Scheduling of Stream-Based Real-Time Applications for Heterogeneous Systems

Bruno Virlet, Xing Zhou, Jean-Pierre Giacalone†, Bob Kuhn†,
María Jesús Garzarán, and David Padua

University of Illinois at Urbana-Champaign
{virlet1,zhou53,garzaran,padua}@illinois.edu

†Intel Corporation
{jean-pierre.giacalone,bob.kuhn}@intel.com

Abstract

Designers of mobile devices face the challenge of providing the user with more processing power while increasing battery life. Heterogeneous systems offer some opportunities to solve this challenge. In an heterogeneous system, multiple classes of processors with dynamic voltage and frequency scaling functionality are embedded in the mobile device. With such a system it is possible to maximize performance while minimizing power consumption if tasks are mapped to the class of processors where they execute the most efficiently.

In this paper, we study the scheduling of tasks in a real-time context on a heterogeneous system-on-chip that has dynamic voltage and frequency scaling functionality. We develop a heuristic scheduling algorithm which minimizes the energy while still meeting the deadline. We introduce the concept of cross-platform task heterogeneity and model sets of tasks to conduct extensive experiments. The experimental results show that our heuristic has a much higher success rate than existing state of the art heuristics and derives a solution whose energy requirements are close to those of the optimal solution.

Categories and Subject Descriptors I.2.8 [*Problem Solving, Control Methods, and Search*]: Scheduling; C.1.3 [*Other Architecture Styles*]: Heterogeneous (hybrid) systems

General Terms Algorithms, Performance

Keywords Dynamic voltage and frequency scaling, heterogeneous system, scheduling

1. Introduction

Advances in portable devices demand higher speed of computation as well as longer operational autonomy. To address the challenge of these opposite goals we study techniques that balance program execution speed and energy consumption in the context of typical portable device applications. We focus on streaming computations, which are networks of tasks that operate on a data stream that flows into the computation at a fixed rate [10, 20]. Upon completion, each task passes its output to its successor(s) in the network. Streaming computations are often used to implement video post-processing applications, which are among the most important for the future of mobile devices. In video post-processing computations, tasks implement filters and the data stream is a sequence of video frames. The computation is typically subjected to a real-time constraint which is to display between 15 and 30 frames per second for many of today's video post-processing applications.

The objective of the techniques discussed in this paper is to minimize the energy consumed by streaming computations under the constraint of a minimum output rate. We assume that there could be multiple classes of processors embedded in the mobile device and that they have dynamic voltage and frequency scaling (DVFS) functionality. Since the maximum possible frequency is not typically required to achieve the desired output rate, energy consumption can often be minimized by lowering the voltage and hence the frequency as much as the real time constraint allows. This minimization process is complicated by three situations. First, only a discrete number of frequencies are possible in today's machines. Second, changing the voltage/frequency consumes time, which means that such changes must be applied judiciously. Third, energy efficiency can be controlled not only by DVFS but also by choosing the class of processor on which to map each task since different processors are efficient for different types of tasks. A GPU will excel in vector operations but will be inferior to a conventional CPU for applications rich in control flow. An example of Systems-on-a-Chip (SOC) that integrate a multiprocessor core with graphics cores is the AMD Fusion Zacate E350/E240 [1]. Another example is the PowerVR SGX 5XT from Imagination [2] for smartphones and other mobile devices, which can be integrated with a multiprocessor core to obtain a heterogeneous SOC. Although these systems are not yet available for mobile devices, we expect them to reach the market soon.

The technique that we propose in this paper takes the form of a two step heuristic that first chooses on what class of processors to map each task and then uses a homogeneous scheduling algorithm to apply the voltage and frequency scaling within each homogeneous subsystem. We allow frequency scaling at the granularity of the tasks. This enables us to place the code for frequency scaling at natural locations. Our heuristic outperforms other heuristics such as Greedy and LR, both presented in [22]. It has a high success rate at finding a feasible schedule and, for most of the cases we studied, the resulting schedule is within 5% of the one of the optimal schedule. It also offers very stable results when the architecture heterogeneity and task uniformity vary. Additionally, the memory usage is linear in the size of the input. Finally, we demonstrate the importance of the concept of the cross-platform heterogeneity of a

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.

task in the choice of the processors and show how combining two heuristics enables even better results.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the problem we are trying to solve. Section 4 discusses our scheduling algorithm. Section 5 gives some details about other techniques that we use to compare against our proposed scheduling algorithm. Section 6 describes the environmental setup that we use to run the experiments in Section 7. Finally, Section 8 discuss future work and our final conclusions.

2. Related Work

Task scheduling for power efficiency is a fairly recent problem. There have been studies on scheduling for one processor with DVS capabilities [19, 23, 12, 14] and for homogeneous multiprocessors also with DVFS capability [16, 17, 5]. In Section 4.1.3, we discuss one of the heuristics for homogeneous multiprocessors, the SpringS algorithm [17]. We use this algorithm as part of our scheduling strategy for heterogeneous systems.

Mixed software-hardware strategies in which the application is partitioned between hardware and software components [11] have also been studied but they require some of the scheduling to happen at hardware design time. Mixed strategies may offer higher performance because they make use of specialized hardware.

The problem of scheduling on heterogeneous processors is discussed only in a few papers. The case of heterogeneous single-voltage setup system has been studied in [9, 13]. The single-voltage setup problem consists in choosing a fixed frequency for each processor. This technique helps system designers choose the most efficient operating point for the products. However, such systems clearly lack flexibility when optimizing different applications.

Luo and Jha addressed the heterogeneous scheduling problem with continuous voltage scaling [18]. They assume that the frequency can be scaled continuously between a minimum and a maximum value. Yang, Chen, Kuo and Thiele [21] study the heterogeneous multi-level voltage scheduling problem and propose an algorithm which accepts a factor $\rho > 1$ and generates schedules whose energy consumption are less than ρ times the optimal possible energy required.

To the best of our knowledge the paper by Yu and Prasanna [22] is the only one so far to propose a reasonably efficient algorithm for the problem of scheduling in a heterogeneous system with multi-level discrete frequencies. They propose the LR heuristic that we discuss in Section 5.3 and evaluate in our experimental results in Section 7. The LR heuristic is a fast heuristic algorithm based on the linear relaxation of the linear-programming problem. The heuristic we introduce in this paper outperforms the LR heuristic by succeeding in multiple cases where LR fails, as shown in the experimental results in Section 7.

3. Problem formulation

3.1 Hardware and Power Model

We assume that the target system contains m processing elements (PE) P_j for $j = 1 \dots m$. We also assume a finite number of possible frequencies. The system considered is heterogeneous, which means that it contains different classes of processors (or processor species): we assume that there are q different processor types $S_1 \dots S_q$ in the system. We say that $P_j \in S_k$ if P_j is of type S_k.

For each processor type S_k there is a set F_k of allowed frequencies. If the processor type does not support DVS, the set F_k is reduced to the singleton containing the only frequency offered by processors of type S_k.

