4 Basic Design-Time Scheduling

In this chapter, we present the development of a novel task scheduling algorithm for the design-time scheduling phase of our two-phase TCM framework. *Scheduling* has different meanings in different contexts. In this chapter, scheduling has two meanings. First, assigning every thread node of a given TF to one of the processors of a given platform. Second, deciding the start time to execute every node on its assigned processor. We have two assumptions for this scheduling problem. First, whether a node can be executed on a certain processor should be known. Second, if a node can be executed on a certain processor, the execution time should be known, calculable or predictable. Based on the second assumption, having decided the start time of executing a node on a certain processor, the scheduler can derive the finish time of such an execution immediately. Therefore, the execution order of all the nodes assigned to each processor is known. In summary, scheduling in our context means node-to-processor assignment and ordering.

We will first formulate the problem. Then the kernel heuristic is presented. Next, a number of extensions are introduced to improve the kernel heuristic one way or another. We will use extensive experiments to demonstrate at each development phase where the strong point and weak point are, which we should keep and which we should improve. Finally, the scheduler is extended with a pruning step to handle the timing constraints imposed between two nodes.

4.1 Problem Formulation

A graph consists of *node*s and *edge*s as shown in Fig. 4.1. The *edge* represents a dependency between two *node*s. The dependency can be a control dependency or a data dependency. From the execution viewpoint, a dependency dictates the precedence constraint between two *node*s, i.e., the *node* at the end

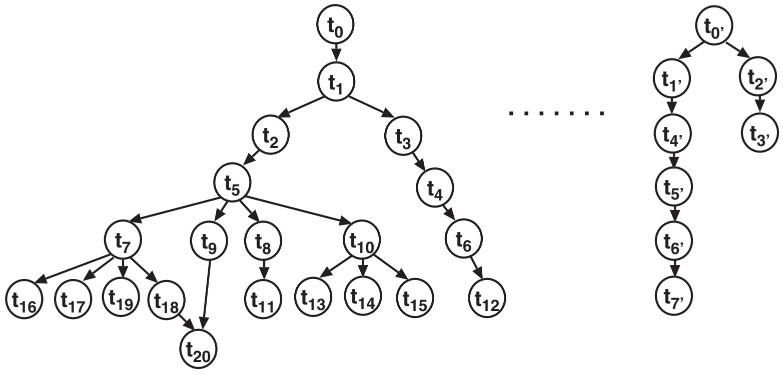

Fig. 4.1. An example graph

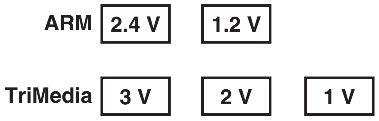

Fig. 4.2. An example platform

of the *edge* can only start execution when the *node* at the start of the *edge* finishes execution.

A node, which does not have a parent, is a *source* node, like node t_0, t'_0 in Fig. 4.1; while a node, which does not have a child, is a *sink* node, like node t_{16}, t_{17}, t'_7, ... It is possible for a node to be both a *source* and a *sink* node. A graph can have one or multiple source nodes.

A platform has a number of processors as shown in Fig. 4.2. These processors can be of the same type or of different types. And they can have control knobs allowing them to exhibit different Pareto trade-off points. In Chapters 4 and 5, we will focus on the trade-off between execution time (T) and energy (E) as an illustration. The approach can be readily extended to more Pareto axes though. Moreover, we will illustrate the control knobs by the presence of (a discrete range of) V_{dd} voltages. So we assume that the processors and other resources can operate in a given range of different supply voltages.

According to the recent literature [104, 119, 171], some of today's processor cores have the supply voltage varying from 0.9 to 5.0 V. Within this supply voltage range, the execution time of a node scales inversely proportional with the supply voltage while the energy consumption scales quadratically with the supply voltage.

$$T \propto \frac{1}{V_{dd}} \tag{4.1}$$

$$E \propto V_{dd}^2 \tag{4.2}$$

These two formulas are known as the execution time model and energy model, respectively.[1]

Actually, our scheduling algorithm does not require the execution time model and the energy model to be strictly kept as Eq. (4.1) and Eq. (4.2). What the scheduler needs are execution time and energy data of every node on every processor. As long as the user provides these data, either in a table format or in a model formula format, the scheduler can work with those data. However, to have performance and energy-cost trade-offs, the scheduler requires that the energy consumption increases at a faster pace than the execution time decreases when the supply voltage changes. The reason is as follows. If we take the decrease of execution time as *performance gain*, and the increase of energy consumption as *loss*, since we gain more in the performance than we lose in the energy consumption when raising the supply voltage, we may always favor the high supply voltage solution. Therefore we have little trade-off to play between the *performance gain* and *energy consumption*. Though semantically, the scheduler does not impose this requirement, i.e., a user can feed whichever data to the scheduler, the scheduler cannot derive trade-off curves if the problem itself does not allow trade-off between performance and energy-cost.

4.2 Exact Scheduling Algorithms

An exact scheduling normally has to exhaustively search the complete search space. However, the more we know the problem, the more efficient algorithm we can come up with to solve that problem specifically. For example, branch-and-bound is a well-known framework for exhaustive search. It is a framework because it only dictates the guidelines of how to make scheduling decisions, namely, branching rule, lower bound calculation, search strategy, and dominance rule. Actually, branch-and-bound is a generic framework for decision-making, not restricted with scheduling decisions. Then it is the responsibility of the algorithm developer to implement the abstract guidelines to concrete operations. The more we know the features of a problem, e.g., in our context, the topology of the graph, how the node execution time is distributed in the graph, etc., the better we can tune and implement the concrete operations to solve the problem efficiently. In other words, the tuned algorithm only goes over a small fraction of the search space, and hence finds solutions faster. However, the more specific an algorithm is to a problem, the more it looses in its generality. In other words, for a different problem, it can be extremely slow to find the solutions. High sensitivity to the input data should always be avoided. That is the motivation for us to employ the

[1] Note these are approximated formulas for V_{dd} above or around 1 V. In the future, when processing technologies go below 45 nm, the V_{dd} range will become much lower than 1 V.

heuristic approach, by which we can prune the search space significantly no matter which input data is given.

For our problem, the scheduling complexity comes from the following two origins.

1. The combinatorial complexity. This is due to the node-to-processor assignment possibilities. Assume we have a graph of n nodes and a platform of m processors, the possible assignment number is m^n.
2. The permutation complexity. For each fixed node-to-processor assignment, a number of node execution orders exist. Assume for node-to-processor assignment i, we divide the ordering process into s_i steps. In each step j, we order the execution of a number of a_j independent nodes. The number of possible orders at step j is $a_j!$.

Therefore, the total complexity of our scheduling problem calculated by the following equation.

$$\sum_{i=1}^{m^n} \prod_{j=1}^{s_i} a_{ij}! \tag{4.3}$$

$$\sum_{j=1}^{s_i} a_{ij} = n \tag{4.4}$$

The complexity depends on how we divide the ordering process into steps for each node-to-processor assignment. In the worst case, all the nodes are independent from each other, i.e., no precedence constraint exist between any two nodes. Then for each node-to-processor assignment, we only have one step to order the node execution. The number of node execution orders is $n!$. This case has the largest ordering freedom. No matter how we divide the ordering process into steps, for a realistic size problem, apparently, the design space is far too large to explore exhaustively.

Example A 40-node graph is going to be scheduled on a 4-processor platform. To produce one scheduling, i.e., assignment plus ordering, it takes 1 clock cycle. We have a computer with 1 GHz clock frequency.

Case 1 Use the worst case to get a rough estimation. For one node-to-processor assignment, the number of node execution orders is $40!$, which equals to 8.15×10^{47}. Then to exhaustively search the possible execution orders, it will take the computer 2.58×10^{31} years. Apparently, this is not realistic.

