6

Fast and Scalable Run-time Scheduling

As explained in previous chapters, a run-time scheduler is indispensable to efficiently explore the design space and make system level trade-off according to the dynamic context. For that sake, a fast and effective heuristic is needed. In this chapter, we first review again why we need a two-phase approach for task scheduling and how it is applied. The problem is then defined in a more formalized way and a greedy heuristic is described. After that, experimental results on both randomly generated and real-life applications are explained. In this chapter, we will illustrate our method on 2-dimensional Pareto trade-offs with execution time vs energy as axes. But the underlying techniques can also be applied to other axes and more dimensional trade-offs, which will be demonstrated in the next chapter.

6.1 Two-Phase Task Scheduling: Why and How

The design of concurrent real-time embedded systems, and embedded softwares in particular, is a difficult problem, which is hard to perform manually due to the complex consumer–producer relationships, the presence of various timing constraints, the nondeterminism in the specification and the sometimes tight interaction with the underlying hardware. Our TCM methodology provides a novel and effective cost-oriented approach to the concurrent task-scheduling problem, by carefully distinguishing what can be modeled and optimized at design time from what can only (or better) be done at run time.

As shown in Chapter 3, we model applications with TNs and TFs. The design-time scheduling is applied on the thread nodes inside each TF at compile time, including a processor assignment decision of the TNs in the case of multiple processing elements. On different types of processors of a heterogeneous platform, the same TN will be executed at different speeds and

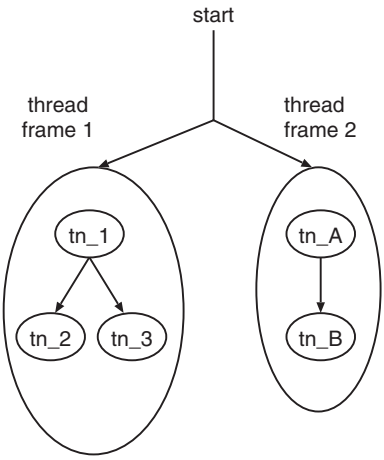

Fig. 6.1. The gray-box model of a simple example

Table 6.1. The execution time and energy consumption of TNs in Fig. 6.1

	Processor 0					Processor 1				
	1	2	3	A	B	1	2	3	A	B
Execution time (μs)	10	30	15	20	32	20	60	30	40	64
Energy (μJ)	30	86	41	75	90	8	22	10	19	23

with different costs, i.e., energy consumption in this chapter. These differences provide the possibility of exploring the cost-performance trade-off at the system level.

The idea of our two-phase scheduling can be illustrated with the simple example in Fig. 6.1. Here we assume a dual-processor platform. For the five thread nodes in that example, we assume they have different execution times and energy consumptions on different processors. The numbers are summarized in Table 6.1.

Now for every TF, the design-time scheduler will try different mapping and ordering of the TNs of that TF, satisfying all the dependency and time constraints. An example is given for TF 1 in Fig. 6.2, where the execution time and energy consumption are shown also. When TN 2 and TN 3 are assigned to the same processor (e.g., (d)), it makes no difference which one has to be executed first. For simplicity, we show only one possible order. However, extra constraints may exist and they will further fix the order. The design-time scheduling result can be represented as a Pareto curve and it is shown in Fig. 6.3. From that figure, we can see that not all scheduling decisions are beneficial. For instance, (a) and (e) neither run faster nor consume less energy compared with all the other schedulings. We say they are dominated or they