The power consumed by the system will fluctuate over time depending on which processor units are in use. The power function indicating the average energy consumption rate of the processor as a function of the frequency can be obtained from specifications or can be measured. For each frequency $f \in F_k$, let $p_k(f)$ be the associated average dynamic power of a processor of type S_k when running at frequency f. We will not consider the static power because we assume that it is constant. We assume that $\forall k \in [1, q]$, $f \longmapsto p_k(f)$ is an increasing function of the frequency. Generally, $p_k(f)$ is also convex such that a slight increase in the frequency at low frequency will not have much impact on power whereas the same increase at high frequency produces a much higher power increase. In fact, for CMOS DVS processors, the dynamic power function $p_k(f)$ can be approximated by $p_k(f) = Cf^3/\kappa^2$ where C is the switch capacitance and κ is a design specific constant [8].

3.2 Application Model

The scheduling problem of dependent tasks without dependence cycles can be reduced to the scheduling of a kernel of independent tasks as it has been shown by Liu et al. [17]. We will discuss this point more in depth in section 4.1.1.

Therefore, we will consider a set of n independent tasks T_i ($i = 1, 2, \dots, n$) and a deadline d defined as the maximum time allotted to process one dataset (in the case of video post-processing, this would be the time required to generate one video frame). We define C_{ik}^S as the number of cycles required to execute task T_i on a processor of type S_k. C_{ij} is also defined as the number of cycles to execute T_i on processor P_j. We assume that the number of cycles is not data dependent. This is the case for the video post-processing filters we studied. If it was data dependent, C_{ik}^S could be defined as the worst-case number of cycles so that the algorithm can guarantee the feasibility of the found schedule.

Let $\hat{C}_{ik}^S = \frac{1}{q}\sum_{k=1}^{q} C_{ik}^S$ be the average cycles of a task on the different types of processing units. We define the cross-platform *heterogeneity* of a task by:

$$H_i = \frac{\sum_{k=1}^{q}(C_{ik}^S - \hat{C}_{ik}^S)^2}{\hat{C}_{ik}^S}$$

A task T_{i_1} is *more heterogeneous* than another task T_{i_2} if $H_{i_1} > H_{i_2}$. This means that choosing the right processor for T_{i_1} has a more significant impact on its execution time than choosing the right processor for T_{i_2} would have on T_{i_2}'s execution time.

Notice that the execution time of task T_i on processor P_j and at frequency f is $\frac{C_{ij}}{f}$.

3.3 Scheduling Problem

Let $V_z = (P_{j_z}, f_z)$ be a processor-frequency pair. Our goal is to find a mapping of each task T_i onto a pair V_z that minimizes energy consumption and produces results at the rate of $1/d$ results per second, where d is determined by the real-time constraint of the application. For the algorithm presented in this paper we ignore the communication costs. Thus, our model does not account for the time consumed to bring the data to the processor the first time that they are accessed or the time required to move data from the cache or memory in a processing element to the cache or memory of another processing element, independently of whether these two processing elements are of the same or of different type. To measure to what extent this simplification could affect our results we ran some experiments with four filters of a video post-processing application from MJPEGTools [3] and we observed that the processor was very effective at hiding the latency the first time the frame was brought to a core, most likely because the hardware prefetcher managed to hide the latency of the memory accesses due to the regular access pattern of the filters in this application. We do not have data for the transfer costs of the data between processing units of different types, but we expect that in a SOC the communication cost between processing elements of different types will be signif-

icantly smaller than today's cost between the processor cores and the off-chip GPU. In addition, prefetching (hardware or software) should be able to help at decreasing communication costs. Next, we describe more formally the scheduling problem that the algorithm presented in this paper is trying to solve.

Given S_k, the set of processing elements of type k and F_k, the set of frequencies allowed in each processor type, there are $v = \sum_{k=1}^{q} |S_k| \times |F_k|$ pairs. Let x_{iz} be 1 if task T_i is mapped to the pair V_z (which means that the task T_i will run on processor P_{j_z} at frequency f_z) and 0 otherwise. The value $e_{iz} = p_k(f_z)\frac{C_{ij_z}}{f_z}$ is the energy consumed by task T_i running at frequency f_z on processor P_{j_z} of type S_k. Given $V_z = (P_{j_z}, f_z)$, the fraction of processor P_{j_z} utilized by the task T_i when mapped to a pair V_z is $u_{iz} = \frac{C_{ij_z}}{d \times f_z}$ where d is the deadline. If this fraction is smaller than 1, there is still room in the processor to accommodate other tasks and still meet the deadline. The utilization U_j of processor P_j is the sum of the utilizations for all tasks: $U_j = \sum_{i=1}^{n} \sum_{V_z} u_{iz} x_{iz}$, where $V_z \in \{(P_{j_z}, f_z) | j_z = j\}$. For each processor P_j, the real-time constraint requires that $U_j \leq 1$.

The scheduling problem can be formulated as an integer linear programming problem ILP0 which consists in the minimization of:

$$\sum_{i=1}^{n} \sum_{z=1}^{v} e_{iz} x_{iz} \tag{1}$$

such that:

$$U_j \leq 1 \qquad 1 \leq j \leq m \tag{2}$$

$$\sum_{z=1}^{v} x_{iz} = 1 \qquad 1 \leq i \leq n \tag{3}$$

$$x_{iz} \in \{0, 1\} \qquad 1 \leq i \leq n, 1 \leq z \leq v \tag{4}$$

While Equation 2 states that the deadline must be respected, Equations 3 and 4 specify that each task has to be mapped entirely to one processor. We call *optimal solution* any solution to this problem, if it exists. We call *feasible schedule* any solution to this problem without the minimization constraint (Equation 1). Finally, a mapping of all the tasks which does not respect the time constraint (Equation 2) but satisfies Equations 3 and 4 is called an *unfeasible schedule*.

For discrete frequencies, finding the optimal schedule that minimizes the energy consumption while meeting the time constraint is clearly a NP-hard problem since the scheduling problem is NP-hard even ignoring the energy issues. We therefore must solve the problem with a heuristic to avoid the exponential complexity. The algorithm we use to address this problem is presented in the next section.

4. Scheduling Algorithm

4.1 Algorithm

Before scheduling an application, it is necessary to identify which tasks to schedule. In a first section we study how to build a kernel of independent tasks. Our heuristic has then two phases which apply to this kernel. The first, *mapping*, chooses the processor type for each task. The power function is used in that phase to guide the choice. The second phase, *frequency choice*, chooses one of the processors within the type selected in the first phase and the frequency for each task.

4.1.1 Task Set Identification

A stream-based application consists of a set of filters where each filter is applied in sequence to each input item. Thus, the stream application contains a total of n tasks, where $n = number\_of\_filters \times number\_of\_input\_items$. In order to take advantage of the several processing units, we need to partition the total number of tasks into sets of independents tasks or kernels that can execute in parallel. To extract these kernels we follow an approach similar to software-pipelining [17]. This strategy can be easily applied when there are not data dependence cycles between the tasks, i.e., the task graph can be represented as a directed acyclic graph or DAG.

