Effective Task Scheduling for Embedded Systems Using Iterative Cluster Slack Optimization

Jongdae Kim, Sungchul Lee, Hyunchul Shin

Electrical and Computer Engineering, Hanyang University, Ansan Kyunggi-do, South Korea
Email: kbigbell@nate.com, sclee@digital.hanyang.ac.kr, shin@hanyang.ac.kr

Received August 7, 2013; revised September 7, 2013; accepted September 14, 2013

ABSTRACT

To solve computationally expensive problems, multiple processor SoCs (MPSoCs) are frequently used. Mapping of applications to MPSoC architectures and scheduling of tasks are key problems in system level design of embedded systems. In this paper, a cluster slack optimization algorithm is described, in which the tasks in a cluster are simultaneously mapped and scheduled for heterogeneous MPSoC architectures. In our approach, the tasks are iteratively clustered and each cluster is optimized by using the branch and bound technique to capitalize on slack distribution. The proposed static task mapping and scheduling method is applied to pipelined data stream processing as well as for batch processing. Furthermore, an energy-aware task mapping and scheduling algorithm based on our cluster slack optimization is developed. Experimental results show improvement in latency, throughput and energy.

Keywords: Multi-Processor; Mapping; Scheduling

1. Introduction

As many systems in communication and multi-media become more and more complex and require a large amount of computation, a single processor can’t frequently satisfy the performance criteria. Since designing a high performance custom integrated circuit requires great cost and a long design period, using multiple processors (or multi-cores) is increasingly seen as an alternative approach. However, mapping of applications to MPSoC architectures and scheduling of tasks are key problems in high-level design of embedded systems.

System performance or throughput can be improved by using parallel processing or pipelining. In parallel processing, multiple functions can be executed in parallel or a single function can be executed for multiple data sets. In pipelining, consecutive data streams are processed with overlaps in processing time for larger throughput. Data streams of audio and video signals should be processed within appropriate interval between consecutive inputs to provide the correct information for users. The most important factor to decide the performance of a pipeline is the interval between the consecutive inputs, and this interval can be fixed or varied. A small interval means a higher throughput.

Mapping and scheduling problems for multiple processors belong to the class of NP-hard problems [1]. Research has been carried out in solving task mapping and scheduling problems. Different tools and models have been developed to address the problems above.

The branch and bound algorithm can be used to map an application to architecture so that the total execution time is minimized. This algorithm checks all possible cases of assignments of n tasks to m processors leading to huge search space and minimizes the search area by using bounding procedure to reach an optimum solution in less time [2]. However, due to the worst-case exponential complexity, this method can only be used for “small” sized problems.

2. Related Works

Mapping and scheduling problems for multi processors belong to the class of NP-hard problems [1]. We consider Problem Model like the one described in [2] where n tasks are to be allocated to m heterogeneous processing
elements (PEs). The mapping has to be found under constraints where all tasks are assigned to PEs and every task is assigned to only one PE. The task scheduling problem is the problem of assigning the order of tasks in the system to optimize the overall performance for the given application.

A very popular heuristic method to schedule tasks is list scheduling [3]. Tasks are scheduled from a prespecified list and the task with the highest priority is removed from the priority list and then scheduled, one by one.

For scheduling heterogeneous processing elements, the Heterogeneous Earliest Finish Time (HEFT) algorithm [4] is used. This algorithm uses upward rank value which represents the longest path from each task to set the priority of each task. The task list is ordered by decreasing value of upward rank. Based on this list, tasks are scheduled onto the processors that have the earliest finish time. Another scheduling algorithm based on list scheduling is the Critical-Path-On a Processor (CPOP) algorithm [4]. In contrast to the HEFT algorithm, this algorithm uses upward ranking and downward ranking to set the priority of each task. The downward ranking is computed by adding the average execution time of the task and maximum downward rank value of the predecessors. The task with highest priority is selected for execution. If the selected task is on a critical path, then it is scheduled for the critical path processor. A critical processor is one that minimizes the cumulative computation cost of the tasks on the critical path. Otherwise, the task is assigned to the processor which has the minimum execution finish time of the task.

Similarly, Performance Effective Task Scheduling (PETS) algorithm [5] consists of three stages, Level sorting, Task prioritization, and Processor selection. In Level sorting, each task is sorted at each level in order to group the tasks. Because of this, tasks in the same level could be executed in parallel. In Task prioritization, the priority of each task is computed by using average computation cost, data transfer cost, and highest rank of the predecessors. In Processor selection, task is assigned to the processor which gives the minimum finish time. Compared to the Heft and CPOP algorithm, PETS algorithm shows good performance.