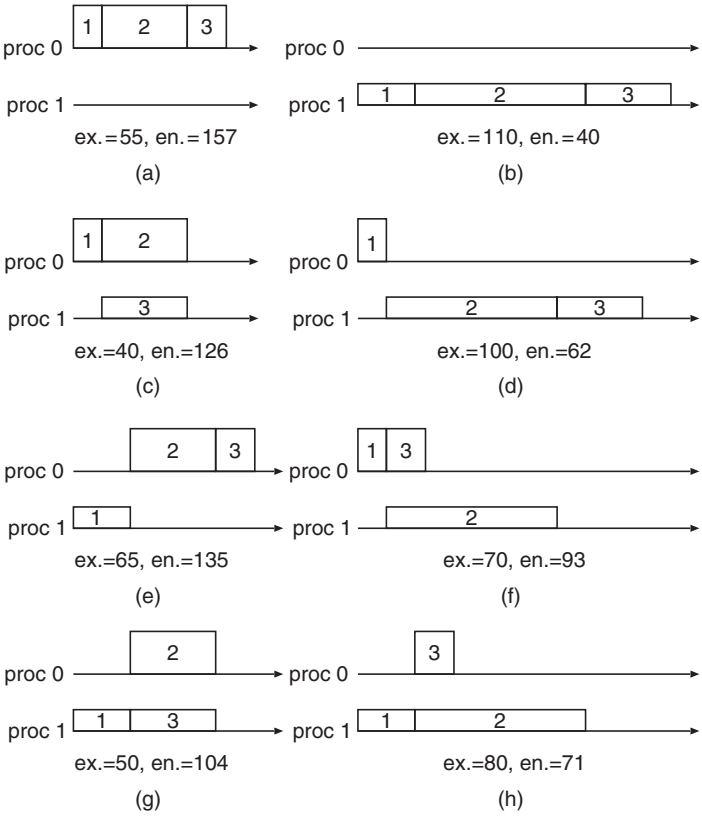

Fig. 6.2. The design time scheduling of thread frame 1

are not on the boundary of a Pareto curve. Similar results can be obtained for TF 2 (see Figs. 6.4 and 6.5).

Here we illustrate only a simple example. With the increase of the number of TNs/processors, complex inter-TN dependencies and time constraints, it becomes impractical to do the design-time scheduling by hand. Therefore an automatic tool, known as the TCM design-time scheduler presented in Chapters 4 and 5, is needed.

Only at run time the system-level information required to decide on a cost-effective schedule meeting all real-time constraints will be complete. First of all, the active scenario is identified (see Chapter 3). Next, given the number of TFs, the Pareto curve of each TF and system constraints such as the global deadline, the run-time scheduler will select a mapping and/or ordering decision pre-computed by the design-time scheduler for every active TF and combine them together to get the system scheduling. For the above example, when the global deadline is 125 μs, the run-time scheduler will select design-time scheduling (g) for TF 1 and (c) for TF 2,

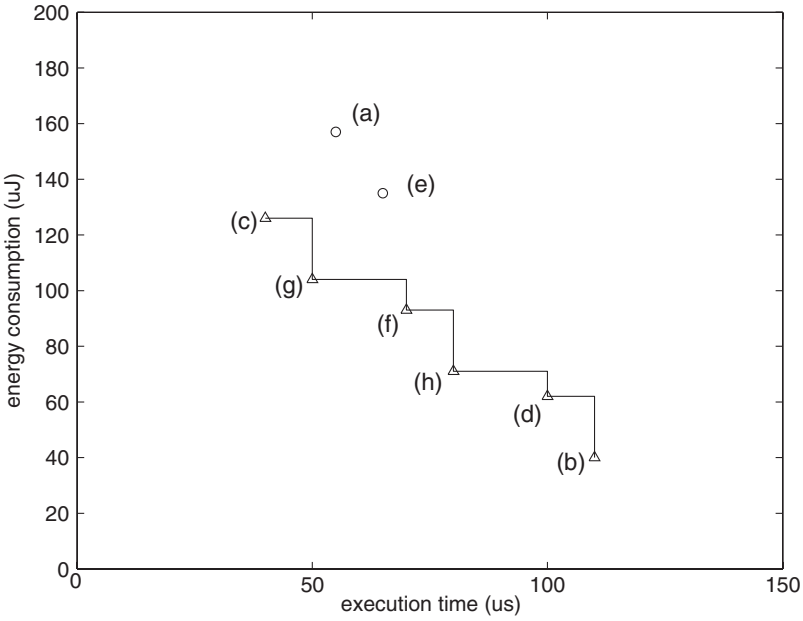

Fig. 6.3. The Pareto curve of thread frame 1. Scheduling (a) and (e) are not on the curve

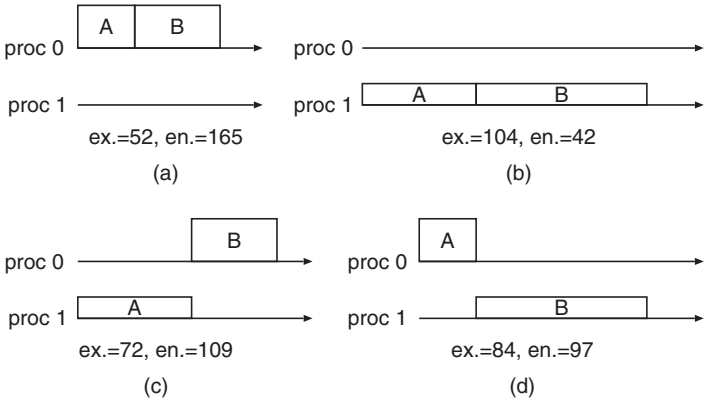

Fig. 6.4. The design time scheduling of thread frame 2

combine them together and find the system scheduling with the minimum energy consumption. The main goal of this chapter is to solve the problem of how to find the global scheduling and how to support it with implementable run-time systems on real platforms.