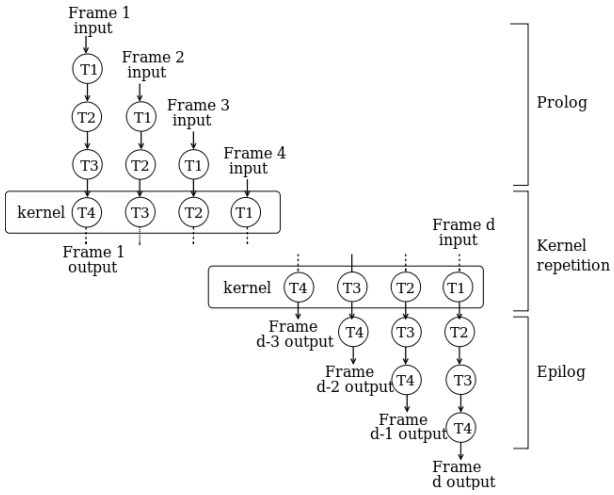


Figure 1: Kernel identification for four video filter tasks and d frames.

If we call $T_i[k]$ to filter T_i when applied to item k, the normal chain of dependencies would be $T_1[k] \rightarrow T_2[k], \cdots \rightarrow T_n[k]$. To obtain independent tasks one can select filters that apply to different items. This way $T_1[k], T_2[k+1], T_3[k+2], \ldots T_{m-1}[k+m-2], T_m[k+m-1]$ form a kernel whose tasks are independent on each other, as they operate on different items. Figure 1 illustrates an example when $m = 4$, where the tasks in the kernel repeat several times, and a prolog step at the beginning and an epilog at the end fill and drain the pipeline, respectively.

Notice that in the case of the video post-processing application we considered four different filters from the MJPEGTools [3]: denoise, sharpen, increase frame rate, and up-scale, where the input items to these filters were the frames of the video being processed.

Once a kernel of independent tasks has been built, it is usually possible to create sub-tasks. For instance, in the case of video post-processing applications, the filters can be easily tiled. Then, instead of considering task T_i working on frame k, we can consider two subtasks, $T_{i,1}$ and $T_{i,2}$ working respectively on two subsets of frame k. The result is a more flexible scheduling, as the increase of the number of tasks increases the number of tasks for the scheduler to choose from. In fact, our experimental results in Section 7 show that increasing the number of tasks has an impact on the optimality of the schedule.

Finally, consider the case where the dependence graph of the tasks is not a DAG, that is, there are back arcs due to dependences across data stream entities. In this case, applying software pipelining may produce a kernel with multiple steps instead of the single step that arises when the dependence graph is a DAG. The methodology described below can be applied to each one of these steps.

4.1.2 Mapping Phase

This Section describes the mapping of each task in the kernel to a processor type. The mapping algorithm is shown in Algorithm 1. The power function is used in this phase to guide the search. The power function is an increasing function, that is, the faster a pro-

cessor computes, the more energy it consumes. Thus, to minimize energy consumption this algorithm uses a heuristic that maps tasks to processor types at the lowest possible frequency that still meets the deadline.

Algorithm 1 Heuristic algorithm

1: *Input:* kernel of independent tasks T_i, set of processors P_j.
2: *Output:* variables x_{iz} set to 1 if task i is mapped to the pair processor/frequency V_z.
3: **for all** i, j **do**
4: $x_{ij} = 0$
5: **end for**
6: **for** each task T_i in decreasing H_i order **do**
7: **for** each processor P_j of type S_k **do**
8: $f_{opt,j} = \frac{\sum_{i'} x_{i'j} C_{i'j} + C_{ij}}{d}$.
9: $f_{new,j} = f_{opt,j}$ round to the next discrete frequency available on S_k or $max(F_k)$ if $f_{opt,j} \geq max(F_k)$.
10: $\delta e_j = p_k(f_{new,j})[(\sum_{i'} \frac{x_{i'j} C_{i'j}}{f_{new,j}}) + \frac{C_{ij}}{f_{new,j}}] - p_k(f_j) \sum_{i'} \frac{x_{i'j} C_{i'j}}{f_j}$
11: **end for**
12: Choose the pair V_z such that δe_z is minimal and task T_i fits on P_{j_z} if T_i and all the tasks already assigned to P_{j_z} were to run at maximum frequency. Fail if no such processor exists.
13: $x_{iz} = 1$.
14: **end for**

The algorithm assigns processors to tasks following a decreasing cross-platform heterogeneity order (line 6), where the cross-platform heterogeneity of a task is computed as shown in Section 3.2. By following this order, we are giving more choices to those tasks whose energy consumption is more affected by the type of processor where they run. The end goal is to minimize the overall energy consumption. Our experimental results in Section 7.2 will show the effectiveness of this heuristic.

Then, the algorithm computes the "insertion frequency" of this task on each processing element and the corresponding energy increase if the task were to be mapped on this processing element (lines 7 to 11).

The "insertion frequency" $f_{opt,j}$ is the ideal frequency [15, 6] at which the processor P_j should run so that all the tasks already assigned to P_j finish within the time constraint while minimizing the energy consumed. This frequency converges to an approximation of the frequency at which all the tasks on the processor should run in an ideal situation. In general this frequency is not available and the tasks would have to run at the smallest higher discrete frequency available $f_{new,j}$. If the insertion frequency is higher than the highest available frequency, this processor will not be able to execute this task and all the previously assigned tasks within the deadline d. In this case, if space is not available on another processor, the scheduling algorithm fails.

At line 12 we choose the processor which results in the minimum energy increase when the task is assigned to that processor at the insertion frequency. We also make sure that the deadline is always met when running at the maximum frequency. By checking this, the algorithm makes sure that it is not generating an infeasible schedule. It might be possible that, at this step, no processor can run an additional task within the defined deadline d. If this is the case, the heuristic fails. This, however, does not mean that no feasible schedule exists. Failure is inherent to the heuristic approach. In section 4.2, we discuss ways to reduce the failure rate in finding a feasible solution.

Finally, we record the pair processor/frequency V_z where the task under consideration will run (line 13) and proceed to the next task.

4.1.3 Frequency Choice Phase

Once this first phase finishes, each task is associated with a given processor type S_k. In the process, we actually assigned each task to a specific processor. In this last phase, we rearrange the tasks mapped to processors of the same type and assign them a final processor and frequency. We consider each group of processors of the same type in Algorithm 2 and apply to them an homogeneous scheduling technique as shown in Algorithm 3. We chose to use the SpringS algorithm by Liu et al. [17]. SpringS reorganizes the tasks mapped to the processors of the same type and search for the appropriate frequency. Applying the SpringS algorithm is only possible because the processors of the same type have the same characteristics (frequency and cycles for each task). This algorithm starts with an existing schedule. In our application, we start with the schedule found by the mapping phase and we reset all the frequencies to the minimum frequency (Algorithm 3, line 3). Then the SpringS algorithm, as its name indicates, behaves like a Spring. If a processor does not meet the deadline, that is, its utilization is larger than 1, it will find the best task for which to increase the frequency (the one that results in the smallest energy increase) and try to reschedule a subset of the tasks (lines 7 to 13). On the other hand, if the schedule has some slack to meet the deadline, it tries to slow down a task to save some energy (lines 15 to 17). After this phase, the variables x_{iz} are final and define a feasible schedule for ILP0.