The force-directed scheduling algorithm [6] prioritizes and schedules the subtasks based on the resource utilization probability in each step. Recently, other techniques, such as simulated evolution based method [7], were reported. The genetic algorithm [8,9] is a widely researched random search technique. This algorithm executes in generations, producing better solutions using crossover and mutation operators. In the genetic algorithm, candidate solutions are represented by sequences of symbols called chromosomes. Evolutionary algorithms (EAs) operate on a population of potential solutions, applying the principle of survival of the fittest to produce successively better approximations to a solution. At each generation of the EAs, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and reproducing them using variation operators. This process may lead to the evolution of populations of individuals that are better suited to their environment than the individuals from which they were created, just as in natural adaptation. EAs are characterized by the representation of the individual, the evaluation function representing the fitness level of the individuals, and the population dynamics such as population size, variation operators, parent selection, reproduction and inheritance, survival competition method, etc. To have a good balance between exploration and exploitation, those components should be designed properly [10].

Like all other EAs, a quantum-inspired evolutionary algorithm (QEA) also consists of the representation of individuals, the evaluation function, and the population dynamics. The only difference is that it uses quantum bits as probabilistic representation for individuals instead of binary representation of genes [7].

In [11], energy aware scheduling that uses task ordering and voltage scaling in an integrated manner is presented, and the voltage level instead of the speed level was considered in the cost function for each processor. Dynamic voltage scaling (DVS) is one of the most powerful techniques to reduce energy consumption [12,13]. All these methods are heuristic approaches.

3. Mapping and Scheduling for Batch Processing Systems

By using the branch and bound algorithm [2], one can obtain an optimum solution for “small” mapping problems. However, one cannot afford run-time (CPU time) for large problems due to the exponential worst case run-time. To solve the complex mapping problems in a reasonable time, we propose a cluster slack optimization algorithm. In this algorithm, we divide all tasks in a task flow graph by a given number of clusters. Each cluster is optimized to find a best solution capitalizing on the slack distribution. The algorithm to minimize the execution (finish) time can be written as in Algorithm 1.

The cluster slack optimization algorithm consists of five major steps, which are initialization, As Late As Possible (ALAP) scheduling, making clusters, cluster selection, and slack optimization for the selected cluster.

In the initialization step, we find an initial solution by using a greedy list scheduling method. In the ALAP scheduling step, we reschedule all tasks by using ALAP scheduling to maximize the slacks in the optimization zone to capitalize the slacks during optimization. By
Algorithm 1. Cluster slack optimization algorithm for task mapping and scheduling.

Input : task flow graph, execution time table, processors  
Output : mapped and scheduled tasks

Find an initial greedy solution;  
while ( solution can be improved ) do  
ALAP scheduling for slack maximization;  
Make clusters;  
while ( not completed ) do  
Select a cluster to optimize;  
Find the task optimization zone for the selected cluster;  
Optimize the tasks in the selected cluster within the task optimization zone;  
end  
end  
Output the solution;

clustering the given tasks into several clusters in making cluster step, only a small number of tasks in a cluster are remapped and re-scheduled, and thus the run-time can be controlled by adjusting the cluster size. In the cluster selection step, an un-processed task cluster is selected in order of execution start times. In the optimization step, we optimally remap and reschedule the tasks in the selected cluster within the task optimization zone. The task optimization zone is defined by two boundaries called floor and ceiling. When the optimization of the current cluster is finished, the floor and the ceiling will be updated and thus the task optimization zone is moved. The optimization is repeated when the new solution is better than the previous best solution.

Figure 1 shows an example from [5], in which 10 tasks are to be mapped to 3 heterogeneous processors. Figure 1(a) shows the task flow graph and Figure 1(b) shows the execution time table from [5]. For example, the execution of task T1 by processor P1 takes 14 cycles. The communication time to transfer the output of task 1 to the input of task 2 takes 18 cycles as shown by the edge weight. Different processors may take different number of cycles to finish a given task. Figures 1(c)-(e) show the scheduling results by using the HEFT, CPOP and PETS algorithm respectively [5]. The produced solutions use 80, 86, and 79 cycles, respectively.

