Scheduling Moldable Parallel Streaming Tasks on Heterogeneous Platforms with Frequency Scaling

Sebastian Litzinger 1, Jörg Keller 1, Christoph Kessler 2

1Faculty of Mathematics and Computer Science, FernUniversität in Hagen, Germany
2Dept. of Computer and Information Science (IDA), Linköping University, Sweden

Motivation

- Signal processing applications are often implemented by a set of streaming tasks
- Throughput requirement gives execution unit maximum time span for single execution of assigned tasks (one scheduling round)
- Low energy consumption and low average power consumption are desirable with regard to purchasing, operational, and maintenance costs
- High throughput is desirable but power and energy consumption are often constrained
- Tasks may have to be executed in parallel (if possible) to facilitate low makespan of round
- Core operating frequency influences energy consumption as well as runtime
- Architecture might be heterogeneous, complicating scheduling
- Tasks may differ in execution speed or power consumption, e.g. due to instruction mix
- Static scheduling pays off since application runs for years in a large number of devices
- For optimal schedule, solve (mixed) integer linear program (MILP/ILP)

Contributions

- We present a static scheduling algorithm for a set of tasks on a heterogeneous platform with frequency scaling, to meet a deadline and minimize energy consumption, given that the tasks are of different types and thus have different power and speed profiles on this platform.
- We extend the scheduling algorithm to situations where an energy budget per round or an average power budget is given, and the makespan for this round is minimized.
- We perform experiments with accurate profiles of ARM’s big.LITTLE architecture

Streaming Task Graph

Each task does a specific job, input tasks take input, follow-up tasks are provided with results from predecessors. All tasks are activated repeatedly, as the input data repeatedly arrives, i.e. forms a data stream.

Further reading


Crown Scheduling

In crown scheduling, a task is mapped to a particular processor group, which lowers scheduling complexity. Possible allocations thus are powers of 2. Moreover, each task is assigned an operating frequency during scheduling.

Optimization problems for scheduling n tasks to p cores with s discrete frequency levels (\(x_{i,j,k} = 1\) if task \(j\) is mapped to core \(i\) at frequency level \(k\)).

Variables:

- binary \(x_{i,j,k}\) \(i = 1..2p - 1, j = 1..n, k = 1..s\)

Real \(T_{max}\)

(1) Min. energy \(E\) for given deadline \(M\)
- \(\forall i: T_i \leq M\)

(2) Min. makespan \(T_{max}\) for energy budget \(E_{max}\)
- \(\forall i: T_i \leq T_{max}\)
- \(E \leq E_{max}\)

(3) Min. makesp. \(T_{max}\) for av. power budget \(P_{avg}\)
- \(\forall i: T_i \leq T_{max}\)
- \(E \leq P_{avg} \cdot T_{max}\)

Additional constraints for all targets

\(\forall j: \sum_{i,k} x_{i,j,k} = 1\)

\(\forall j: \sum_{i,k} \sum_{x_{i,j,k} = 1}\)

Figure 2. Top: A binary crown for \(p = 8, c = 2\) of 2 different types, where the core types are given by the color coding (orange = A15-cores (big), green = A7-cores (LITTLE)). The boldface numbers 1, . . . , 15 show the processor group indices.

Figure 3. (M)ILPs for different optimization targets ("scenarios"). \(T_i\) signifies the runtime of core \(i\).

Experiments

- 40 synthetic task sets of varying cardinality (10–80 tasks), 5 different task types
- Real frequencies and power consumption values for the ARM big.LITTLE architecture
- TAS: Task type-aware approach for sequential tasks (Keller & Holmbacka 2017)
- TIP: Task type-ignorant crown scheduler for parallelizable tasks (Melot et al. 2015)
- TAP: Task-type-aware crown scheduler for parallelizable tasks
- Implementation in Python with Gurobi solver, 5 minute wall clock timeout for each (M)ILP

Results

Table 1. Runtime, timeout occurrences and number of infeasible models for all scenarios and scheduling approaches

<table>
<thead>
<tr>
<th>Scenario scheduling</th>
<th>Runtime [min]</th>
<th>#Timeouts</th>
<th>#Infeasible</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAP</td>
<td>563</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>TAS</td>
<td>637</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>TIP</td>
<td>254</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TAP</td>
<td>764</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>TAS</td>
<td>797</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>TIP</td>
<td>1383</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>TAP</td>
<td>683</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>TAS</td>
<td>733</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>TIP</td>
<td>1653</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. Results for scenario 1, relative to TAP

<table>
<thead>
<tr>
<th>Scheduling task set cardinality</th>
<th>Makespan</th>
<th>Energy</th>
<th>Deadline violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAS</td>
<td>1.000</td>
<td>1.004</td>
<td>6.0</td>
</tr>
<tr>
<td>TIP</td>
<td>1.000</td>
<td>1.001</td>
<td>6.0</td>
</tr>
<tr>
<td>TAP</td>
<td>1.225</td>
<td>1.316</td>
<td>8.0</td>
</tr>
<tr>
<td>TIP</td>
<td>1.157</td>
<td>1.313</td>
<td>9.0</td>
</tr>
<tr>
<td>TAP</td>
<td>1.109</td>
<td>1.341</td>
<td>8.0</td>
</tr>
<tr>
<td>TIP</td>
<td>1.184</td>
<td>1.307</td>
<td>32</td>
</tr>
</tbody>
</table>

Scenario 1 (min \(E, M\) given):

- TAP vs. TAS: advantage TAP for small task sets (feasible schedule in any case), tasks executed sequentially anyways for larger task sets
- TAP vs. TIP: lower makespan (more pronounced for small task sets), lower energy consumption (more pronounced for larger task sets), TIP: deadline violation in 80% of all cases

Scenario 2 (min makespan, \(E\) given):

- TAP vs. TAS: same behavior as for scenario 1, relative performance of TAP better
- TAP vs. TIP: TAP’s relative performance even better than for scenario 1

Scenario 3 (min makespan, \(P_{avg}\) given):

- TAP vs. TAS: TAP still better for small task sets, feasible solution can always be found (due to nature of constraints)
- TAP vs. TIP: lower makespan due to TIP overestimating energy consumption and thus not exploiting power budget