Given a TF, our design-time scheduler will try to explore different assignment and ordering possibility, and generate a Pareto-optimal set [167], where

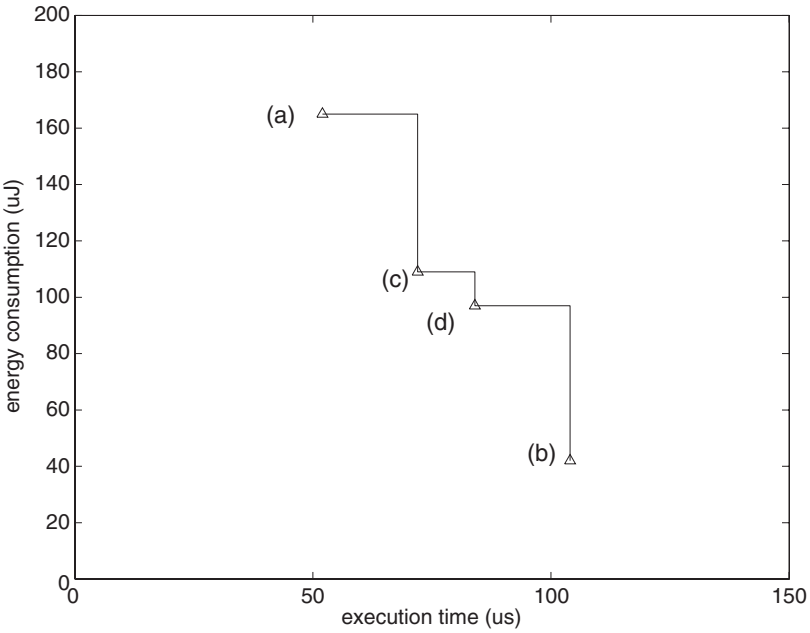

Fig. 6.5. The Pareto curve of thread frame 2

every point is better than any other one in at least one way, i.e., either it consumes less energy or it executes faster. This Pareto-optimal set is usually represented by a Pareto curve. Since the design-time scheduling is done at compile time, computation efforts can be paid as much as necessary, provided that it can give a better scheduling result and can reduce the computation efforts of run-time scheduling in the later stage. However, if very data-dependent behavior is present inside the TF, the design-time exploration still has to assume worst-case conditions to guarantee hard real-time requirements. In such a case, a TF can be further classified into a few typical execution scenarios to give a more accurate prediction.

At run time, the run-time scheduler will then work at the granularity of TF. Whenever new TFs are initiated, the run-time scheduler will try to schedule them to satisfy their time constraints with an aim to minimize the system energy consumption. The details inside a TF, like the execution time or data dependency of each thread node, can remain invisible to the run-time scheduler and this reduces its complexity significantly. Only essential features of the points on the Pareto curve will be passed to the run-time scheduler by the design-time scheduling results, and will be used to find a reasonable cycle budget distribution for all the running TFs.

In summary, we separate the task scheduling into two phases, namely design-time and run-time scheduling, for three reasons. Firstly, it gives more run time flexibility to the whole system. We can indeed accommodate more

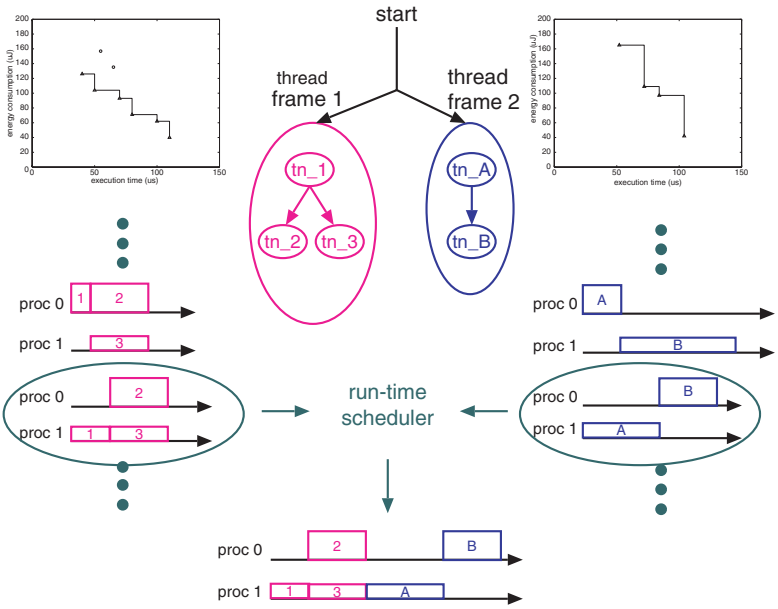

Fig. 6.6. When the global deadline is 125 µs, the run-time scheduler selects design-time scheduling (g) for TF 1 and (c) for TF 2, combines them together and finds the system scheduling