Case 2 Assume then the ordering process is divided into ten steps. At each step, we have four independent nodes to be ordered. Then for one node-to-processor assignment, the number of node execution orders is $(4!)^{10} = 6.34 \times 10^{13}$. To exhaustively search the possible execution orders, it will take the computer 17.61 h. Since this is for one node-to-processor assignment, which takes all the possible node-to-processor assignments into account, the total scheduling number should be multiplied by 4^{10}. The result will be 2107 years.

Actually, no matter how well we design the algorithm, to find one scheduling solution, it must take more than one operation. That is to say, we cannot find one solution within 1 clock cycle. Therefore, in reality, we can never afford the run time of an exhaustive search algorithm. Exhaustive search algorithms work on small toy examples but not on problems of realistic sizes.

Fortunately, a lot of the solutions are apparently not Pareto-optimal solutions. If we have a method to either identify the optimal solutions or prune the non-optimal solutions, we can solve the problem much more efficiently than using the exhaustive search. In the following sections, we will explain step by step how we have designed the scheduling algorithm to solve our problems efficiently.

4.3 Forward Search Algorithm

4.3.1 The Kernel Heuristic

The basic function of a scheduler is to assign the nodes to processors and to order the nodes on each assigned processor. The problem can be approached from many angles. We take the following way.

The Framework of the Kernel Heuristic

The problem is decomposed into a certain number of decision steps. At each decision step, a node is assigned to a processor. Then the start time of executing this node on this processor is decided. It immediately follows that the number of nodes in a graph determines how many scheduling decision steps there will be.

In this scheduling policy, when the node has parent nodes that all finished execution, i.e., the node has its precedence constraint resolved, and when the processor, to which the node is assigned, finishes its job at hand, the processor will execute this node. Idle time on the processor is possible when the node has to wait for one or more of its parent nodes to finish execution. Then the processor may have already finished its previous job for some time. As a result, this scheduling policy will allow idle time on the processors only due to precedence constraints.

The Node Selection Policies

Under the scheduling framework is how to select a node and a processor at each step. It is possible that at one step, several nodes are precedence-resolved, and several processors are ready to accept a node.

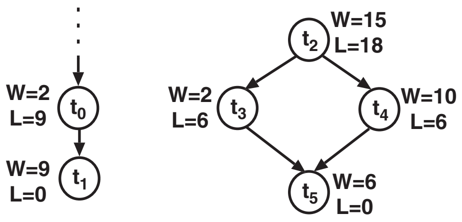

Fig. 4.3. Illustration of *weight* and *load* concept

Assume that one processor has been selected. The kernel heuristic then uses two criteria to select one node among several candidate nodes.

The first criterion is the *weight* of a node. *Weight* is defined as the execution time of the node on a reference processor. All the nodes should be calibrated on the reference processor. So the measurements for all the nodes are consistent. The larger the *weight* of a node, the higher priority it has to be assigned to the selected processor.

The second criterion is the *load* of a node. The *load* of a node is defined as the sum of the *weight*s of all its descendant nodes. The descendant node can be a child node, a grand-child node, ... Formally speaking, Node B is a descendant node of Node A if and only if there is an edge or a series of edges starting from Node A and ending at Node B. The larger *load* a node has, the higher priority it has to be assigned to the selected processor.

Figure 4.3 illustrates the concept of *weight* and *load*.

Based on these two measures, given the selected processor, the following policy takes care of selecting a node.

1. When a node is dominant in both *weight* and *load* over the other candidate nodes, it will be assigned to the selected processor;
2. When one node has the dominant *weight* and another node the dominant *load*, either of them can be assigned to the selected processor. By alternating their priorities, different solution points on the energy-cost vs. time-budget plane will be generated.

When node t_0 and t_2 in Fig. 4.4 are precedence-resolved, the scheduler will select t_2 to be assigned since it is dominant both in *weight* and *load*.

After t_2 is assigned, t_0, t_3, and t_4 are precedence-resolved, the scheduler will select t_0 and t_4 alternatively to be assigned (Fig. 4.5).

The kernel algorithm favors high-performance execution. It tries to schedule as tight as possible. Two small examples, an artificial one in Fig. 4.6 and a real-life one in Fig. 4.8, are used to show how optimal the kernel heuristic is. As explained in Section 4.2, we can only use small examples to avoid run time explosion of the exhaustive search.

All the artificial examples in this chapter are generated by Task Graph for Free (TGFF, Release 2.0) [70].

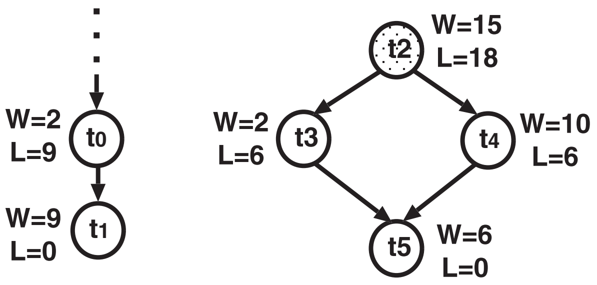

Fig. 4.4. Decision example of the kernel scheduling heuristic

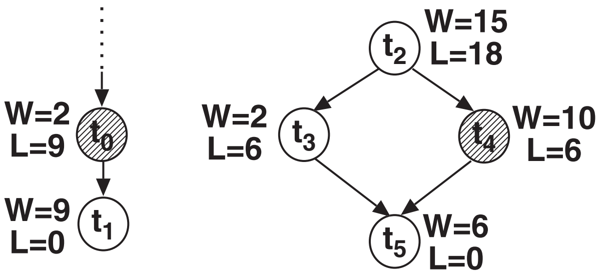

Fig. 4.5. Decision example of the kernel scheduling heuristic

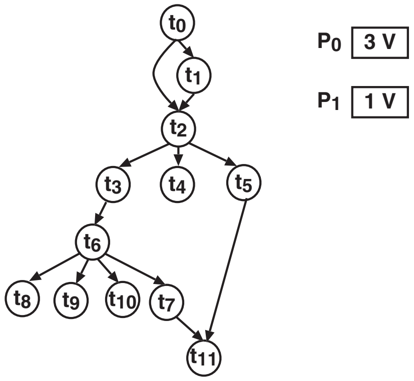

Fig. 4.6. A 12-node artificial graph with 2 processors

An Artificial Example

Figure 4.6 is an artificial graph with 2 processors. The TGFF input parameter file for this example are listed in Appendix A. The execution-time and energy-cost data of this example are listed in Table A.1. For this artificial example, The kernel heuristic gives 4 Pareto optimal solutions, while the exhaustive search gives 49 Pareto optimal solutions. The detailed solutions are listed in Tables A.2 and A.3. Figure 4.7 compares the solutions given by

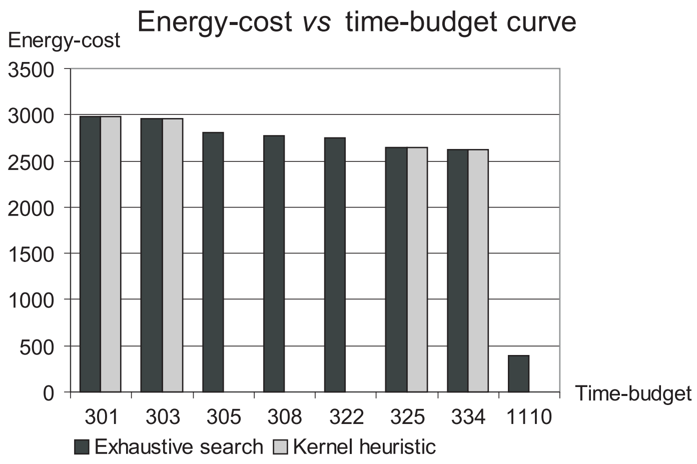

Fig. 4.7. Solution comparison for Fig. 4.6

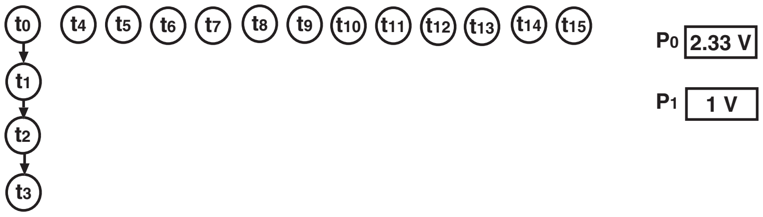

Fig. 4.8. A real-life graph with 2 processors

the exhaustive search and the kernel heuristic. This figure only shows the comparable solutions between the exhaustive search and the kernel heuristic. The exhaustive search produces 41 solutions between time budget 334 and 1110, which are not shown in Fig. 4.7. Since the kernel heuristic does not generate solutions in that range, there is no sense yet to make a comparison over that part. But if the solutions are of high optimality in the operational range, we can use it as the kernel and make extensions to improve the solution range. If the solutions are of poor quality, we should invent other algorithm as the corner stone. So we concentrate on the comparable part between the kernel heuristic and the exhaustive search.

