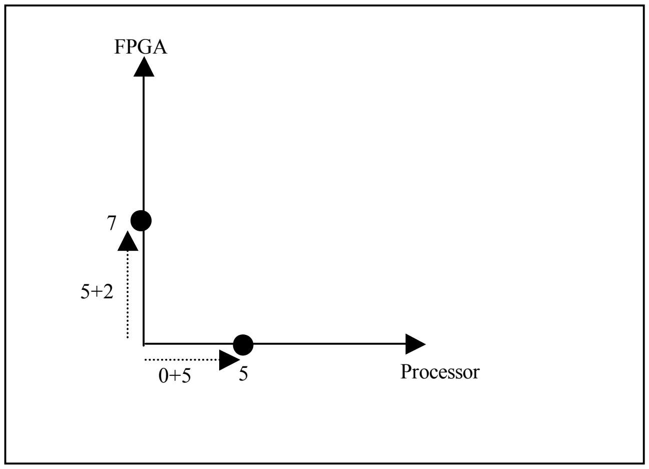

Figure 1. Assigning Task 1

As shown in Figure 1, we begin with a two dimensional space, where the x-axis represents the processor and the y-axis represents the FPGA. Then we begin by assigning the first task to both of the resources. In doing so we plot the points (0+4, 0) and (0, 5+2). Note that we include the reconfiguration cost in the calculations. Figure 2 demonstrates the assignment of task 2, to the already assigned task 1. Points (0,15), (4, 7), (5,8), and (9,0) are all plotted, while the previous points are discarded. At this point it is obvious that (5,8) will not be part of the optimal solution. Since the point (4,7) is clearly a better choice up to this point. Thus this point can be eliminated without compromising the optimality of the algorithm.

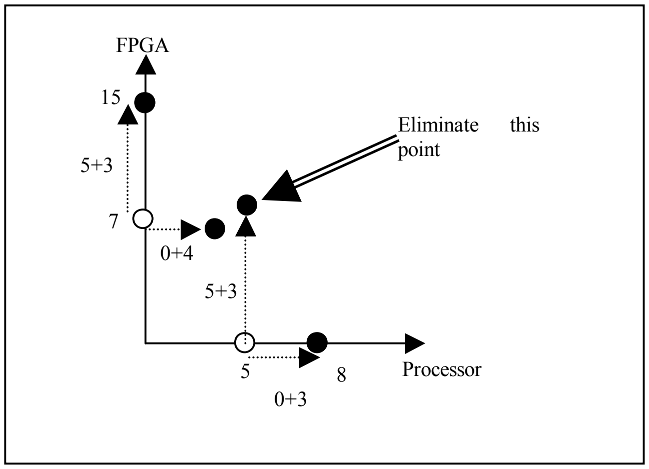

Figure 2. Assigning Task 2

4.2 Algorithm

As demonstrated in the example, an *m* dimensional space is created, where *m* is the number of resources. Points are plotted onto the graph, according to their execution times, or cost. 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. The psuedocode for this algorithm is given below.

Optimal Algorithm

```
1: Sort tasks to group together according to their type
2: for each tasks t
3:    ►Keep track of what group has been assigned
4:    if this task has a different type than the previous task
5:       currentTaskType = t.type
6:       for each old point o OldPointsList
7:          for each resource r
8:             o.reconfiged[r] = false
9:          end for
10:      end for
11:   end if
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.configed[r1] = o.configed[r1]
20:         end for
21          ►Add new costs to the point
22:         if n.configed[r] == false
23:             n.Cost[r]= n.Cost[r] + resource[r].ReconfigCost
24:             n.Cost[r] = true
25:         end if
26:         n.Cost[r] = n.Cost[r] + t.Cost[r]
27:         add n to list of NewPointsList
28:       end for
29:   end for
30:   RemoveEliminatablePoints(NewPointsList)
31:   OldPointsList = NewPointsList
32: end for
33: return assignment with the minimum makespan
```

The algorithm proceeds as follows. First, the tasks are sorted according to their type, to allow for concise record keeping of reconfigurations. 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 sun 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 the reconfiguration cost has not yet been incurred on the current task.

In the pseudocode, the position of the point in the space is represented by the array named *Cost*. *Cost* stores the cost incurred on each resource. In other words, *Cost* stores the distance of the point from the origin along each axis.