unforeseen demands for more execution time by any TF, by "stealing" time from other TFs, based on their available Pareto sets. Secondly, we can minimize energy for a given timing constraint that usually spans several TFs by selecting the right combination of points. Finally, it minimizes the run-time computation complexity, which reduces the energy and time penalty so that faster reaction time can be achieved (up to 1 ms). This is needed for modern multimedia and wireless communication applications. The design-time scheduler works at the gray-box level but still sees quite a lot information from the global specification. The end result hides all the unnecessary details and the run-time scheduler can operate mostly on the granularity of TFs, not single TNs. Only when a large amount of slack is available between the TNs, a run-time local refinement on the TF schedule points can result in further improvements.

This methodology can in principle be applied in many different contexts as long as Pareto-curve like trade-offs exist. For example, in the context of DVS, the cost can be the energy consumption. Thus our methodology results in an energy-efficient system. When the cost is energy and the horizontal axis is replaced by the QoS, the problem becomes the energy minimization with a guaranteed QoS, as e.g., formulated in [183]. Also the deadline miss rate can be optimized in soft hard real-time applications (e.g., video decoding) for a given platform and a set of deadlines.

6.2 Run-Time Scheduling Algorithm

In the previous section, we have shown the concept of our two-phase, Pareto-curve-based scheduling methodology. The key step of this method is the run-time scheduler. Given a set of TFs and a deadline, the run-time scheduler has to select one and only one point from the Pareto curve of each TF and combine them into the final scheduling. It has to be done fast because that will allow a more frequent (re)evaluation of the run-time scheduling decision or the handling of more tasks in a single shot. Both will result in still more energy savings. The quality of the solution is also important because it affects the amount of energy which can be saved.

In this section, we will first formulate the problem in a formal mathematical model. Then a greedy heuristic is proposed for our specific problem.

6.2.1 Application Model

We model applications as a set of interacting TFs, which have to be mapped to a multi/uni-processor platform. We mainly consider the frame-based systems, which issue a set of TFs when the input data is ready (normally it is the start of a time frame or period).[1] Most typically, there is an end-to-end deadline by which all TF should finish. Examples of this kind of system include MPEG2 decoding and MP3 decoding. Therefore, we have the following application model.

- At the beginning of every time frame, there are k TFs waiting to be executed, each represented by a Pareo curve.
- Each TF i has N_{ij} Pareto points, i.e., N_{ij} different ways of mapping and ordering on the given platform and they are represented with their execution time t_{ij} and energy consumption e_{ij}.
- At any moment, only one TF can be executed on the given platform. In other words, that TF occupies the platform exclusively.
- There is a global deadline D before which all the TF have to finish.

The run-time scheduling problem can be stated as picking a mapping/ordering pattern for every active TF and minimizing the total system energy consumption while meeting the global deadline.

In most situations, dependencies exist between TFs (e.g., TF 2 can only start after TF 1 and TF 4 finish). These dependencies can be handled by assigning priority levels to TF and the priority levels can be decided at design time. Hence the dependencies will not impact the scheduling algorithm we present later, though they will require the final run-time system to identify the TF priority levels and react appropriately.

[1] Aperiodic TF sequence is just a special case of this model, for which we have only to consider one time frame.

6.2.2 Problem Formulation

For the application given above, we can formulate our run-time scheduling as follows. Since k TFs exist and each of them has N_i Pareto points, we can introduce an integer variable x_{ij} to denote whether the jth Pareto point of TF i is selected (x_{ij} equals 1) or not (x_{ij} equals 0). For each TF, one and just one Pareto point can be selected, which leads to:

$$\sum_{j=1}^{N_i} x_{ij} = 1, i = 1, \ldots, k$$

For a Pareto point i of TF j, the execution time of that TF is t_{ij} and the energy consumption is e_{ij}. The total system execution time can never exceed the global deadline D for real-time systems. Therefore we have:

$$\sum_{i=1}^{k} \sum_{j=1}^{N_i} t_{ij} x_{ij} \leq D$$

The goal of our run-time scheduler illustrated for the 2D execution-time vs energy trade-off is to reduce the total system energy consumption as much as possible. This can be represented as:

$$minimize : z = \sum_{i=1}^{k} \sum_{j=1}^{N_i} e_{ij} x_{ij}$$

Putting the above equations together, we have a constrained minimization problem.

$$\begin{aligned} minimize : \quad & z = \textstyle\sum_{i=1}^{k} \sum_{j=1}^{N_i} e_{ij} x_{ij} && (6.1) \\ subject\ to \quad & \textstyle\sum_{i=1}^{k} \sum_{j=1}^{N_i} t_{ij} x_{ij} \leq D, && (6.2) \\ & \textstyle\sum_{j=1}^{N_i} x_{ij} = 1, i = 1, \ldots, k, && (6.3) \\ & x_{ij}\ is\ 0\ or\ 1, i = 1, \ldots, k,\ j = 1, \ldots, N_i. && (6.4) \end{aligned}$$

The total number of Pareto points can be denoted by n, $n = \sum_{i=1}^{k} N_i$.