Checking these four solutions produced by the heuristic against the solutions produced by the exhaustive search, we can verify that these four solutions are indeed optimal.

A Real-Life Example

Figure 4.8 is a real-life example. The graph is extracted from the MPEG-4 IM1 player. The IM1 player is a well-known prototype (executable specification)

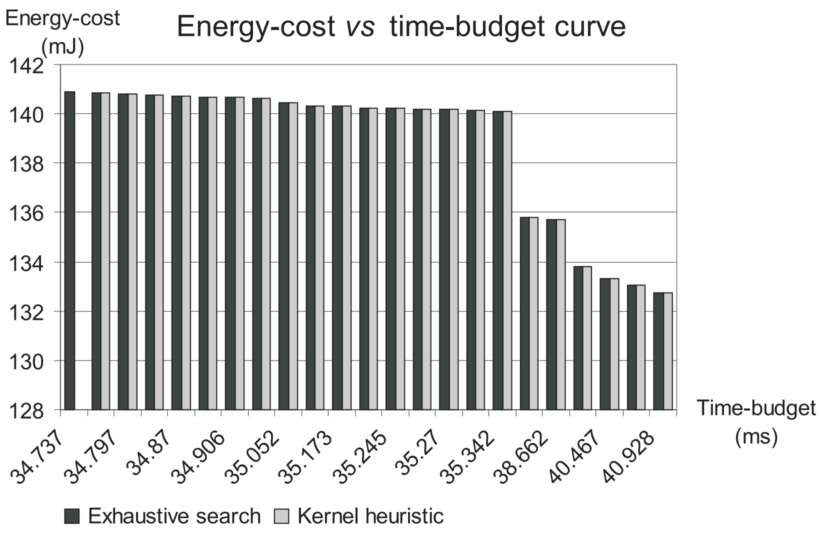

Fig. 4.9. Solution comparison for Fig. 4.8

of MPEG-4 standards in the MPEG community. IM1 means IMplementation 1. The graph is scheduled on a 2-processor platform.

For this real-life example, the kernel heuristic gives 22 Pareto optimal solutions, while the exhaustive search gives 3550 Pareto optimal solutions. The detailed solutions are listed in Table A.5. Figure 4.9 compares the solutions given by the kernel heuristic and the exhaustive search.

Checking the solutions from the kernel heuristic against the solutions from the exhaustive search, we show that these 22 solutions are indeed optimal. However, the exhaustive search again gives a much broader solution range while the kernel heuristic favors only the high-performance solutions.

4.3.2 Tuning the *Load* Calculation

In the kernel heuristic, the *load* of a node only reflects the total *weight* of all the descendant nodes of this node. It does not reflect how the descendant nodes are topologically connected. For example, nodes t_1 and t_5 in Fig. 4.10 both have three descendant nodes. Although they can have identical *load*, when doing scheduling, the importance of t_1 and t_5 can be different according to how their descendant nodes are connected. In other words, t_1 and t_5 can have very different impacts on the performance of the task graph execution.

We have also tried another way to calculate the *load*. That is, for a certain node, we first identify the longest path following this node. Longest means the maximum total *weight* of the nodes on this path. Then this maximum total *weight* is taken as the *load* of this node.

We have applied this *load* calculation on three examples, namely, a 29-node artificial graph on 4 processors (Fig. 4.11), a 41-node artificial graph on

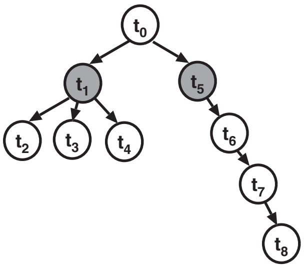

Fig. 4.10. *Load* difference

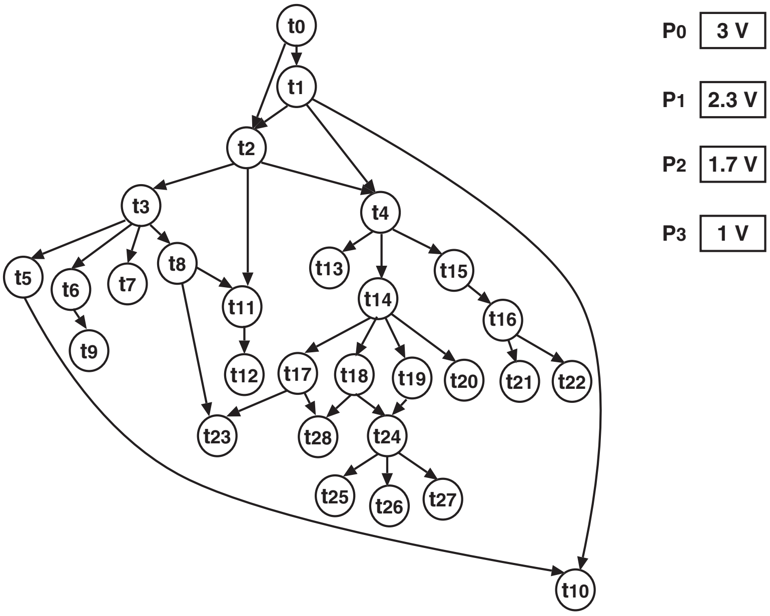

Fig. 4.11. A 29-node artificial graph with a 4-processor platform

4 processors (Fig. 4.12), and the same real-life graph as in Fig. 4.8 but on 4 processors (Fig. 4.13). The experiment results are discussed in Section 4.3.4.

4.3.3 Tuning the Processor Selection Priority

The kernel heuristic does not tell us how to select a processor. Actually, it does the selection implicitly.

The kernel heuristic first finds the earliest precedence-resolved nodes. Then those processors, which have finished their jobs by this earliest resolved time, make up the candidate processors. Among these candidate processors, the processor type with a smaller index has priority over the type with a larger index. Within the same processor type, the processor with the highest V_{dd} has the priority over the others.