Notice that in the case of homogeneous scheduling, the mapping phase can be skipped and our heuristic is reduced to the SpringS heuristic.

Algorithm 2 Frequency Choice Phase

for each set $\mathcal{S}_k$ **do**
Apply the SpringS algorithm to $\mathcal{S}_k$ with the schedule found by the mapping phase.
end for

4.2 Improving the Heuristic

Heuristics can fail to find a feasible solution, as shown by our experimental results in Section 7. Thus, we have search different solutions to improve the success rate of the heuristic. Heuristics run fast. Hence, the first solution is to run a different heuristic in combination with our heuristic. We call this combination the *hybrid heuristic*. This heuristic selects the best results between LR-heuristic [22], which we will present later, and our heuristic. As we will see in the next section, not only does the hybrid heuristic help improve the optimality but it also reduces the failure rate of the scheduler. In addition, since both heuristic run fast, this can also be used as a strategy to improve a given schedule.

Additionally, it is clear that solving the scheduling problem for a deadline tighter than the required constraint also satisfies the original problem. For instance, if the original problem requires a constraint d, any solution to the same problem with a new constraint βd with $0 \leq \beta < 1$ is also a solution to the original problem. Although the failure rate is higher for tighter constraints, the heuristic algorithms are sensitive to small changes in the constraint since the deadline is involved in the computation of the optimal insertion frequency (line 8 of Algorithm 1). Therefore, if our heuristic fails for the initial constraint, we can retry with a slightly tighter one and may find a valid schedule.

5. Other Approaches

We compare our results with a greedy heuristic and a heuristic based on the linear relaxation of the integer linear programming

Algorithm 3 SpringS Algorithm. The operators argmin and argmax refer respectively to the index of the minimum and of the maximum in an ordered set.

Input: initial schedule of tasks in $\mathcal{T}$ onto the processing elements in S_k.
Output: optimized schedule of tasks in $\mathcal{T}$ onto the processing elements in $\mathcal{S}_k$.
$\forall T_i \in \mathcal{T}$ such that $x_{iz} = 1$: $x_{iz} = 0$ and $x_{im} = 1$, where $\{V_m = (P_{j_m}, f_m) | (P_{j_m} = P_{j_z})$ and $(f_m = min(F_k))\}$
while true **do**
  $j_0 = \text{argmax}(\{U_j | P_j \in \mathcal{S}_k\})$
  if $U_{j_0} > 1$ **then**
    Find the task T_{ref} on P_{j_0} with the minimum energy increase when increasing its assigned frequency to the next step f_{ref+1} if its frequency is not already the highest.
    Let $\mathcal{R}$ be the set of tasks whose current execution time is smaller than the execution time of T_{ref} running at f_{ref+1}.
    for each task T_i in $\mathcal{R}$ in decreasing number of cycles order **do**
      $j_1 = \text{argmin}(\{U_j | P_j \in \mathcal{S}_k\})$
      Assign T_i to P_{j_1} without changing its frequency if all the tasks assigned to P_{j_1} and not in $\mathcal{R}$ fit at maximum frequency. If not, return the schedule previously found.
      Remove T_i from $\mathcal{R}$.
    end for
  else
    $j_1 = \text{argmin}(\{U_j | P_j \in \mathcal{S}_k\})$
    On P_{j_1}, find the task T with the minimum execution time increase when decreasing its frequency to the lower step.
    Decrease T's frequency if possible. If not, return the current schedule.
  end if
end while

problem (LR-heuristic) both presented in [22]. To be able to compare the algorithms in an absolute fashion, we also search for the optimal solution. We present these different approaches in the next subsections.

5.1 Solving Integer Linear Programming

The problem ILP0 is an NP-hard problem. For problems of small size, it can be solved by exhaustive search. We search for a solution of this previously defined integer linear programming problem by using *lp_solve*, an open-source linear solver [7]. By using a branch-and-bound approach, *lp_solve* gives us the optimal solution if it exists. For problems small enough, *lp_solve* will find the solution in a reasonable amount of time, which allows us to compare the optimality of the algorithms. Searching for the optimal solution to ILP0 may be slower than expected for small problem sizes depending on the characteristics of the input data.

5.2 The Greedy Heuristic

The Greedy heuristic will pick a task after another – order doesn't matter – and for each task consider all the possible processors and frequencies, and choose the task mapping that minimizes the energy consumption among all the possible combinations that are within the deadline. The Greedy heuristic is presented in Algorithm 4.

Clearly a problem with the Greedy algorithm is that it tends to allocate the first tasks considered at a low frequency and then the remaining tasks might not fit anymore. Therefore, the success rate of Greedy is expected to be low.

Algorithm 4 The Greedy heuristic

T = set of all tasks
while T is not empty **do**
  For each task T_i, find the pair V_z such that e_{iz} is minimum and that $U_{j_z} \leq 1$. Save the value as (T_i, V_z, e_{iz}).
  Among all the triplets, choose the one with the minimum energy e_{iz} and map the corresponding T_i it to the processor P_z and frequency f_z: $x_{iz} = 1$.
  Remove this task T_i from T.
end while

5.3 The LR-heuristic

The LR-heuristic [22] uses properties of the linear relaxation of the scheduling problem to iteratively map tasks to processors while reducing the size of the problem at each step by removing the tasks that are already mapped. The LR-heuristic considers the integer linear programming problem ILP0 defined previously. However, in that problem, the variables x_{iz} are constrained to be binary. The LR-heuristic solves the more general relaxed problem LP0 in which Equation 4 is changed into

$$x_{iz} \in [0, 1] \qquad 1 \leq i \leq n, 1 \leq z \leq v \tag{5}$$

The only difference with ILP0 is that the variables x_{iz} are allowed to take any value between 0 and 1. The LR-heuristic is then as described in Algorithm 5.

Algorithm 5 The LR-heuristic (LinRel)

repeat
  Remove all the useless x_{iz} variables which set to 1 would make $U_{j_z} > 1$.
  Solve the linear relaxation problem LP0. As proved in [22], at least one variable x_{iz} will be equal to 1, in spite of being able to take any value between 0 and 1.
  All the variables $x_{iz} = 1$ are fixed and removed from the problem.
until all tasks are mapped or no feasible schedule is found

6. Environmental Setup

In this Section we describe the environmental setup that we use to run our experiments.

6.1 Task Set Generation

To demonstrate the quality of our heuristic compared to the previously described heuristics, we ran a large number of experiments using synthetic task sets. In order to generate a synthetic task set, we define two parameters, the single-platform task uniformity τ [1] and the architecture heterogeneity η as presented in [4]. τ represents how different the cycles number of the tasks will be on the same platform. η allows to tweak how different this number will be between the various platforms. For a given task i, we draw τ_i from a uniform distribution $U(1, \tau)$ and for each processor type we draw $\eta_{i,k}$ from a uniform distribution $U(1, \eta)$ and we set C_{ik}^S to $\tau_i \eta_{i,k}$ cycles.