The minimization problem can be transformed into a different form [155]. Taking into account that each Pareto curve is an ordered set, we can substitute e_{ij} with s_{ij} as

$$s_{ij} = (e_{i0} - e_{ij}), s_{ij} \geq 0. \quad (6.5)$$

Thus Eq. 6.1 becomes a maximization problem:

$$maximize : z^{'} = \sum_{i=1}^{k} \sum_{j=1}^{N_i} s_{ij} x_{ij} \quad (6.6)$$

With the same set of constraints, this is a classic Multiple Choice Knapsack Problem (MCKP) and it is known as NP hard [151].

When of limited size, MCKP can be solved optimally in pseudo-polynomial time through dynamic programming (DP). For bigger instances, it is generally solved by a DP algorithm constructed from the exact solution of its linear relaxation, LMCKP, by replacing Eq. 6.4 with

$$0 \leq x_{ij} \leq 1, i = 1, \ldots, k, j = 1, \ldots, N_i. \tag{6.7}$$

Several exact algorithms have been proposed to solve the reduced LMCKP problem in $O(n)$ time [151]. To evaluate the result of our algorithm, we use the DP algorithm presented in [172]. However, the worst-case computation complexity of DP is still exponential, which is not acceptable as an run-time algorithm for medium problem size. Another issue is that the computation time of DP is nondeterministic, which is undesirable for real-time systems.

Several approximate algorithms exist for MCKP but all have limitations or are not suitable for our problem. Current heuristics are designed for big problems, which can not be solved easily by any accurate algorithm due to the problem's NP-hard feature. They rival each other in which can get a solution closer to the optimal value or which can handle a bigger (or more difficult) problem. Execution time is only the second or the third concern to them, which makes them unsuitable to work as an run-time algorithm. In addition, most of the heuristics do not recognize that in our case, all points are already Pareto optimal and ordered. That can save quite extra computation effort.

The goal of our heuristic is to find a good enough solution in as short as possible time for a typical problem size. It is not our major interest to improve the solution by 1% if it means 2 times longer execution time. Moreover, the heuristic should be interruptible, which improves its solution incrementally in every iteration so that it can be interrupted if the time slot assigned to the run-time scheduler expires. Then it returns its best solution at that moment. This can guarantee a deterministic computation time for the run-time scheduler.

6.2.3 Greedy Heuristic

We have developed a fast and effective greedy heuristic with the above considerations in mind. Algorithm 5 consists of two stages, the initialization (line 2–16) and the iteration stage (line 18–41). Every point i of our Pareto curve m is denoted by two basic parameters, $t_{m,i}$ and $e_{m,i}$, standing for the execution time and energy consumption if that point is selected by the scheduler (the corresponding concepts in MCKP are weight and profit). D is the deadline. In the initialization stage, we compute the changes of t and e if we move to the right (from point i to $i+1$, see Fig. 6.7) or to the left (from point i to point $i-1$) and the corresponding slopes (line 5–12). Here a superscript "$+$" means the rightward direction and "$-$" means the leftward direction. The initial solution is found at line 13 and 14: a portion of the deadline (s_m) is assigned

Algorithm 5 The greedy heuristic algorithm.

INITIALIZATION
step 0:
slack=0;
for all curve m **do**
  for all point i on curve m **do**
    $\delta e^+_{m,i} = e_{m,i} - e_{m,i+1}$;
    $\delta e^-_{m,i} = e_{m,i-1} - e_{m,i}$;
    $\delta t^+_{m,i} = t_{m,i+1} - t_{m,i}$;
    $\delta t^-_{m,i} = t_{m,i} - t_{m,i-1}$;
    $slope^+_{m,i} = \delta e^+_{m,i} / \delta t^+_{m,i}$;
    $slope^-_{m,i} = \delta e^-_{m,i} / \delta t^-_{m,i}$;
  end for
  $s_m = t_{m,0} D / \sum_{l=0}^{k-1} t_{l,0}$;
  search for maximal j with $t_{m,j} \leq (s_m + slack)$;
  update slack;
end for
ITERATIVE IMPROVEMENT
step 1:
sort $slope^+$ descendingly and $slope^-$ ascendingly;
for all curve m in $slope^+$ **do**
  for all curve n in $slope^-$ and $m \neq n$ **do**
    if $slope^+_m \leq slope^-_n$ **then**
      goto step 2;
    end if
    if $\delta e^+_m > \delta e^-_n and \delta t^+_m < \delta t^-_n + slack$ **then**
      change solution of curve m from i to $i+1$;
      change solution of curve n from j to $j-1$;
      update slack;
      goto step 1;
    end if
  end for
end for
step 2:
sort $slope^+$ descendingly;
for all curve m in $slope^+$ **do**
  if $\delta t^+_m < slack$ **then**
    change solution of curve m from i to $i+1$;
    update slack;
    goto step 2;
  end if
end for