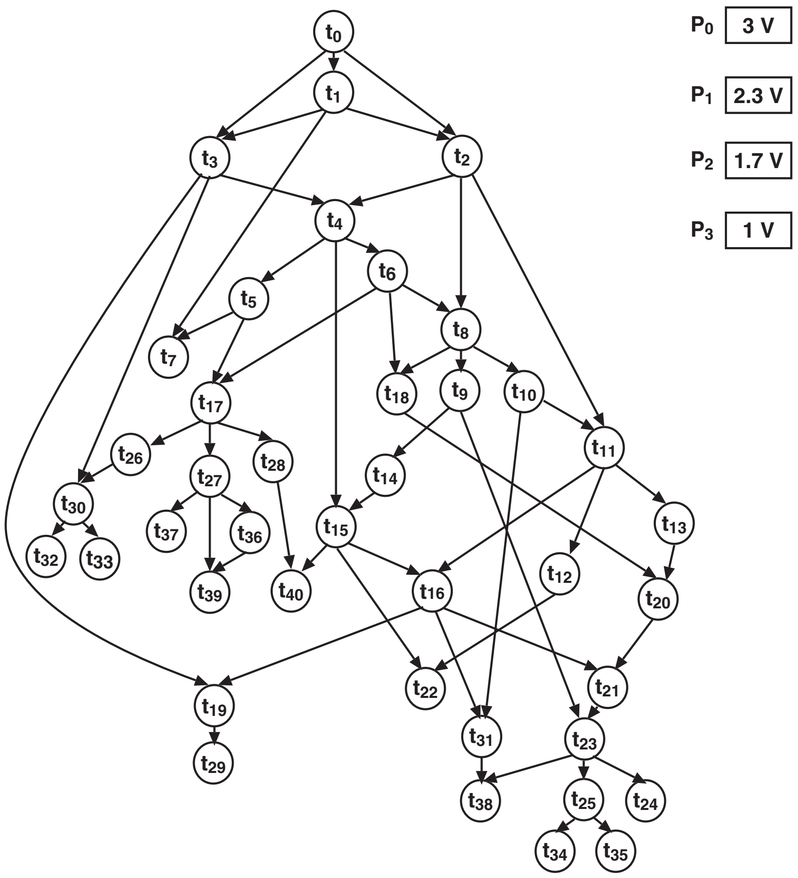

Fig. 4.12. A 41-node artificial graph with a 4-processor platform

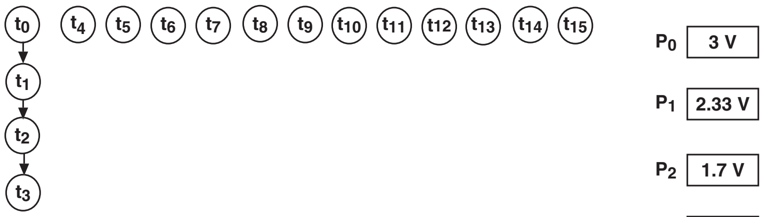

Fig. 4.13. A real-life graph with a 4-processor platform

This processor selection policy is not fair. It favors the high-execution speed processors. Or viewed differently, it cares more about the node execution speed than the energy consumption.

By reversing this processor selection policy, i.e., selecting the lowest V_{dd} processor when several processors are available simultaneously, the scheduler will favor the lower-energy consumption processors at the cost of node execution speed.

This tuned processor selection policy is applied to the same three examples as in Section 4.3.2. The experiment results are discussed in Section 4.3.4.

4.3.4 Tuning both *Load* Calculation and Processor Priority

Applying the *load* tuning (Section 4.3.2) and the processor-priority tuning (Section 4.3.3) simultaneously yields yet another scheduling algorithm.

Figures 4.14, 4.15, and 4.16 compare the solutions given by the kernel heuristic, the load-tuning algorithm, the processor-priority tuning algorithm and the load-and-processor-priority tuning algorithm on the three examples in 4.11, 4.12, and 4.13, respectively. The detailed solution report can be found in the appendix of Wong [245].

Figure 4.14 shows that compared with the kernel heuristic, tuning the *load* calculation

1. gives more optimal solutions at some time-budgets, but it also gives worse solutions at some other time-budgets;
2. gives three more solutions;
3. does not increase the solution range.

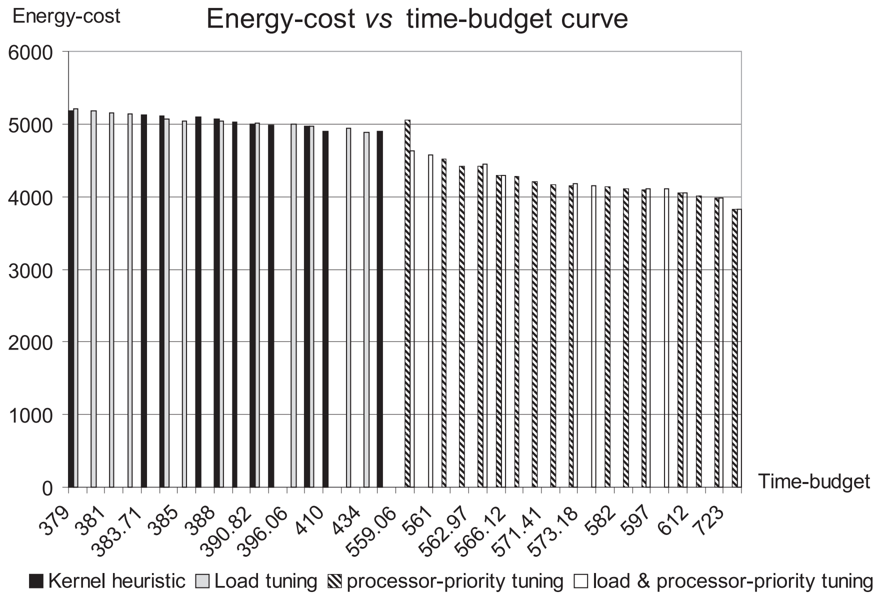

Fig. 4.14. Solution comparison for Fig. 4.11

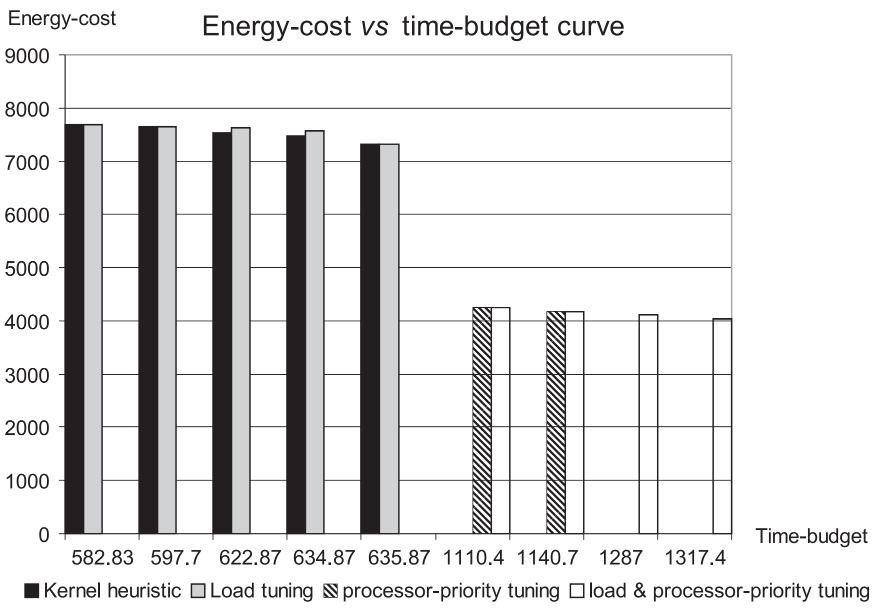

Fig. 4.15. Solution comparison for Fig. 4.12

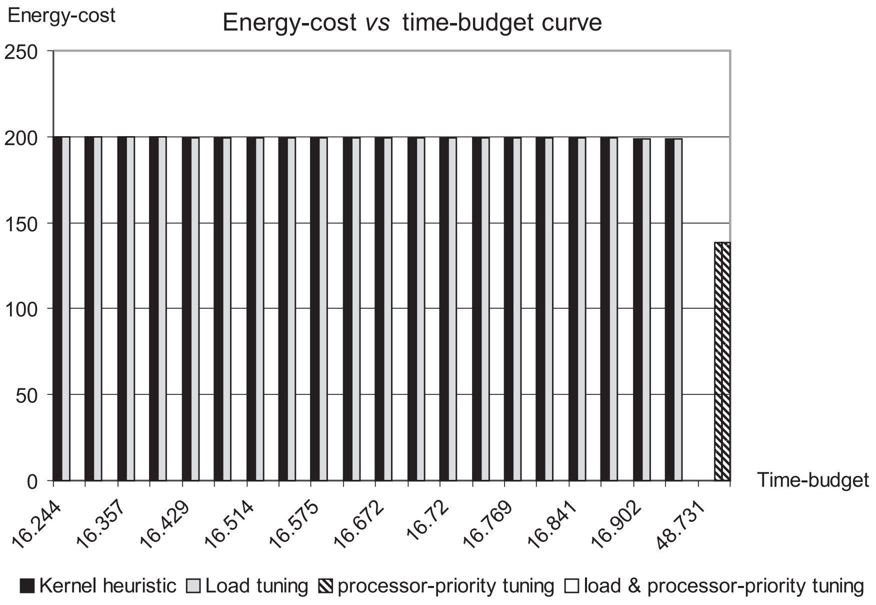

Fig. 4.16. Solution comparison for Fig. 4.13

Figures 4.15 and 4.16 show that tuning the *load* calculation has never beaten the kernel heuristic regarding the solution optimality at any time-budget. On the contrary, it only gives worse solutions at some time-budgets.