The reconfiguration history for each point on each resource is maintained using the variable *reconfiged*. At 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 can simply be 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. The psueodcode for this elimination procedure is given here.

```
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
4:             if n.configed[r] >= p.configed[r] for all resources r
or currentTaskType <> nextTaskType
5:                 remove n from newPointsList
6:             end if
7:     end for
8: end for
```

Consider the case where all the points of one type have been plotted, but before any of the points of the following type have been plotted. The points that are larger in all dimensions can be eliminated. The larger points will not be a part of the optimal solution, because a better assignment exists up to that point. Thus eliminating these points will not diminish the quality of the solution.

In between plotting points for tasks of the same type, a greater cost is not enough to eliminate a point. Points with a larger cost may have one, because of an already incurred reconfiguration cost. If a point is larger than another point, which 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 *configed*. The *configed* array stores whether the point's assignment, so far, has involved a reconfiguration of the resource, *r*, on the current task type.

Up until now, we have only considered the case where the reconfiguration cost is associated with the resource. There exist many cases where the reconfiguration is associated with the task. For example, when considering data transmission delay to a remote processor, the reconfiguration cost is proportional to the size of the data. The algorithms presented in this paper are robust enough to handle both of these situations and even a combination of the two. The only change is that the reconfiguration cost is stored for each task type instead of for each resource.

Theorem 1: The algorithm presented in this section schedules the input tasks, such that the makespan is optimally minimized.

Proof: The algorithm carries out an exhaustive search of all of the possible assignment of tasks. The algorithm only eliminates points from its consideration, when they will obviously not lead to the optimal solution. Thus the algorithm produces an optimal solution. □

The algorithm, however, has an exponential time and space complexity on the order of $O(m^n)$, where *m* is the number of resources and *n* is the number of tasks.

5. APPROXIMATION ALGORITHM

The problem as formulated in section 2 is an NP-complete problem, proven by reduction to the well-known set-sum problem. Therefore, an optimal polynomial time algorithm does not exist, unless P=NP[3].

The optimal algorithm expressed above is an exponential algorithm. By using a trimming procedure, however, a fully polynomial approximation algorithm can be produced. Proportional to the amount of time given, the approximation algorithm can find solutions a factor of ε away from the optimal solution.

5.1 Algorithm

As shown below the only difference between the optimal algorithm and the approximation algorithm is the addition of an approximation scheme.

Approximation Algorithm

```
1: Sort tasks to group together according to their type
2: for each tasks t
3:    ► Keep track of what group has being assigned
4:    if this task has a different type than the previous task
5:       currentTaskType = t.type
6:       for each old point o OldPointsList
7:          for each resource r
8:             o.reconfiged[r] = false
9:          end for
10:      end for
11:   end if
12:   ► Add m new points to the graph for each old point
13:   for each old point o in list of old point
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.configed[r1] = o.configed[r1]
20:         end for
21          ► Add new costs to the point
22:         if n.configed[r] == false
23:            n.Cost[r]= n.Cost[r] + resource[r].ReconfigCost
24:            n.Cost[r] = true
25:         end if
26:         n.Cost[r] = n.Cost[r] + t.Cost[r]
27:         add n to list of NewPointsList
28:      end for
29:   end for
30:   RemoveEliminatablePoints(NewPointsList)
31:   ► The only change to the code
32:   Approximate(NewPointsList)
33:   OldPointsList = NewPointsList
34: end for
35: return assignment with the minimum makespan
```

The idea is that if two values are close enough together, then in an approximation scheme, only one of them needs to be further examined. A point, *o*, in the original space is not transferred to a new space if

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

where *a* is the point in the new space and δ the factor by which points are trimmed.

When approximating points, two considerations need to be made. The first is that the latest finish time of the points *o* and *a* are on the same resource. This requirement is in place, so that points with the same makespan, but in two very different parts of the space do not approximate each other. Second, when approximating in between the assignment of tasks of the same type, the point in the new space needs to have a larger or equal approximation history. This is based on the same reasoning applied to eliminating points.

The psuedocode for the approximation scheme is given below.