In order to have realistic parameters for these task sets, we use numbers from experiments on Intel(r) Atom(tm). We ran the five filters denoise, sharpen, color correction, increase frame rate and up-scale from MJPEGTools, and measured an average cycle

[1] Single-platform task uniformity is called task heterogeneity in [4]. We call it here single-platform task uniformity to avoid confusion with the cross-platform heterogeneity defined in Section 3.2

number of 10^{10}. Therefore, we chose $\tau = 10^5$ and $\eta = 10^5$ for our experiments to achieve an average cycle number of 10^{10}. In Section 7.3, we will consider different values of τ and η.

For the experiments we generate task sets of different sizes. The limit to the task set size is set by the execution time of the linear solver. We run experiments with up-to 40 tasks.

6.2 Time Constraint

Once we have a synthetic task set, we want to generate a reasonable deadline. The minimum execution time of a task i (t_i^{min}) is the execution time of this task on the processor best suited for this task at the maximum frequency available on this processor:

$$t_i^{min} = min(\{\frac{C_{ik}^S}{max(F_k)} | k \in [1..q]\})$$

We define the tight time constraint as the sum of the minimum execution time of the n tasks distributed among the m available processors:

$$d_{tight} = \frac{\sum_{i=1}^{n} t_i^{min}}{m}$$

Unless the tasks were perfectly balanced, the tight time constraint would be impossible to meet. This is why we define the relaxed time constraint:

$$d = \alpha \cdot d_{tight}$$

where α is an input parameter of the experiment. By varying α we obtain a constraint more or less tight. In our experiments with MJPEGTools, a deadline of 30 frames per second translated to a value $\alpha = 1.5$. In section 7, we present results for different values of α: 1.1, 1.5 and 2.0.

6.3 Hardware Configurations

We consider two different hardware configurations, configuration 1 and 2. Configuration 1 is composed of three processing units. Two of the processing units follow the power function and the available frequencies of an Atom CPU as described in Table 1a. The other processing unit has the power characteristics of a GPU with only one power state of 344 mW at 800 MHz as shown in Table 1b. Configuration 2 is another platform composed of four Atom-like and two GPU-like processing units with the power characteristics presented in Tables 1a and 1b, respectively.

6.4 Algorithms evaluated

We run experiments with five different algorithms. Greedy is the Greedy algorithm described in Section 5.2. LinRel is the LR heuristic described in Section 5.3. Heuristic is the algorithm proposed in the paper and described in Section 4.1. Heuristic with retry and Hybrid are the algorithms described in Section 4.2. The Heuristic with retry tightens the constraint βd up to 20% of the original constraint ($\beta = 0.80$) with a new try every 1%. This of course requires 20 runs of the heuristics and, consequently, the Heuristic with retry can run up to 20 times slower. We based the heuristic with retry on our heuristic. It would be possible to base it on the LR-heuristic which might improve it, or even on the hybrid heuristic which would of course lead to even better results.

7. Experiment Results

In this section, we present our experimental results. Section 7.1 evaluates the optimality of the different algorithms for different numbers of tasks and different constraints. Section 7.2 discusses different ways of sorting the tasks and evaluates the impact on the optimality of the schedule. Section 7.3 analyzes the sensitivity of the different heuristics to different values of the task uniformity and architecture heterogeneity τ and η. Section 7.4 compares the execution time of the different algorithms. Finally, Section 7.5 summarizes our results.

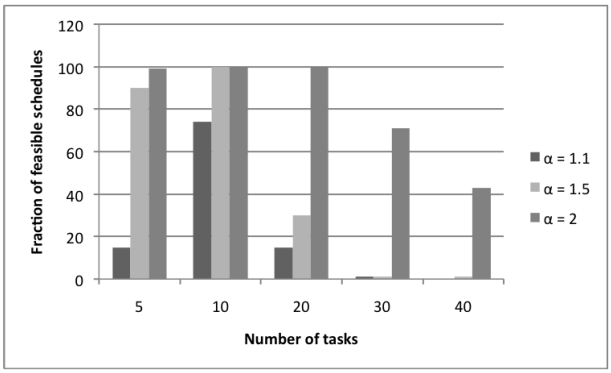


Figure 2: Fraction of feasible schedules for the linear solver with the hardware configuration 1. This shows what percentage of the generated task sets leads to feasible schedules. If the linear solver fails, it means that there is no feasible schedule for this task set.

7.1 Energy Savings and Success Rate

For each experiment, we generate one thousand synthetic task sets. Not every task set allows a feasible schedule but the linear solver (Section 5.1) will always find the optimal schedule if there is one. Running the linear solver first tells us if there is a feasible schedule and, if there is one, gives us the optimal energy. If there is no feasible schedule, we discard the task set. Figure 2 reports the percentage of feasible schedules over the one thousand task sets generated for different values of α and different number of tasks.

For each heuristic we measure its *success rate* as the ratio of number of schedules found to the number of feasible schedules. In addition, if the heuristic succeeds, it returns a schedule and its associated energy. We measure the optimality of the different heuristics as the ratio of the energy found to the optimal energy. We call it *error to optimal*.

Configuration 1: Two CPUs and one GPU The first set of experiments considers $\alpha = 1.1$ with configuration 1, two CPUs and one GPU. Such a value for α leads to a very tight constraint. The number of feasible schedules is very low and is close to 0% for more than 20 tasks, as seen on Figure 2. Therefore, we only present results up to 20 tasks for these experiments. The success rate of all the heuristics is shown in Figure 4a. The ratio of extra energy required when the heuristic does not find the optimal schedule is shown as the error to optimal plot in Figure 3a. As Figure 4a shows Heuristic outperforms the Greedy and the LR-heuristic, especially for 10 and 20 tasks. The Greedy heuristic finds only a few schedules. Although the success rates are low, the heuristics perform well when they find a schedule, with an average error of less than 5%, as shown in Figure 3a.

For $\alpha = 1.5$, success rate and error to optimal are shown in Figures 4b and 3b, respectively. As it can be seen the success rate of the heuristics improves significantly. We only show results for 20 tasks because the number of feasible schedules is still close to 0% for more than 20 tasks (See Figure 2). The Greedy heuristic is still lagging behind with less than 30% success rate for 5 tasks and almost 0% for 10 or more tasks. The LR-heuristic has a decent success rate that is above 70%, but all our heuristics outperforms it with, with close to 100% success rate.