For the above reasons, we will keep the original *load* calculation in the further algorithm development.

The above three examples all show that reversing the processor selection priority gives slower and energy-cheaper solutions. These solutions are at least twice slower than those given by the kernel heuristic. That is to say, if we use a combination of the kernel heuristic and the processor-priority reversed algorithm, we can increase the solution range at least twice. However, the energy cost of these slower solutions does not drop as expected. According to Eq. (4.1) and Eq. (4.2), for a single processor, when the execution-time doubles, the energy-cost should drop to $1/4$ of the original value. Given that these examples all use a 4-processor platform, i.e., the scheduler can gain extra execution-time due to the parallel executing of the nodes, we still can expect more energy reduction than it currently gets. That is to say, the solution quality regarding the energy-cost could be improved. An obvious example is shown by Fig. 4.14. In this figure, at the time-budget of 561, the processor-priority tuned algorithm gives a solution with an energy-cost larger than the smaller time-budget solutions given by the kernel heuristic. Nevertheless, these three examples have shown us that carefully tuning the processor selection is a correct direction to favor the slow solutions and hence to increase the solution range. In later sections, we will follow this direction to further improve the scheduling algorithm.

Moreover, we can improve the scheduling algorithm at the following two places. The first one is about the big gap in the time-budget between the solutions produced by the kernel heuristic and by the processor-priority tuned algorithm. This gap means these two algorithms favor and concentrate on each of the two extremes respectively. Secondly, Fig. 4.16 clearly shows that reversing the processor selection priority only produces one slow solution. This is far from enough. We need to increase the solution count.

Finally, it is easy to observe that because tuning the *load* calculation does not have a pronounced improvement, combining it with the processor priority tuning does not gives us more benefit than tuning the processor priority alone.

4.3.5 Improving Node Selection Policy

The kernel heuristic, the load-tuned algorithm (Section 4.3.2), the processor-priority-tuned algorithm (Section 4.3.3), and the load-processor-priority-tuned algorithm (Section 4.3.4) all implicitly assume that there are always processors available to accept a node. This is a simplified picture of the scheduling problem. Actually, the following two cases accurately model the relation of node and processor in the timing aspect.

Case 1 At a decision step, one or more nodes are precedence-resolved while all the processors are busy. This situation happens when the graph has many nodes, which can be executed in parallel, while the processor number is relatively small. In other words, the available processors cannot accommodate all the precedence-resolved nodes. Then at a certain moment, one processor finishes its job or several processors finish their jobs simultaneously. Consequently, one or more nodes can become precedence-resolved. Therefore, the set of precedence-resolved nodes should be updated if necessary. Moreover, their priority regarding *weight* and *load* will be reshuffled. Now that precedence-resolved nodes and idle processors are present, the scheduler will make the node-to-processor assignment.

Case 2 At a decision step, one or more processors are waiting to accept a node. This situation happens when the platform has many processors while the graph is relatively sequential, i.e., due to precedence constraints, most of the nodes can only be executed one after another. Then at a certain moment, one processor finishes it job or several processors finish their jobs simultaneously. As a result, one node or more nodes become precedence-resolved. Moreover, the set of waiting processors should be updated.

For both of the above two cases, when the precedence-resolved nodes and the idle processors are collected, the scheduler will sort the precedence-resolved nodes according to their *weight* and *load*. If there is only one node with the largest *weight*, only one node with the largest *load*, and only one idle processor, this is the situation which the kernel heuristic expects.

However, more than one node can share the largest *weight* or the largest *load*, and more than one idle processor can exist. To show a simple example, all the *sink* nodes of a graph have 0 as their *load*. In Fig. 4.11, the graph has 13 *sink* nodes. It is quite possible that at a certain decision step, the scheduler has to make the assignment for some or all of those *sink* nodes.

To avoid missing any important scheduling decisions, the scheduler should try to assign each of these nodes to each of the idle processors alternatively. Apparently, for a real-size graph and platform, such a policy can lead to run-time explosion because many nodes and processors can be available for assignment. If the scheduler tries all those assignment possibilities one by one, the run time can be very huge. Therefore, we should improve the node selection policy while keeping the processor selection policy as it is in the kernel heuristic, i.e., the smaller index processor type and the higher V_{dd} processor within the same type have priority over the others.

In Section 4.3.2, we have seen that *load* as a measurement for the priority in scheduling is not enough, neither is the improved *load* calculation. So in addition to the node selection tuning discussed above, an extra node selection policy is introduced.

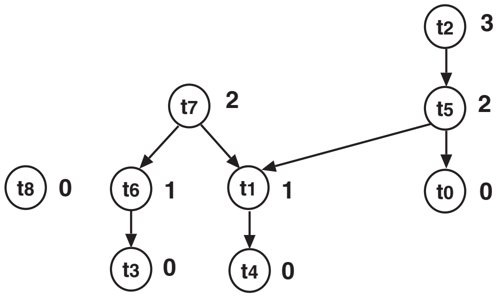

Fig. 4.17. Illustration of the *generation* concept

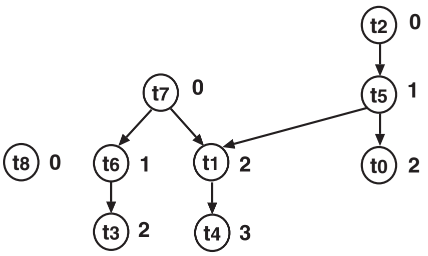

Fig. 4.18. Illustration of reverse *generation* counting

This new criteria is called *generation*. The *generation* of a sink node is 0. Starting from the sink nodes, the *generation* of the other nodes are calculated. For a node other than the sink node, *generation* is the maximum *generation* of all its child nodes plus 1. It is illustrated as in Fig. 4.17. The *generation* of each node is annotated on the right side of each node.

The reason that we calculate *generation* from the sink node is that the sink node does not have any child node, so it can impact the scheduling only from its own execution time. Therefore it is less important in the scheduling decisions than the other nodes. In addition, if we count *generation* from the source node, shown in Fig. 4.18, though node t_8 has much less impact on the other nodes than node t_2 and t_7, it has the same *generation* value as node t_2 and t_7. From another view point, though node t_8, t_3, t_4, and t_0 are all sink nodes, they have different *generation* values. Since our intention is to use *generation* to capture the impact of a node on the other nodes, forward counting is not in line with this intention.

The new node selection policy is that in addition to the nodes with maximum *weight* and the maximum *load*, the nodes with maximum *generation* will also be given the assignment priority. That is, the node with one of these three maximum measurements will be assigned to the selected processor.