Now we explain our method by using the same example. Figure 2(a) shows the initial solution by using greedy list scheduling and Figure 2(b) shows the solution after the ALAP scheduling step. The produced initial solution used 82 cycles.

After the ALAP scheduling step, all tasks are clustered in the order of execution start time, and each cluster is re-mapped and re-scheduled. We perform the branch and bound by evaluating the slacks of all possible mappings and schedules for tasks in the current cluster. The objective of slack optimization is to maximize the minimum slack for each processor. Slack optimization tries to achieve a near global optimum solution by solving iterative local optimization problems. Figure 3 shows an example process of slack calculation. Let ST i be starting time of task i and FT i be finishing time of task i. Let C a,b be communication time from n a to n b and E i,j be execution time of task i on processor j. Since n a, n b, and n d are executed on P1, the communication times C a,i and C i,d can be 0. Now, we define the slack when the task n i is mapped to P1, as follows.

\[
\text{Slack}_i = \min \left[ ST_d - \max \left( FT_a + C_{a,j}, FT_b + C_{b,j} \right) + ET_{j,i} + C_{i,d} \right], \\
ST_f - \max \left( FT_a + C_{a,j}, FT_b + C_{b,j} \right) + ET_{j,i} + C_{i,d}
\]

The goal of the optimization is to remap/re-schedule to maximize the slack, since the minimum slack of all output tasks in the current cluster is the gain of the optimization. In other words, the latency can be reduced by the minimum slack.

Figure 4 shows the process of our algorithm for the example shown in Figure 1. After ALAP scheduling, all tasks are clustered to form 3 clusters as shown in Figure 4(a). In this case, 1st_cluster = \{n1, n2, n4, n5\}, 2nd_cluster = \{n3, n6, n7\}, and 3rd_cluster = \{n8, n9, n10\}. Figures 4(b)-(d) show optimized results after optimization of each cluster. In Figure 4(b), the slacks of each processor are 13, 19, and 7 cycles, respectively. So we can reduce 7 cycles after the 1st cluster optimization. In Figure 4(c), there is no change during the 2nd cluster optimization. In the same way, we can reduce 1 cycle after the 3rd cluster optimization. Finally, the solution is reduced to 74 cycles, and 8 cycles are reduced when compared to the initial solution. This is really the optimum solution, as verified by using the branch and bound algorithm. Compared to the three previous algorithms shown in Figure 1, our new iterative slack optimization algorithm produced a significantly better solution (the latency has been reduced to 74 cycles from 79 cycles or more), which is the optimum solution in this case.

4. Mapping and Scheduling of Pipelined Systems for Data Stream Processing

4.1. Throughput Increase by Pipelining

Figure 5(a) shows the task flow graph with 15 tasks, and Figure 5(b) shows the execution time table for three processors (P1, P2, P3). For example, execution of task T1 by processor P1 takes 5 cycles. The edge weights
Figure 1. An example and scheduling results. (a) Task flow graph; (b) Execution time table; (c) The HEFT algorithm [4]; (d) The CPOP algorithm [4] (e) The PETS algorithm [5].

show the data communication times. Different processors may take different numbers of cycles to finish a given task.

In batch-mode task parallel execution, the goal is to minimize latency. However, continuous data stream processing is needed to process audio and video signals. Therefore, the most important factor of scheduling for data stream processing is to reduce the interval between consecutive data inputs for larger throughput. A small interval means higher throughput. To reduce data input interval (DII), we divide all tasks into several stages and use pipelining. For example, three candidate schedule positions of task 5 (T5) of the task flow graph shown in Figure 5(a) are shown in Figure 6(a) for the batch-mode task parallel execution. For pipelined execution, data can be processed by forming several stages. When the first stage processes i-th input data, the second stage process (i-1)-th input data. This pipelined scheduling allows 7 candidates schedule positions for T5, as shown in Figure 6(b).

4.2. Optimization Using a Cost Function

Usually, the throughput can be increased by increasing the number of stages. However, this may increase the latency and memory cost. To optimize the trade-off, we define the following cost function.

$$\text{Total Cost} = \alpha \cdot \text{DII cost} + (1 - \alpha) \cdot \text{Memory cost}$$  \(1\)

DII_cost is inversely proportional to the throughput and Memory_cost represents the cost of memory to store intermediate data among stages. The Memory_cost can
be computed as follows.

\[ \text{Memory\_cost} = \sum_{i=1}^{N} (M_i \times S_i) \]  

(2)

where \( N \) is the number of stages, \( M_i \) is the amount of memory for \( i \)-th stage, and \( S_i \) is the number of pipeline stages to store the data of \( i \)-th stage.