Figure 3c presents the results for $\alpha = 2$. We can see that the success rate of Heuristic is close to the optimal (in average less than 3%) and outperforms the LinRel and the Greedy heuristic. For

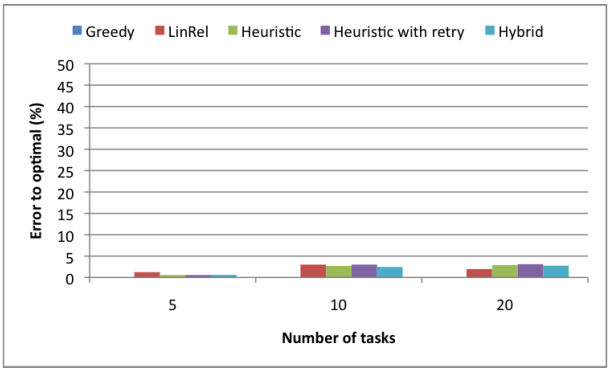
Frequencies (GHz)	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.4
Power (mW)	240	300	360	750	1100	1620	2160	3240

(a) ATOM CPU power function

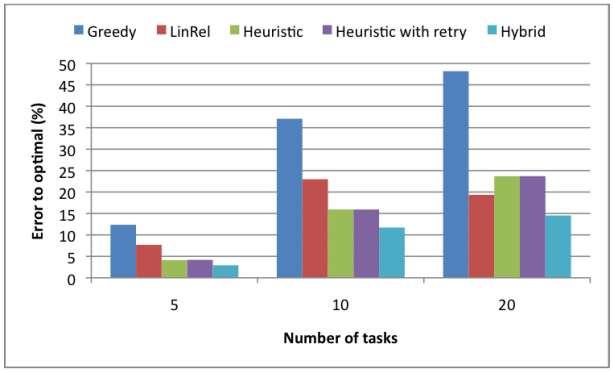
Frequencies (GHz)	0.8
Power (mW)	344

(b) GPU power function

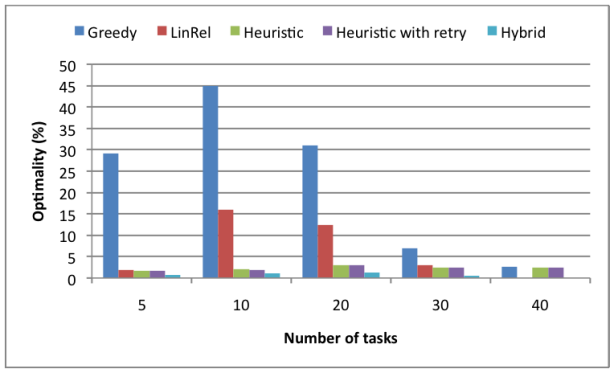
Table 1: Power functions of the processing units



(a) $\alpha = 1.1$

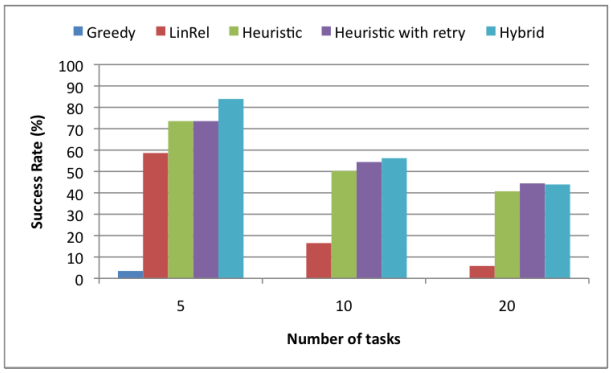


(b) $\alpha = 1.5$

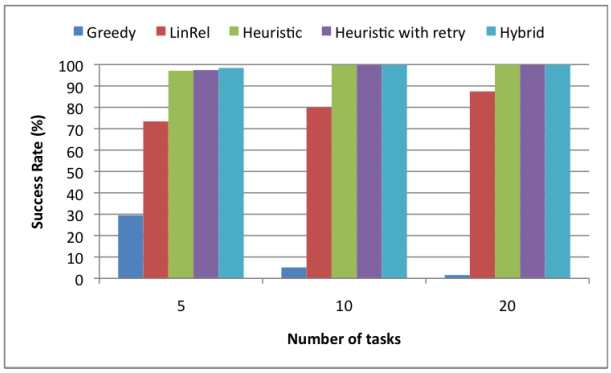


(c) $\alpha = 2$

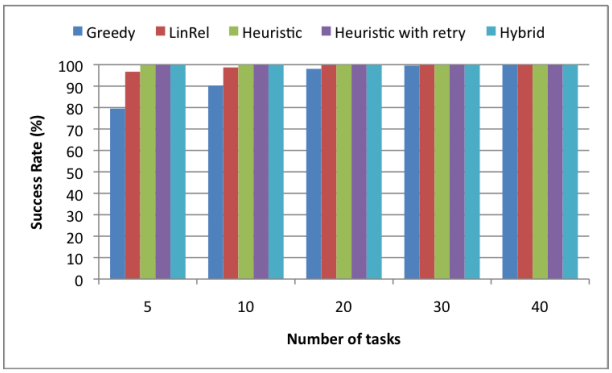
Figure 3: Error of the heuristics for different values of α. Two CPUs and one GPU. The horizontal axis is the number of tasks. The vertical axis represents how far from the optimal the heuristics are. The optimal is computed with the linear solver.



(a) $\alpha = 1.1$



(b) $\alpha = 1.5$



(c) $\alpha = 2$

Figure 4: Success rate of the heuristics. Two CPUs and one GPU. A success rate of 100% means that the heuristic found a schedule each time the linear solver found one.

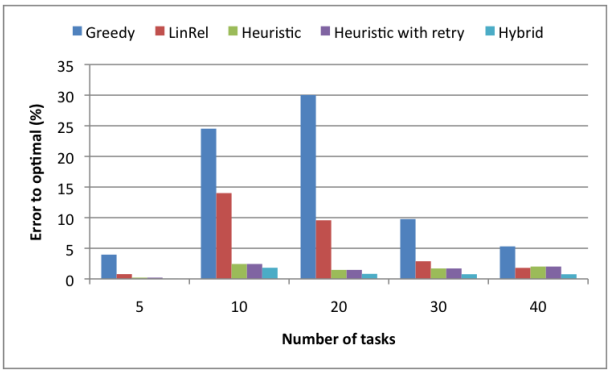


Figure 5: Error of the heuristics. Four CPUs and two GPUs with $\alpha = 2$. The horizontal axis is the number of tasks. The vertical axis represents how far from the optimal the heuristics are. The optimal is computed with the linear solver.

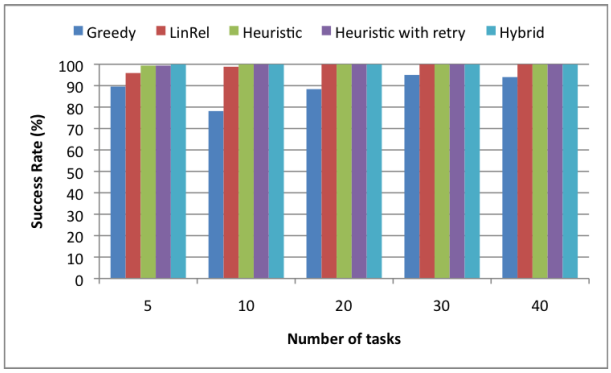


Figure 6: Success rate of the heuristics. Four CPUs and two GPUs with $\alpha = 2$. A success rate of 100% means that the heuristic found a schedule each time the linear solver found one.

a small number of tasks, Greedy and LinRel do not perform very well. As shown in Figure 4c, the success rate of our heuristic is 100%, whereas the Greedy heuristic does not succeed in finding a good schedule 20% of the time for 5 tasks, and 10% of the time for 10 tasks. Similarly, LR-heuristic also fails to find a feasible schedule for some small task sets.