Approximate(NewPointsList)

```
1: Sort points in NewPointsList in non-increasing order of makespan
2: Place the first point in ApproxPointList and remove it from NewPointsList
3: for each point n in NewPointsList
4:    for each point p in ApproxPointsList
5:       if makespan of n and p is due to different resources
6:          continue
7:       end if
8:       if n.configed[r] > p.configed[r] for at least one resources r and currentTaskType = previousTaskType
9:          continue
10:      end if
11:      if n.makespan*(δ+1) < p.makespan
12:         continue
13:      end if
11:      remove n from NewPointsList
12:      break
13:   end for
14:   if n not removed from NewPointsList
15:      add n to ApproxPointsList
16:      remove n from NewPointsList
17:   end if
18: end for
19: NewPointsList = ApproxPointList
```

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 them.

5.2 Proof

The approximation procedure presented in the previous section, takes as input *n* tasks, *m* resources, and an approximation parameter ε, where 0 < ε < 1. It returns a schedule for the tasks on the resources, where the makespan of the schedule is 1+ ε factor of the makespan of the optimal schedule.

The ε value is related to the previously discussed δ by the following equation: $\delta = \frac{\varepsilon}{2n}$.

Theorem 2: 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.

Since $makespan_o^* \le makespan_a^*$, we need to show that $\frac{makespan_a^*}{makespan_o^*} \le 1+\varepsilon$, where the subscript *a* represents the approximation solution, the subscript *o* represents the optimal solution, and the asterisk represents the solution of the final iteration.

By induction on *i*, it can be shown that for every point *o* belonging to in the original space at iteration *i*, there is a point, *a*, belonging to the new space in iteration *i*, such that

$$\frac{makespan_a}{(\frac{e}{2n}+1)^i} \le makespan_o \le makespan_a$$

.

This inequality must hold for the final iteration also.

$$\Rightarrow \frac{makespan_a^*}{(\frac{\varepsilon}{2n}+1)^n} \le makespan_o^* \le makespan_a^*$$

$$\Rightarrow \frac{makespan_a^*}{makespan_o^*} \le (\frac{\varepsilon}{2n}+1)^n$$

$$\Rightarrow \frac{makespan_a^*}{makespan_o^*} \le e^{\varepsilon/2} \text{as n} \to \infty$$

$$\Rightarrow \frac{makespan_a^*}{makespan_o^*} \le 1+\frac{\varepsilon}{2}+(\frac{\varepsilon}{2})^2$$

because $e^{\varepsilon/2} \le 1+\frac{\varepsilon}{2}+(\frac{\varepsilon}{2})^2$

$$\Rightarrow \frac{makespan_a^*}{makespan_o^*} \le 1+\varepsilon$$ because 0<ε<1 □

Now, that we have proven the quality of the algorithm, we must prove that the algorithm is indeed a polynomial-time algorithm.

Theorem 3: The approximation procedure presented in section 5.1 is a fully polynomial-time algorithm.

Proof: First we bound the number of points in the space. We do this by using a maximum value for the makespan, M. M can simply be the makespan of all the tasks scheduled onto one of the resources. The optimal solution will obviously be smaller than or equal to this value. Previously we stated that each point with a maximum finish time on the same resource must differ by at least a factor of 1+ε/2n. Therefore, each space will contain at most $(\lfloor \log_{1+\varepsilon/2n} M \rfloor + 1)\times m$ points.

$$(\log_{1+\varepsilon/2n} M + 1)\times m = (\frac{\ln M}{\ln(1+\varepsilon/2n)}+1)\times m$$

$$\le (\frac{2n(1+\varepsilon/2n)\ln M}{\varepsilon}+1)\times m$$

$$\le (\frac{4n\ln M}{\varepsilon}+1)\times m$$

The number of points in the space is polynomial in the size of the input and 1/ε, since the number of resources is supposed to be a fixed value. Since the complexity of the algorithm is polynomial in terms of the number of points in the space and ε, the approximation algorithm presented in section 5.1 is a fully polynomial-time approximation scheme. □

6. EXPERIMENTAL RESULTS

6.1 Experimental Framework

The value of the scheduling algorithms presented in this paper was evaluated experimentally. The goal of the experiments was to schedule the encryption of heterogeneous inputs using the Rijndael encryption algorithm. Provided for the encryption was a general-purpose processor and an FPGA, with a known full reconfiguration cost.