Figure 7 shows the pipelined schedule results of the task flow graph given in Figure 5. The clustering results are shown in Figures 7(b) and (c) when \( \alpha = 0.7 \) and \( \alpha = 0.3 \), respectively. Five stages (A, B, C, D, E) are used when \( \alpha = 0.7 \) as shown in Figure 7(b) and four stages (A, B, C, D) are used when \( \alpha = 0.3 \) as shown in Figure 7(c). The result of batch processing is shown in Figure 7(d), in which the data input interval (DII) is 49. Figure 7(e) shows the pipelined schedule using 5 stages with DII of 34.

When the number of stages is 5, the latency can be up to \( 5 \times \text{DII} \) (\( 5 \times 34 = 170 \) cycles), even though one set of input data can be processed every 34 cycles. Figure 7(f) shows the pipelined schedule using 4 stages with DII of 37. When the number of stages is 4, the latency can be up to \( 4 \times \text{DII} \) (\( 4 \times 37 = 148 \) cycles). Memory_cost is 44 and 28 when \( \alpha \) is 0.7 and 0.3, respectively. When \( \alpha \) is small then the weight of Memory_cost in (1) is large and thus memory cost is reduced even though DII can be increased. This shows that \( \alpha \) can be used to optimize the trade-off between DII and memory cost. Figure 8 shows how the data streams flow through the pipelining stages.

5. Energy Aware Mapping and Scheduling

Over the last decade, manufacturers have competed to advance the performance of processors by raising the clock frequency. However, recent computer systems are focused on battery-driven devices such as portable handheld devices, sensors, and robots, rather than traditional large devices and desktops. Therefore, technical issues are miniaturization and low energy consumption. Specially, low power is extremely important for many real-time embedded systems. To apply our iterative slack optimization algorithm for energy aware mapping and scheduling, we only need to modify the cost function in slack optimization. The modified cost we used is shown in (3). The energy-aware cluster slack optimization algorithm is shown in Figure 9.

\[ \text{Cost}_e = \beta \text{latency} + (1 - \beta) K \times \text{energy} \]  

(3)
Figure 7. (a) Task flow graph; (b) Pipeline stage ($\alpha = 0.7$); (c) Pipeline stage ($\alpha = 0.3$); (d) Scheduling result for non-pipeline system; (e) Scheduling result for pipeline system ($\alpha = 0.7$); (f) Scheduling result for pipeline system ($\alpha = 0.3$).
Figure 8. Processing procedure of an input data stream.

where $\beta$ is the weighting factor and $K$ is the scaling constant.

6. Experimental Results

We implemented the cluster slack optimization algorithm by using the “C” programming language under the Windows operation system. The experiments were performed by using both real applications and randomly generated task flow graphs with 20 to 100 tasks. The parallel Gaussian elimination, LU decomposition [9] and molecular dynamics [14] are used as real applications. The tasks are mapped and scheduled by our method and by branch and bound, HEFT [4], CPOP [4] and PETS [5] algorithms for comparisons.

Table 1 shows the solution comparisons for real applications. A base algorithm for comparisons is the branch and bound algorithm which can find an optimum solution by spending much more time than other algorithms.

However, in molecular dynamics applications, we use our algorithm as a base algorithm because the branch and bound algorithm cannot obtain the solution within 24 hours.

Table 2 shows the solution comparisons for the randomly generated task flow graphs. In experiments, we used 5 randomly generated task graphs for each number of tasks (# tasks). The results of our algorithm are obtained by taking up to 20 tasks in a cluster.

Table 3 shows the solution comparisons for real applications. A base algorithm for comparisons is the branch and bound algorithm which can find an optimum solution by spending much more time than other algorithms.

Table 2 shows the solution comparisons for the randomly generated task flow graphs. In experiments, we used 5 randomly generated task graphs for each number of tasks (# tasks). The results of our algorithm are obtained by taking up to 20 tasks in a cluster.

Table 3 shows DII_cost and Memory_cost for QEA and our method. In Equation (1), $\alpha = 0.7$ was used. On the average, our method shows better results. DII_cost is improved by 2% and Memory_cost is improved by 23%.

Table 4 shows tradeoff between DII_cost and Memory_cost by changing the weight $\alpha$. As $\alpha$ gets smaller, DII increases and Memory_cost decreases.