Such a scheduling policy has been implemented and applied to the small artificial example of Fig. 4.6. Figure 4.19 graphically compares the solu-

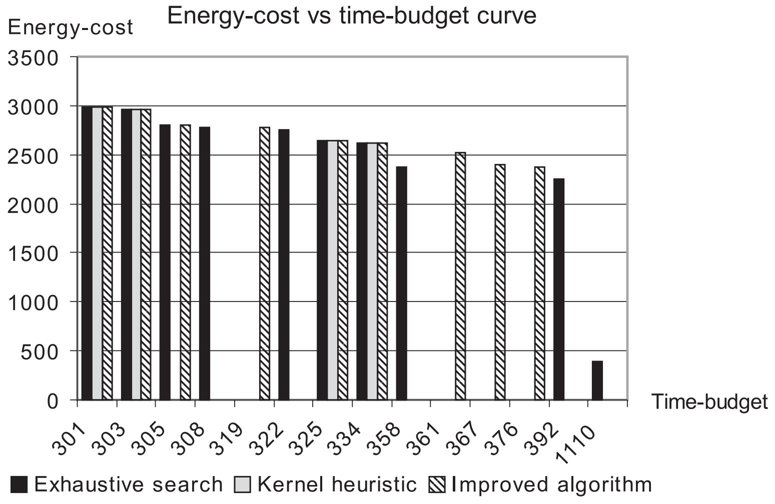

Fig. 4.19. Solution comparison of the exhaustive search, kernel and improved heuristic for Fig. 4.6

tions produced by the exhaustive search, kernel heuristic, and the selection-improved algorithm.

From the comparison, we can conclude that

1. The improved algorithm produces more solutions than the kernel heuristic;
2. However, the solution range does not increase significantly;
3. The improved algorithm produces optimal solutions at the low time-budget end;
4. At the high time-budget part, the improved algorithm still needs to improve its solution quality regarding the energy-cost; for example, at time-budget 367 the improved algorithm gives an energy-cost greater than the exhaustive search does at time-budget 358.

The reason for the last observation is easy to understand. Because to avoid run-time explosion, the improved algorithm still uses the old processor selection policy, which favors the high V_{dd} hence high-speed and high-energy processors, then it favors the low time-budget solutions.

While trying to improve the node selection and processor selection, we come across the problem of too many candidate nodes. This means the search space explosion.

4.3.6 Comparing ASAP–ACAP and ASAP

In Section 4.3.5, we have come across the following dilemma. On the one hand, we need to increase the processor selection possibilities, i.e., the scheduler should not only favor the high V_{dd} processors. On the other hand, at a decision step, normally we already have more than one node to be assigned. If the number of candidate processors is increased, and the scheduler tries every possibilities of the node-to-processor assignment, the search space will explode.

This is one side of the problem. Another side is the processor selection policy. Currently, the scheduler selects one processor for all the candidate nodes to be assigned. On a heterogeneous platform, different nodes can favor different processors for execution speed or energy cost. So it is better to customize the processor selection for each candidate node. To avoid the search space explosion problem, we can start with the following processor selection policy.

For a selected node, the scheduler tries the following two sets of processors one by one.

1. Select fastest processors, then select the energy-cheapest processors among these fastest processors;
2. Select the energy-cheapest processors, then select the fastest processors among these energy-cheapest processors.

We call the first set of processors As Soon As Possible (ASAP) processors while we call the second set of processors As Cheap As Possible (ACAP) processors.

This node selection policy favors the two extremes. One extreme is the high-performance, the other extreme is the cheap energy-cost.

To check how much gain we can get through such a processor selection policy, we also implemented a simplified version, which only calculates ASAP processors for a selected node.

Applying these two algorithms on the example of Fig. 4.6, we get the following results shown in Fig. 4.20. The detailed solutions are listed in the appendix of [245]. Following conclusions can be drawn from this figure.

1. The ASAP–ACAP algorithm produces almost four times more solutions than the ASAP algorithm does.
2. The ASAP algorithm concentrates on the high-performance part. This is as expected since it selects the ASAP processors only.
3. At the high-performance part, ASAP–ACAP and ASAP algorithms have almost identical solutions. Actually, they are only different at time-budget 359 and 361.
4. Compared with the exhaustive search (solutions listed in Table A.3), before time-budget 359, both ASAP–ACAP and ASAP produce optimal solutions as the exhaustive search.
5. The ASAP–ACAP algorithm covers 40% of the solution range of the exhaustive search.

On the same Unix machine, the ASAP–ACAP algorithm takes 0.45 s while the ASAP algorithm takes 0.04 s. That is because the ASAP–ACAP algorithm

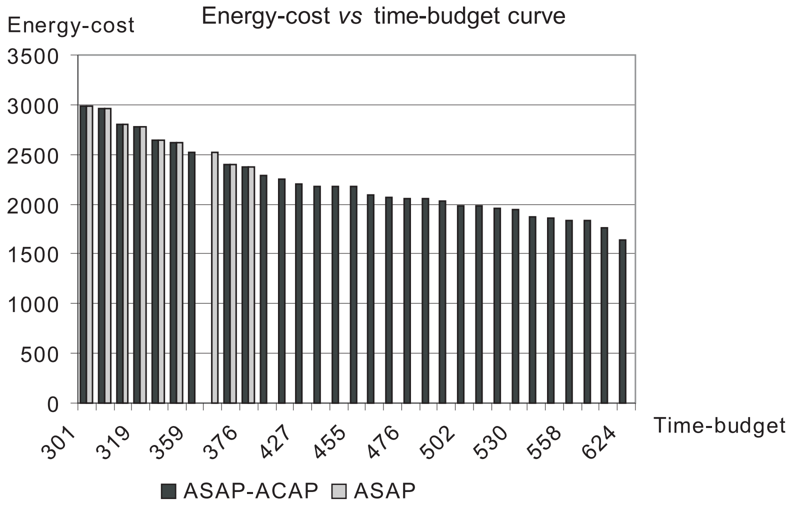

Fig. 4.20. Solution comparison of the ASAP–ACAP and ASAP for Fig. 4.6

does more calculation to find the processors for a selected node and hence it explores a larger search space.

For the design-time scheduling, fast exploration is an essential requirement. Most of the search space exploration should be covered at this stage. It simply because we cannot afford too much time wasted at runtime and hence no time can be spent on finding extra intermediate solutions then. That means for a reasonable size task set (represented by the graph) and a reasonable size platform, the run time of the design-time scheduler normally should not go beyond one night. Otherwise, other design steps in this TCM design flow will be blocked by this design-time scheduling phase.

Unfortunately, applying the ASAP–ACAP algorithm on the example of Fig. 4.11, the scheduler cannot produce final results within 1 h.

In conclusion, the ASAP–ACAP algorithm produces high enough solution quality. However, it still needs to improve the solution range and the algorithm run time.

This is again a dilemma. If we increase the search space to expand the solution range, we will pay more algorithm run time. So we need to tune either the node selection policy or the processor selection policy further to target more accurately at the optimal solutions and hence save the time spent on the non-optimal solutions.

4.3.7 Pruning Techniques for Tie-Breaking

By examining both artificial graphs and graphs extracted from real-life applications, we have noticed that the node with a higher *generation* normally also has a larger *load*. There are few exceptions. However, if we target the typical cases, we can tolerate those few exceptions.

Compared with *generation*, *weight* indicates more of the node itself than the impact it has on the descendant nodes. Though *weight* can have a big impact on the system total execution time, we can treat this feature by exploring more processor assignments, which will be addressed in Section 4.3.9.

So we introduce a pruning technique by first selecting the maximum *generation* nodes. If more than one maximum *generation* node is present, we further prune by selecting the maximum *load* nodes from these maximum *generation* nodes. If still more than one node remain, we further prune by selecting the maximum *weight* nodes. Normally, if the graph is not created by intention to fight against such a selection policy, only one node will be left after these three pruning steps. However, if there are still more than one node left after the three pruning steps, the scheduler will pick up the nodes one by one to make the assignment.

In Section 4.3.6, we have seen that the ASAP algorithm will give high optimality solutions in the high-performance part. That is to say, the solution quality is guaranteed though it produces solutions of a rather small range. To avoid the algorithm run time explosion, the ASAP processor selection policy is used.

Since we prune the nodes in the assignment exploration, and we use the ASAP processor selection policy, the search space should decrease and hence the solution number should decrease.

How to increase the solution range will be treated by Section 4.3.9. This section cares more about finding the solutions of high optimality in a short algorithm run time.