Notice that the error is small (¡ 5%) for all our heuristics when α is 1.1 or 2.0. When α is 1.5, all the heuristics have a higher error. We believe the reason for this is that for values for α of 1.1 the constraint is very tight. So if the algorithm cannot find the correct schedule, it fails. Most likely there are very few feasible schedules. In that situation, we believe all the feasible schedules are likely to be similar. For values for α of 1.5 there are more feasible schedules, but the constraint is still tight and there is not much freedom for the algorithm to rearrange tasks or map a task to the best processor. In this situation, the heuristic can find a solution very different from the optimal, but still feasible. For values of α of 2.0, the constraint is looser, so it is much easier for all the algorithms to find a good schedule.

Finally, notice that the Hybrid heuristic is useful to reduce the error when Heuristic fails. Hybrid also helps improving the success rate. This is particularly useful for tight constraints as in Figure 4a when heuristics are more likely to fail. Finally, the heuristic with retry helps to slightly improve the success rate for the very tight schedules generated with $\alpha = 1.1$.

Configuration 2: Four CPUs and two GPUs Figure 5 and Figure 6 present the results for Configuration 2, that uses four CPUs and two GPUs, when $\alpha = 2$. Our heuristic stays under 2.5% of error for all the task counts considered. The Greedy and LR-heuristic improve a lot with an increasing number of tasks thanks to the greater freedom in scheduling allowed by more granularity. For $\alpha = 1.1$ or $\alpha = 1.5$ and for 1000 experiments run, only a small fraction gave a feasible schedule. Comparing the algorithms for such a small number of schedules would not be significant enough.

7.2 Sorting by Cross-Platform Task Heterogeneity vs. Sorting by Task Size

As discussed in Section 4.1, a key point in our heuristic is sorting tasks by decreasing cross-platform heterogeneity at the beginning of the mapping phase. One alternative logic could consider sorting the tasks by decreasing size to first map the largest tasks which could be the most difficult to fit in the schedule; this logic is still better than picking tasks in a random order or sorting them

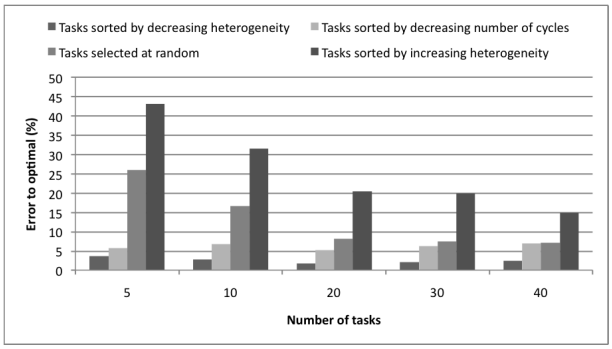


Figure 7: Error to optimal for the first hardware configuration when sorting tasks by decreasing cross-platform heterogeneity, by decreasing cycles, not sorting at all or sorting by increasing cross-platform heterogeneity, respectively. The y-axis shows the average error to the optimal for 1000 experiments with $\alpha = 2$

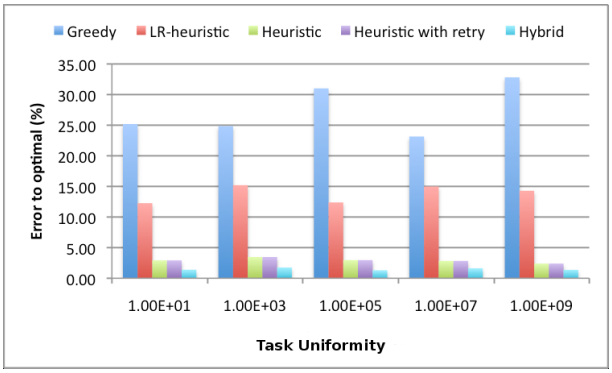
by increasing cross-platform heterogeneity, as shown in Figure 7. Notice our heuristic of sorting tasks by decreasing cross-platform task heterogeneity divides the error by more than 2 in most cases. Success rate for all experiments presented in this figure is greater than 99%.

7.3 Sensitivity Analysis

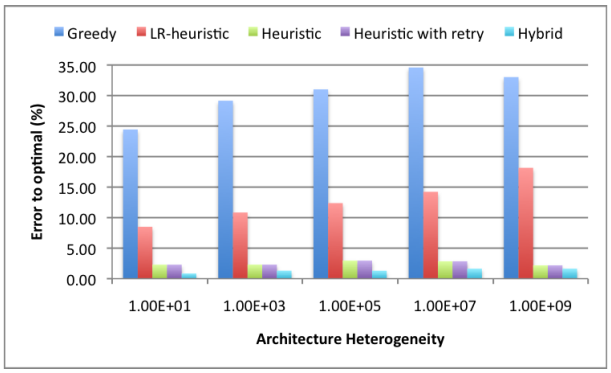
In this subsection, we analyze the sensitivity of the heuristics to the single-platform task uniformity and architecture heterogeneities. We chose the hardware configuration 1 and let the single-platform task uniformity τ and the architecture heterogeneity η vary respectively between 10 and 10^9. Figures 8a and 8b present the results. Our different heuristics are not sensitive to variations on the whole spectrum of heterogeneity considered. On the other hand, both the LinRel and Greedy are sensible to changes in η and τ; on Figure 8b, we see that Greedy and the LinRel are negatively impacted by an increase in architecture heterogeneity.

7.4 Execution Time of the Heuristic

A fast scheduler allows to test a lot of different configurations in a short time. It is also better for online scheduling, especially on real-time systems since the scheduling will consume time and power. In addition, if the scheduler is embedded in a compiler, it will allow shorter compilation times.



(a) Sensitivity to single-platform task uniformity τ: error to optimal as a function of τ



(b) Sensitivity to architecture heterogeneity η: error to optimal as a function of η

Figure 8: Error to optimal of the different heuristics as a function of the task uniformity and architecture heterogeneity.

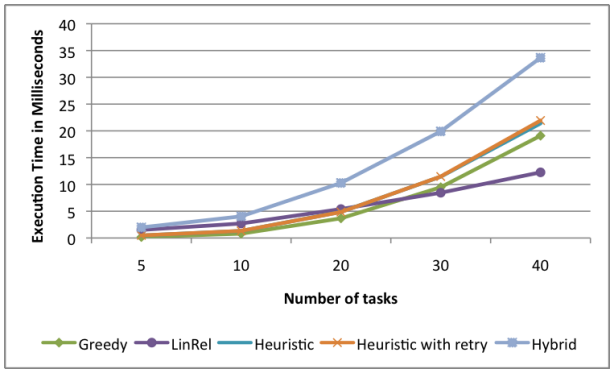


Figure 9: Comparison of the execution time in milliseconds as a function of the number of tasks for scheduling on the first hardware configuration with $\alpha = 2$. The heuristic and the heuristic with retry curve are almost the same because the retry only happens when no schedule is found.