to a curve proportional to the execution time of its leftmost point. Therefore it guarantees a valid initial solution can always be found for that curve. For finding the initial solution we use an on-the-fly strategy. The difference between the time assigned to curve m and the actual execution time of its

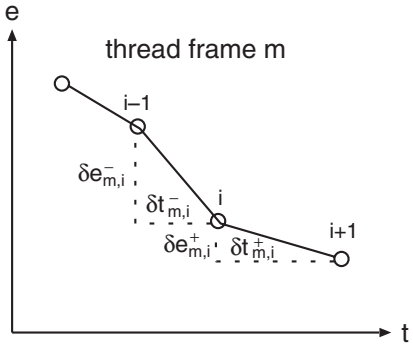

Fig. 6.7. The Pareto curve of thread frame m

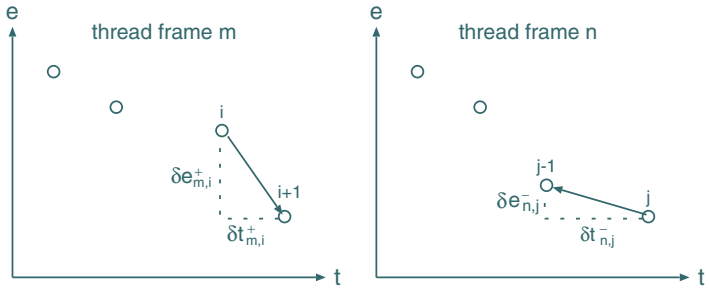

Fig. 6.8. Incremental improvement step 1, when the operating points change from (i, j) to $(i + 1, j - 1)$. $\delta t^+_{m,i} < \delta t^-_{n,j} + slack$ and $\delta e^+_{m,i} > \delta e^-_{n,j}$ have to be satisfied to get a valid and meaningful solution

initial solution will be accumulated in the variable $slack$ and added to the available time of the following curves.

After the initialization, we explore the chances of finer tuning the solution in two steps, $step1$ and $step2$. $step1$ checks the possibility of moving the operating point on one curve to the right and the operating point on another curve to the left in pair. At line 19, all curves are sorted according to the slopes of their current solutions, $slope^+$ descendingly and $slope^-$ ascendingly. Then the algorithm will try to find two curves m and n, which satisfy the time constraint and reduce the energy consumption most, when the solution of m is changed from i to $i + 1$ and the solution of n from j to $j - 1$ (Fig. 6.8). When no such kind of tuning is possible, the algorithm will enter the next step.

$step2$ does the final tuning by finding any curve m which can still satisfy the time constraint if we move its current solution from i to $i+1$. It is possible to switch the order of these two steps. However, our experiments show the current order is faster and generally leads to better solutions. Another option

is to move the operating point to the right as much as possible in $step2$. In that case, if $step2$ is done before $step1$, this will cause the heuristic to converge in fewer iterations but deteriorate the optimality of the final solution.

Assuming k curves and l points are present on each curve, the complexity of the initialization step is $O(k \log l)$ because for every curve we have only to do an ordered search (line 14). The complexity of the iterative stage is also very low. In $step1$ every iteration takes maximally $O(k^2)$ operations, while in $step2$ $O(k)$ operations. The heuristic ends when no improvement is possible, but we can interrupt the iteration at any moment to finish the run-time scheduling in a predefined time slot. In that case the algorithm just returns the best available solution. This capability is very important for a real-time system where bounded and deterministic service is always desirable. The performance of our greedy heuristic is illustrated in Section 6.3.

6.3 Experimental Results

We have implemented the greedy algorithm in C and tested it with both randomly generated and real-life applications. They are discussed separately in the following sections.

6.3.1 Randomly Generated Test Cases

The first test set we have used is based on random task graphs generated by TGFF. For each task graph, a Genetic Algorithm [250] is used to extract the Pareto curve. Finally the heuristic is applied to find the run-time task scheduling within a given deadline. A DP optimal algorithm [172] is used in this step to check the speed and quality of our heuristic.