Experiments are done on three examples to show the optimality and algorithm run time. The first example is Fig. 4.6. The scheduler produces one solution for this example shown in Table 4.1. Comparing with the exhaustive search (Table A.3), we know this is the optimal solution.

Table 4.2 clearly shows that the algorithm run time is indeed significantly reduced.

Table 4.1. Scheduling solution after introducing pruning techniques for Fig. 4.6

Time-budget	Energy-cost
301	2985.67

Table 4.2. Run time comparison of the exhaustive search, kernel heuristic, and algorithm with pruning techniques for Fig. 4.6

	Exhaustive search	Kernel heuristic	Algorithm with pruning
Run time (s)	1.13	0.03	0.01

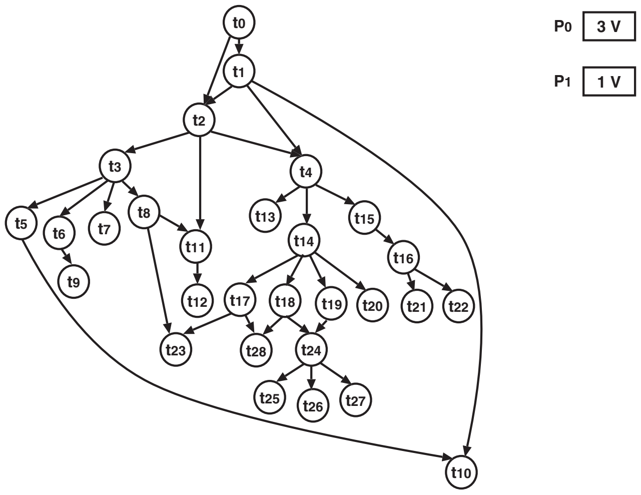

Fig. 4.21. A 29-node artificial graph with a 2-processor platform

Table 4.3. Scheduling solution after introducing pruning techniques for Fig. 4.21

Time-budget	Energy-cost
680	6182.33

The second example is Fig. 4.21. The graph is the same 29-node graph as in Fig. 4.11, but the platform is a 2-processor platform. The scheduler produces one solution, as shown in Table 4.3 for this example. However, the exhaustive search will not work on this example, so we compare this solution with the solutions produced by the kernel heuristic (see details in the appendix of [245]). Figure 4.22 shows this comparison. This only solution at time budget 680 is 3.65% larger than the fastest solution at time-budget 656. The energy-cost is 3.48% higher than the solution at time-budget 689 produced by the kernel heuristic. While the kernel heuristic takes 129.40 s, the algorithm with pruning only takes 4.24 s.

Table 4.4 again shows that with the pruning technique, the algorithm run time is significantly reduced.

The third example is Fig. 4.12. The scheduler finds one solution for this example shown in Table 4.5. Again, we use the kernel heuristic for comparison. The solutions found by the kernel heuristic (listed in the appendix of [245]). Figure 4.23 shows this comparison. The only solution at time budget 635.87, is is 9.1% larger than the fastest solution at time-budget 582.826. The energy-cost is 5.67% higher than the solution at the same time-budget produced by the kernel heuristic.

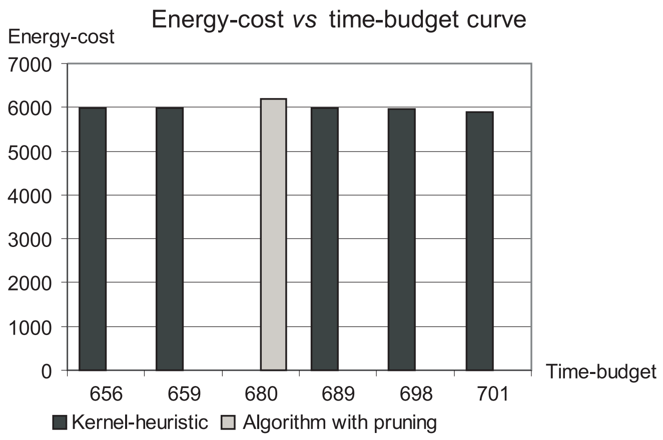

Fig. 4.22. Solution comparison of the kernel heuristic and the algorithm with pruning for Fig. 4.21

Table 4.4. Run time comparison of the kernel heuristic and algorithm with pruning techniques for Fig. 4.21

	Kernel heuristic	Algorithm with pruning
Run time (s)	129.40	4.24

Table 4.5. Scheduling solution after introducing pruning techniques for Fig. 4.12

Time-budget	Energy-cost
635.87	7734.17

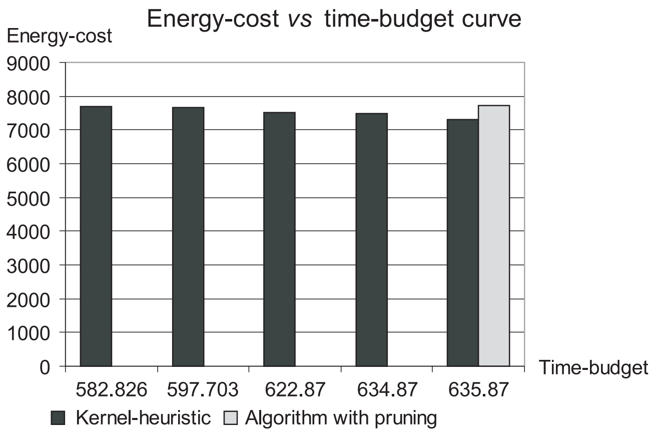

Fig. 4.23. Solution comparison of the kernel heuristic and the algorithm with pruning for Fig. 4.12

Table 4.6. Run time comparison of the kernel heuristic and algorithm with pruning techniques for Fig. 4.12

	Kernel heuristic	Algorithm with pruning
Run time (s)	1.02	0.02

Table 4.6 once again shows that with the pruning technique, the algorithm run time is significantly reduced.

From these three examples, we make the following conclusions

1. With pruning techniques, the algorithm run time is significantly reduced. For large examples, like Figs. 4.21 and 4.12, a reduction factor of at least 30 can be expected.
2. With pruning techniques, the solution number will be restricted. However, the solutions are close to the fastest solutions found by the exhaustive search or the kernel heuristic. They are always less than 10% away from the fastest solutions.
3. The solutions found with pruning techniques are of high optimality. That is, compared with solutions found by the exhaustive search or the kernel heuristic, the energy cost is always less than 6% higher.

4.3.8 Further Pruning Technique for Tie-Breaking

In Section 4.3.7, we have mentioned that if the graph is not created by intention to fight against the pruning techniques, normally only one node should be left after the *generation*, *load*, and *weight* pruning steps. Therefore though the scheduler takes all the nodes after the three pruning steps for assignment, we do not expect the search space explosion with such a node selection policy. But since the chance of more than 1 node left after all the three pruning steps is very low, the scheduler can pick up just one node. The scheduler should target the normal case, not the very rarely happening special case.

However, when extracting graphs from real-life applications, we often end up with graphs like Fig. 4.24. This graph is extracted from the scalable mesh decoding algorithm of MPEG-21 [77, 78, 95, 96, 139]. It has four parallel execution paths. These four parallel execution paths are identical. Actually, $Path_2$, $Path_3$, and $Path_4$ are just duplicate of $Path_1$. They are used to process the parallel incoming data packets.

For such kind of graphs, it is obvious that due to the symmetry of those duplicated paths, the scheduler only needs to explore the assignment possibilities for one path. In other words, the search space is symmetrically divided into several pieces, so the scheduler only needs to search one piece of those search space. One way to implement this search space reduction policy is to select 1 node after the three pruning steps.