Energy aware mapping and scheduling experiments are processed by using randomly generated task flow graphs with 20 to 50 tasks. We use three commercial processor power models consisting of XScale, PowerPC, and DSP [15] for power estimation. The tasks are mapped and scheduled by using our method and the results are compared with those of Energy Gradient-based Multiprocessor Scheduling (EGMS) [11].

Tables 5 and 6 show the energy consumption comparisons when $\beta$ is 0.3 and 0.7, respectively. In experi-
Table 1. Solution comparisons for real applications.

<table>
<thead>
<tr>
<th>#tasks</th>
<th>CPOP</th>
<th>HEFT</th>
<th>PETS</th>
<th>Ours</th>
<th>Branch and Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Gaussian elimination (15 tasks)</td>
<td>120.7%</td>
<td>112.2%</td>
<td>109.7%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>LU decomposition (14 tasks)</td>
<td>113.1%</td>
<td>111.9%</td>
<td>103.5%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Molecular dynamics (40 tasks)</td>
<td>120.4%</td>
<td>117.6%</td>
<td>109.6%</td>
<td>100%</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Table 2. Solution comparisons for randomly generated task flow graphs with 20 to 100 tasks.

<table>
<thead>
<tr>
<th>#tasks</th>
<th>CPOP (100%)</th>
<th>HEFT (100%)</th>
<th>PETS (100%)</th>
<th>Ours (100%)</th>
<th>Branch and Bound (100%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>94.8 (108%)</td>
<td>91.6 (104.5%)</td>
<td>90.6 (103.4%)</td>
<td>87.6 (100%)</td>
<td>87.6 (100%)</td>
</tr>
<tr>
<td>30</td>
<td>142.8 (111.9%)</td>
<td>137 (107.3%)</td>
<td>132 (103.4%)</td>
<td>127.6 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>40</td>
<td>174.6 (114.0%)</td>
<td>169 (110.3%)</td>
<td>163.2 (106.5%)</td>
<td>153.2 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>50</td>
<td>196.6 (110.5%)</td>
<td>194.4 (109.3%)</td>
<td>189.6 (106.6%)</td>
<td>177.8 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>60</td>
<td>228.6 (113.3%)</td>
<td>226 (112.0%)</td>
<td>221.8 (109.9%)</td>
<td>201.8 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>70</td>
<td>245.8 (113.6%)</td>
<td>246 (113.7%)</td>
<td>242.2 (111.9%)</td>
<td>216.4 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>80</td>
<td>270.6 (115.0%)</td>
<td>269.4 (114.4%)</td>
<td>267 (113.4%)</td>
<td>235.4 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>90</td>
<td>291.6 (113.0%)</td>
<td>289 (112.0%)</td>
<td>287.8 (111.6%)</td>
<td>258 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>100</td>
<td>329.8 (111.3%)</td>
<td>328.4 (110.9%)</td>
<td>326.2 (110.1%)</td>
<td>296.2 (100%)</td>
<td>N.A.</td>
</tr>
<tr>
<td>Average</td>
<td>219.5 (112.6%)</td>
<td>216.8 (111.2%)</td>
<td>213.4 (109.5%)</td>
<td>194.9 (100%)</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Table 3. DII and memory cost comparisons in pipelined systems.

<table>
<thead>
<tr>
<th>#tasks</th>
<th>DII cost</th>
<th>Memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>425 (101.9%)</td>
<td>6920 (100%)</td>
</tr>
<tr>
<td>30</td>
<td>831 (101.3%)</td>
<td>12,568 (100%)</td>
</tr>
<tr>
<td>40</td>
<td>1241 (102.8%)</td>
<td>15,599 (100%)</td>
</tr>
<tr>
<td>50</td>
<td>1616 (102.3%)</td>
<td>19,434 (100%)</td>
</tr>
<tr>
<td>Total</td>
<td>4113 (102.2%)</td>
<td>54,521 (100%)</td>
</tr>
</tbody>
</table>

Table 4. Trade off between DII and memory.