We have generated three task sets with TGFF, containing 5, 10, and 20 task graphs, respectively. For every task graph, we have extracted two Pareto curves, one with 5 points and the other with 9 points. The former is just a subset of the latter. The points are distributed almost uniformly, in the sense of execution time, between the lowest and highest possible values. Different deadlines are then tried for the same task set and the same Pareto curves and the results are summarized in Tables 6.2 and 6.3.

The performance of our heuristic can be evaluated in two ways: the execution time and the quality of the result. Tables 6.2 and 6.3 give the overview of the result. In the tables, the first column is the number of curves; the second column is the average speedup of the execution time of the greedy heuristic against the DP solver; the third column is the maximum speedup; the fourth column gives the average error between the heuristic and DP solution; and the fifth column is the maximum error. The next four columns are the same as column 2–5 but for the initial solution given by $step0$ of Algorithm 5.

Table 6.2. The performance of the greedy algorithm compared to DP, 5 points per curve

No. Pareto curves	Average speedup	Max speedup	Average error	Max error	Average initial speedup	Max initial speedup	Average initial error	Max initial error
5	14.9	24.0	1.2%	5.2%	44.0	58.7	4.1%	9.1%
10	8.8	13.2	1.0%	2.9%	42.9	53.3	6.8%	13.4%
20	3.9	7.3	1.0%	2.0%	24.0	50.2	4.5%	8.7%

Table 6.3. The performance of the greedy algorithm compared to DP, 9 points per curve

No. Pareto curves	Average speedup	Max speedup	Average error	Max error	Average initial speedup	Max initial speedup	Average initial error	Max initial error
5	15.4	24.9	0.6%	3.5%	46.0	65.1	3.4%	10.3%
10	8.4	14.5	0.8%	2.1%	34.5	55.6	4.1%	8.7%
20	4.3	7.7	0.9%	1.9%	26.2	43.4	3.5%	7.0%

The results show that our heuristic achieves average speedup up to 15 times gainst the optimal solver, while maintaining a very high solution quality (error within 1.2% on average). If the initial solution is considered, the average speedup is up to 46 times while the solution error is up to 6.8%, on average. This is quite acceptable for an run-time scheduling algorithm, because if the optimal solution means an energy reduction from 1000 nJ to 500 nJ, a 10% error just means the energy is reduced to 550 nJ, which is already a big improvement compared to the original value, especially if we take into account the high speed to find the initial solution.

For the run-time scheduling stage, the time spent on the scheduler itself will not contribute to executing the application functionality. So it has to be minimized or bounded, even though we can have a separate CPU to run the scheduler in some architectures. Our heuristic provides the capability of improving the initial solution iteratively until the time slot assigned to the scheduler depletes. This is especially important for big problem sizes, when the scheduler could not run to its end and still has to find a solution in a short time slot. Table 6.4 shows an example of the iterative improvement of our heuristic. This example is for the 9 points per curve, 20 curves case because it is the worst case in our experiment with respect to the execution time. The optimal result is 37836 nJ and it takes the DP 232 k processor cycles to find it. With the heuristic, to find the final solution 38443 nJ, it takes 119k cycles, which may be too long. However, the final solution is only 1.6% from the optimal one and we are usually already satisfied with solutions which are not that good but can be found rather fast. If we assume we have 50 k (100 k)

Table 6.4. The iterative improvement of the heuristic for a 20 curves, 9 points case

Iterative #	Time (cycles)	Energy (nJ)
0	11554	39366
1	36909	39102
2	48201	38857
3	59389	38695
4	70700	38640
5	81939	38556
6	93502	38538
7	103381	38526
8	113225	38463
9	119312	38443

cycles available for the scheduler, which is 0.25 ms (0.5 ms) on a 200 MHz processor, the result we can find is 38857 nJ (38538 nJ) and it is only 2.7% (1.9%) away from the optimal solution. Even the initial solution is acceptable in this case, which can be found in less than 12 k cycles. Given the fact that the run-time scheduler is triggered by external events (e.g., user related) at the frequency of tens of ms, this result is quite good.