#tasks	5	10	20	30	40
Exhaustive	8.05	88.27	13,000	172,470	420,090

Table 2: Execution time in milliseconds of the linear solver as a function of the number of tasks.

Our heuristic was implemented in C++ without specific performance optimizations. We also reimplemented in C++ the LR-heuristic and the Greedy heuristic presented in [22]. We used the *lp_solve* library [7] to solve the linear programming problems, both for the optimal case and for the LR-heuristic. For the optimal case, we simply make a call to the library, whereas for the LR-heuristic we wrap the call to *lp_solve* in some code controlling the different iterations of Algorithm 5. We ran 100 serial experiments on an Intel(r) Core(tm) i7 machine and timed each experiment with `gettimeofday()`. The results of these experiments are shown in Figure 9, where the average running time of each algorithm is plotted. We kept the exhaustive search numbers apart in Table 2 due to the huge difference in execution time.

Results on Figure 9 show that all the heuristics always perform very fast compared to the linear solver. The linear solver average execution time increases considerably as soon as the number of tasks increases. However, we noticed that in a large number of instances of the problem, even if the number of tasks is high, the execution time of the exhaustive search can be short depending on whether the branch-and-bound approach of the linear solver can eliminate more or less branches depending on the constraints. The average is very high because some instances of the problem take an extremely long time to explore. In the experiment presented in Figure 9, the heuristic with retry performs exactly the same as the heuristic because $\alpha = 2$ is a loose enough constraint that there is almost no need for retry. The hybrid heuristic execution time is higher and is exactly the sum of our heuristic and the LR-heuristic execution times.

7.5 Summary

Our experimental results in the previous Sections show that our Heuristic performs significantly better than Greedy and LR-Heuristic for those cases where there is little flexibility in the scheduling. In our experiments those cases appear when the value of α is low (α = 1.1) or when the number of tasks is small (20 or less). Our experiments on the Configuration 1 also show that when the number of tasks is small (20 or less) our Heuristic is faster than LR-Heuristic, and only slightly slower than Greedy, which performs very poorly. When the scheduling is easier due to extra freedom to map the tasks, all the algorithms obtain better results in terms of success rate and error rate. In particular our experiments for Configuration 1 show that for 40 tasks and α=2 LR-Heuristic is a good heuristic, as it runs faster than our Heuristic and have similar success and error rate. Notice that the are only two cases where the error rate of the LR-Heuristic is smaller than that of our Heuristic. Both cases occur in Configuration 1: the first one occurs with α = 1.5 and 20 tasks and the second one when α=2 and 40 tasks. In the first case our Heuristic has a higher success rate; in the second case (where all the heuristics have a 100% success rate), the error of our Heuristic is less than 3%. In addition, the Hybrid heuristic can take advantage of those cases where the LR-Heuristic performs better, although at the expense of some extra execution time.

Our results also show that the Greedy algorithm performs poorly; in fact, Greedy only performs well with large values of α (α=2) and large number of tasks.

Finally, our Heuristic seems very insensitive to architecture or task heterogeneity, because the heuristic that we use to do the mapping takes care of these variations.

8. Conclusions and future work

In this paper, we explored the scheduling of real-time tasks on a heterogeneous platform with energy minimization as a goal. Our heuristic offers a high success rate and significantly improves the state of the art heuristics, especially for small task sets. Our algorithm is stable in its results when exploring different task sets and platforms of different heterogeneities. Our experiments also evaluate the importance of the order in which the tasks are selected for scheduling. Furthermore, we have shown how to improve the results by combining two heuristics and how to improve the success rate by using the sensitivity of the scheduler to the tightness of the constraint.

Our scheduling strategy relies on a kernel composed of independent tasks and use software-pipelining to build this kernel of independent tasks. However, this can create a problem because it increases the pressure on cache and memory traffic, because now several items are in-flight at the same time. In the particular case of video postprocessing that we were studying, the number of frames in-flight will increase. Thus, we plan to study different approaches to refactor the task set for scheduling while improving the locality and reducing the data traffic.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Awards CCF 0702260, CNS 0509432, and by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign, sponsored by Intel Corporation and Microsoft Corporation.

References

[1] Amd fusion.http://fusion.amd.com.

[2] http://www.imgtec.com/news/release/index.asp?newsid=557.

[3] Mjpeg tools. http://mjpeg.sourceforge.net/.

[4] Shoukat Ali, Howard Jay Siegel, Muthucumaru Maheswaran, Sahra Ali, and Debra Hensgen. Task execution time modeling for heterogeneous computing systems. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 185. IEEE Computer Society, 2000.

[5] James H. Anderson and Sanjoy K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 428–435, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Hakan Aydi, Pedro Mejía-Alvarez, Daniel Mossé, and Rami Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium*, page 95. IEEE Computer Society, 2001.

[7] Michel Berkelaar, Kjell Eikland, and Peter Notebaert. Lpsolve. http://lpsolve.sourceforge.net/5.5/.

[8] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Chi-Sheng Shih. Energy-efficient real-time task scheduling in multiprocessor dvs systems. *Asia and South Pacific Design Automation Conference*, 0:342–349, 2007.

[9] Edward T.-H. Chu, Tai-Yi Huang, and Yu-Che Tsai. An optimal solution for the heterogeneous multiprocessor single-level voltage-setup problem. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(11):1705–1718, 2009.

[10] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[11] Jörg Henkel and Yanbing Li. Energy-conscious hw/sw-partitioning of embedded systems: a case study on an mpeg-2 encoder. In *CODES/CASHE '98: Proceedings of the 6th international workshop on Hardware/software codesign*, pages 23–27, Washington, DC, USA, 1998. IEEE Computer Society.

[12] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48. ACM, 2003.

[13] Tai-Yi Huang, Yu-Che Tsai, and Edward T.-H. Chu. A near-optimal solution for the heterogeneous multi-processor single-level voltage setup problem. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2007.

[14] Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 250–261. IEEE Computer Society, 2001.

[15] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202. ACM, 1998.

[16] Jian Li and José F. Martínez. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 2(4):397–422, 2005.

[17] Hui Liu, Zili Shao, Meng Wang, and Ping Chen. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 92–101, Washington, DC, USA, 2008. IEEE Computer Society.

[18] Jiong Luo and Niraj K. Jha. Power-efficient scheduling for heterogeneous distributed real-time embedded systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(6):1161–1170, 2007.

[19] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102. ACM, 2001.

[20] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sep 2010.

[21] Chuan-Yue Yang, Jian-Jia Chen, Tei-Wei Kuo, and Lothar Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *ACM/IEEE Conference of Design, Automation, and Test in Europe (DATE)*, pages 694–699, 2009.

[22] Yang Yu and Viktor K. Prasanna. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *ICPADS '02: Proceedings of the 9th International Conference on Parallel and Distributed Systems*, page 341. IEEE Computer Society, 2002.

[23] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163. ACM, 2003.