<table>
<thead>
<tr>
<th>#tasks</th>
<th>α = 0.7</th>
<th>α = 0.5</th>
<th>α = 0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DII</td>
<td>Memory</td>
<td>DII</td>
</tr>
<tr>
<td>20</td>
<td>417</td>
<td>6920</td>
<td>459</td>
</tr>
<tr>
<td>30</td>
<td>820</td>
<td>12,568</td>
<td>907</td>
</tr>
<tr>
<td>40</td>
<td>1207</td>
<td>15,599</td>
<td>1331</td>
</tr>
<tr>
<td>50</td>
<td>1579</td>
<td>19,434</td>
<td>1781</td>
</tr>
<tr>
<td>Total</td>
<td>4023 (100%)</td>
<td>54,521 (100%)</td>
<td>4478 (111.3%)</td>
</tr>
</tbody>
</table>
Table 5. Energy consumption and latency comparisons (β = 0.3).

<table>
<thead>
<tr>
<th>#tasks</th>
<th>EGMS Energy consumption</th>
<th>Ours Energy consumption</th>
<th>EGMS Latency</th>
<th>Ours Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>67.746 (100%)</td>
<td>64.974 (95.9%)</td>
<td>86.8 (100%)</td>
<td>91.2 (105.1%)</td>
</tr>
<tr>
<td>30</td>
<td>95.442 (100%)</td>
<td>90.238 (94.5%)</td>
<td>127.2 (100%)</td>
<td>131.8 (103.6%)</td>
</tr>
<tr>
<td>40</td>
<td>131.246 (100%)</td>
<td>127.344 (97.0%)</td>
<td>153.8 (100%)</td>
<td>161.2 (104.8%)</td>
</tr>
<tr>
<td>50</td>
<td>179.134 (100%)</td>
<td>163.128 (91.1%)</td>
<td>173.2 (100%)</td>
<td>179.4 (103.6%)</td>
</tr>
<tr>
<td>Average</td>
<td>118.392 (100%)</td>
<td>111.421 (94.1%)</td>
<td>135.25 (100%)</td>
<td>140.90 (104.2%)</td>
</tr>
</tbody>
</table>

Table 6. Energy consumption and latency comparisons (β = 0.7).

<table>
<thead>
<tr>
<th>#tasks</th>
<th>EGMS Energy consumption</th>
<th>Ours Energy consumption</th>
<th>EGMS Latency</th>
<th>Ours Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>67.746 (100%)</td>
<td>66.464 (98.1%)</td>
<td>86.8 (100%)</td>
<td>88.2 (101.6%)</td>
</tr>
<tr>
<td>30</td>
<td>95.442 (100%)</td>
<td>93.768 (98.2%)</td>
<td>127.2 (100%)</td>
<td>130.4 (102.5%)</td>
</tr>
<tr>
<td>40</td>
<td>131.246 (100%)</td>
<td>128.524 (97.9%)</td>
<td>153.8 (100%)</td>
<td>160.4 (104.3%)</td>
</tr>
<tr>
<td>50</td>
<td>179.134 (100%)</td>
<td>171.642 (95.8%)</td>
<td>173.2 (100%)</td>
<td>176.8 (102.1%)</td>
</tr>
<tr>
<td>Average</td>
<td>118.392 (100%)</td>
<td>115.100 (97.2%)</td>
<td>135.25 (100%)</td>
<td>138.95 (102.7%)</td>
</tr>
</tbody>
</table>

ments, we used 5 randomly generated task graphs for each task.

The results of our algorithm are obtained by taking up to 20 tasks in a cluster for simultaneous optimization.

7. Conclusions

We developed an effective algorithm to map and schedule tasks simultaneously for heterogeneous processors. By partitioning all tasks into several clusters, only a small number of tasks in a cluster are re-mapped and rescheduled at the same time. Therefore, the run-time can be controlled by adjusting the cluster size and can increase linearly with the number of tasks. Experimental results show that our algorithm can obtain 9.5%, 11.2% and 12.6% better solutions compared to PETS, HEFT and CPOP algorithms, respectively, in batch-mode systems. Furthermore, our method can improve the DII_cost by 2% and Memory_cost by 23% when compared to [7] in pipelined systems. Finally, energy-aware cluster slack optimization results show that our algorithm can effectively perform the trade-off between the latency and the energy consumption.

The techniques described in this paper can be applied to static scheduling for multiple processors, to optimize latency, throughput and energy. Future works include developing dynamic scheduling techniques, optimization for networks on chip and consideration of memory bandwidth.

8. Acknowledgement

This work was supported by the Ministry of Science, ICT & Future Planning and IDEC Platform center (IPC) in Korea.

REFERENCES


[6] P. Paulin and J. Knight, “Scheduling and Binding Algo-