6.3.2 Real-Life Applications

We have also tried our heuristic on some real-life applications. One example is the QoS adjustment algorithm of a 3D image rendering application (this experiment has been explained in [253] in detail). On the start of each time frame, depending on the number of visible objects and which kind of objects they are, the QoS controller will adjust the number of vertices assigned to each object, in order to provide the best quality at a fixed computation power. Figure 6.9 illustrates the energy consumption of QoS adjustment algorithm for 1000 frames, with a frame rate of 5 fps (frame per second) or 10 fps. From this figure it is obvious that our run-time scheduler can achieve a very high energy saving (65% for 5 fps and 46% for 10 fps). The inter-task DVS does not work very well here because the number of task graphs and the execution time of each task graph varies dramatically in this application. Having to assume the worst case for the unscheduled task graphs, the inter-task DVS scheduler has a limited chance to scale the voltage. Another observation is that the difference between the greedy heuristic and the DP is very small. This is because, during most of the frames, the heuristic can easily find the optimal solution due to the limited problem size.

Another real-life application we have experimented on is the Visual Texture Coding (VTC) decoder of the MPEG-4 standard. Similar to the QoS example, it is frame based. However, unlike the highly dynamic number of objects in QoS, the number of blocks to be decoded is fixed (3 in this experiment) for every frame, though the workload of each block varies from frame

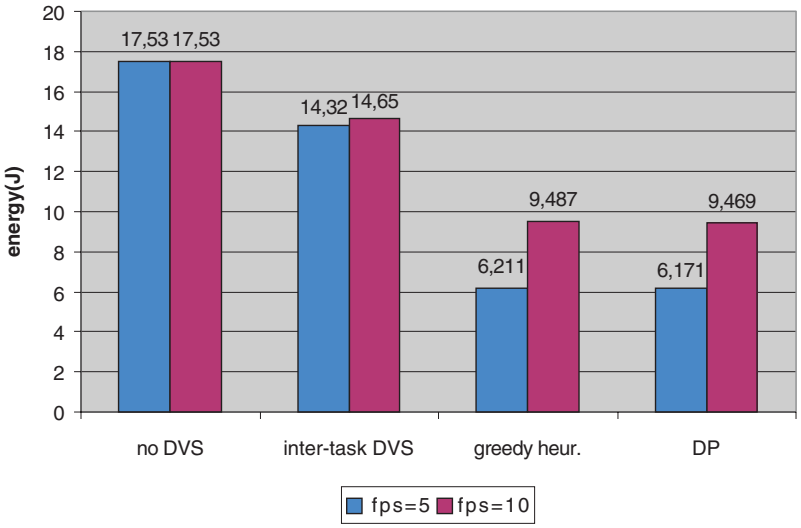

Fig. 6.9. The energy consumption of QoS adjustment algorithm for 1000 frames

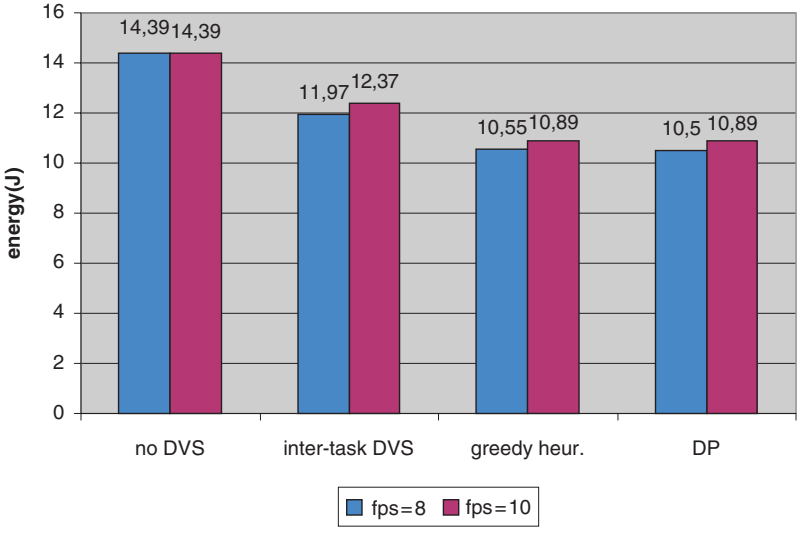

Fig. 6.10. The energy consumption of the VTC decoder for 1365 frames

to frame (see [143] for further discussion). As shown in Fig. 6.10, this example gives less space for voltage scaling because of its relative high and less varying work load. This is mainly due to the sequential feature of the initial task graph.[2] In spite of that, our heuristic still outperforms the inter-task DVS

[2] It can be removed after applying TCM transformation step.

and provides an energy saving of 27%. Again the results from the heuristic and DP are very close.

6.4 Summary

In this chapter we have modeled the Pareto-optimization-based run-time task scheduling as the Multiple Choice Knapsack Problem and have proposed a greedy heuristic for it. Results from randomly generated and real-life applications prove that our heuristic is fast (speedup of more than 10) and accurate (suboptimality less than 5%). The incremental and scalable feature makes the heuristic well suitable for our run-time task scheduling context.