Encrypting input using the Rijndael algorithm is highly parallelizable. Thus the inputs are broken up into blocks, which are then to be encrypted on the parallel resources. The inputs are split into blocks of three different sizes: 128 bits, 196 bits, and 256 bits. Also, depending on the security and time requirements there are three key sizes: 128 bits, 196 bits, and 256 bits. The results are nine different types of tasks. For switching among the different configuration of block sizes and key sizes, a reconfiguration cost needs to be paid on the FPGA.

Three scheduling algorithms were used in the experiments. The optimal and approximation algorithms were compared with the first-available heuristic algorithm. The first-available algorithm schedules the current task on the resource with the earliest finish time at that point. It is a simple algorithm, chosen for the purposes of comparison.

Sample values were chosen for the execution times and the reconfiguration cost. The results of these experiments are discussed in the next section.

6.2 Experimental Results

Figure 3 demonstrates that the approximation algorithm, where ε = .9, is indeed at most a factor of 1+.9 away from the optimal solution. The y-axis shows the number of experiments out of a 1000, and the x-axis represents the maximum factor by which the approximation makespan was different from the optimal value. As shown in the graph, all of the thousand test cases resulted in schedule with a makespan of at most 1.9 times the optimal solution.

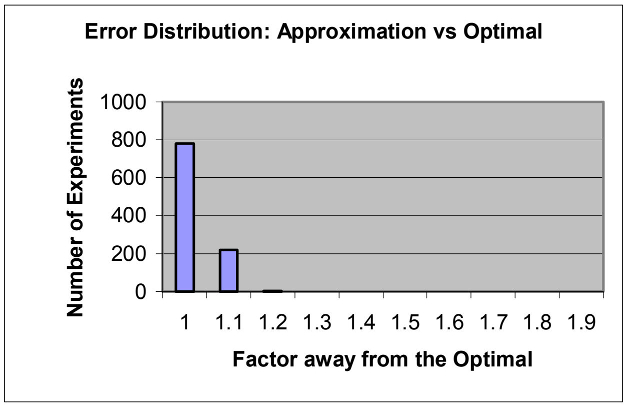

Figure 3.

Figure 4 gives the error distribution of the makespan using the first-available heuristic versus using an optimal algorithm. The results using the first-available heuristic are up to 5.7 times the optimal value. These results reinforce the quality of the approximation algorithm.

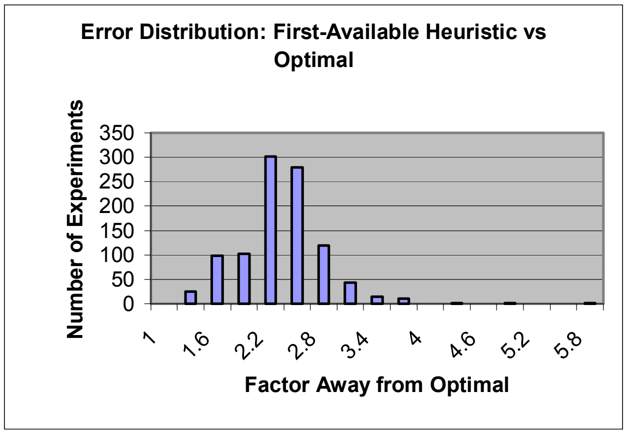

Figure 4.

7. 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+ε)-approximation. It is capable of scheduling the encryption of heterogeneous inputs onto heterogeneous resources, dramatically more effectively than the first-available heuristic. Its quality of solution also compares well with the optimal exponential algorithm.

8. REFERENCES

[1] Cryptography Technology. http://fp.gladman.plus.com/crypotography technology/index.htm

[2] Daeman, J., and Rijmen, V. AES Proposal: Rijndael. In First Advanced Ecryption Standard Conference (1998).

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

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

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

[6] Jansen, K., and Porkolab, L. Improved Approximation Schemes for Scheduling Unrelated Parallel Machines. In Proceedings 31st ACM Symposium on Theory of Computing (STOC '99), 408-417.

[7] Lenstra, J.K., Shmoys, D.B., and Tardos, E. Approximation Algorithms for Scheduling Unrelated Parallel Machines. Mathematical Programming 46 (1990), 259-271.

[8] Potts, C.N. Analysis of a Linear Programming Heuristic for scheduling Unrelated Parallel Machines. Discrete Applied Mathematics 10 (1985), 155-164.

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