We have done the experiments on the same three examples of Section 4.3.7 with this further pruned node selection policy. This further

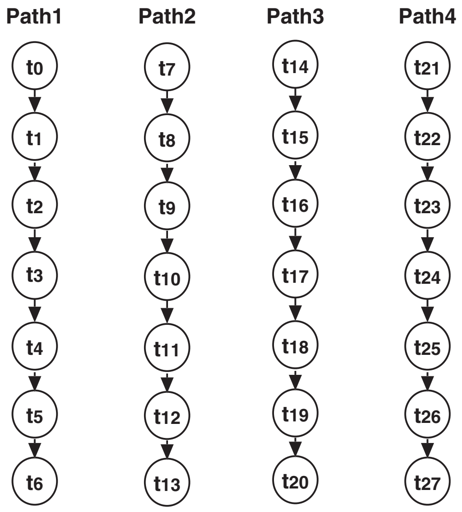

Fig. 4.24. An example graph of path duplication

Table 4.7. Run time comparison between the pruning scheduler and the further pruning scheduler

Example1: Fig. 4.6
Example2: Fig. 4.21
Example3: Fig. 4.12

	Run time (s)		
	Example1	Example2	Example3
Pruning	0.01	4.24	0.07
Further pruning	0.01	4.16	0.03

pruned node selection policy produces exactly the same solutions as the scheduler in Section 4.3.7. The conclusion is that we do not loose solution quality when further pruning the node selection space.

Table 4.7 compares the algorithm run time between the pruning technique and the further-pruning technique. The resolution of the timing measurement is not very small. However, we can see that the further-pruned technique runs faster on these three examples.

In addition to these three examples, we also ran the original pruning technique of Section 4.3.7 on the real-life example in Fig. 4.24 with 4 processors. We did not get final results within 1 h. Apparently, we have run into the search space explosion. Using the further pruned technique, we can get results within 1 h. Those experiments with the further pruned techniques will be explained in the following sections. And the following algorithm extensions are based on the further-pruned technique.

4.3.9 Extension with Exhaustive Search

Though the algorithm with pruning techniques in Section 4.3.7 is very fast and the solutions are of high quality, it does not produce a wide solution range. If we use this algorithm as the kernel and make extensions to increase the solution range, we can keep the solution quality.

The main reason for the small solution range is the processor selection policy. The scheduler selects only the ASAP processors for each node. Normally, the number of the ASAP processors will be very limited. However, if the scheduler uses also the ACAP processors for each node, and when the graph has a large number of nodes, the search space goes up exponentially. Assume, on average, each node has one ASAP processor and one ACAP processor. The scheduler will then try to assign one node to each of these two processors. If a graph has 50 nodes, the scheduler will search all the 2^{50} assignment possibilities. Assume, we have a machine of 1 GHz clock frequency, and it takes this machine one cycle to explore one assignment possibility. To explore all these 2^{50} assignment possibilities, it will take 13.03 days.

In addition, remember the example shown in Fig. 4.20, the ASAP–ACAP algorithm produces a solution range of 40% of the exhaustive search. That is to say, trying to assign the nodes to their ASAP–ACAP processors is perhaps not enough for producing a large solution range. To get a larger solution range, the scheduler should explore more processor assignment possibilities.

We can draw the following conclusions from this example.

1. Increasing the number of processor assignment possibilities of the node is effective in expanding the solution range.
2. Applying the same number of processor assignment possibilities to every node is not computationally affordable.

Since not all the nodes have the same importance in scheduling, if we try the important nodes with more exhaustively processor assignment possibilities, we probably will get a larger solution range. Moreover, since the percentage of the important nodes of a graph will remain stable when the graph size goes bigger, the assignment possibilities will not explode with the graph size. For small graphs, perhaps every node is an important node, i.e., the percentage of the important nodes is very high. However, we can afford exhaustive search for small graphs. So small graphs will not make difficulties in such a scheduling policy.

The previous experiments have shown that the node *generation* is an important parameter in scheduling. In addition, *weight* is also important. A node with a very large *weight* has a big impact on the finish-time of the graph execution. So the scheduler will be extended with an exhaustive search on the nodes with top values of *generation* and *weight*.

Figure 4.25 illustrates the exhaustive assignment search on top *generation* values. In this figure, if we set the exhaustive search depth for the top *generation* values to be 2, node t_0 and t_1 will be exhaustively searched in

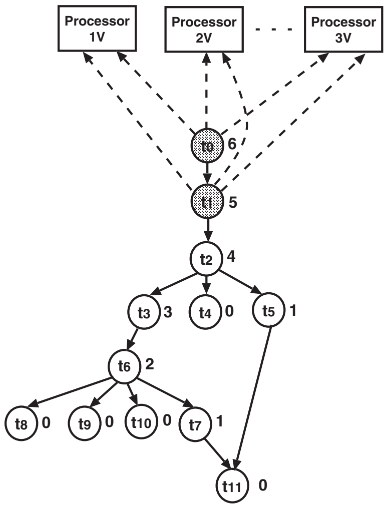

Fig. 4.25. Exhaustive search on top 2 *generation* values

their processor assignment possibilities. In general, if the user sets the search depth parameter to be n, the exhaustive search will be applied to the nodes with the top n *generation* nodes.

Figure 4.26 illustrates the exhaustive assignment search on top *weight* values. In this figure, if we set the exhaustive search depth for the top *weight* values to be 1, and if node t_5 is the heaviest node of this graph, its processor assignment possibilities will be exhaustively searched. In general, if the user sets the search depth parameter to be n, the exhaustive search will be applied to the top n *weight* nodes. However, here the situation is more complicated than in the case of *generation*. If we sort the nodes by their *weight*, and we assign a rank number to each node according to their *weight*, it is possible that we have less nodes than required by the user, namely, n, at rank m, while we have more nodes than n at rank $m + 1$. In such a situation, the scheduler will only pick up part of the nodes of rank $m + 1$ to meet the required node number n.

It is worth noting that the search depths for *generation* and for *weight* are two separate parameters of the scheduler. So we can explore the impact of these two exhaustive search scenarios separately. In addition, these two parameters work differently. The search depth for *generation* dictates that every node with a *generation* larger or equal to this parameter will be exhaustively searched. So the user does not have the direct control of how many nodes will be exhaustively searched. In contrast, the search depth for *weight* does dictate the number of nodes to be exhaustively searched. It is the scheduler which takes care of finding the nodes starting from the maximum *weight* nodes, second largest *weight* nodes, . . . , until the required number of nodes is reached.

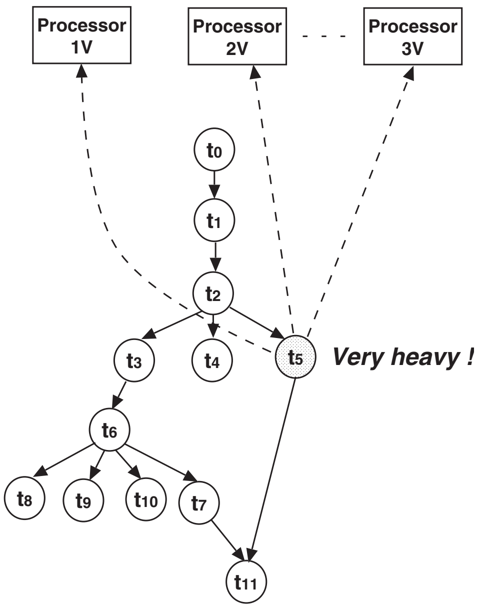

Fig. 4.26. Exhaustive search on top 1 *weight* value

We will use two examples to thoroughly study the effects of these two parameters and the combination of these two parameters. The first example is the artificial example in Fig. 4.21, and the second example is the real-life example in Fig. 4.35.

Experiment with Fig. 4.21

We first study the effect of the *generation* parameter. We have switched off the exhaustive search for *load* (i.e., set the *load* search depth to 0), and applied the scheduler to Fig. 4.21 with the *generation* search depth of 4, 5, 6, 7, respectively. Figure 4.27 compares the scheduling solutions for these four search depths. The detailed solution report is listed in the appendix of Wong [245]. Figure 4.28 compares the scheduling solutions of *generation* 7 and the kernel heuristic. Figure 4.29 and Table 4.8 compare the run time at different *generation* values and the run time of the kernel heuristic.

The solution comparison provides us with the following conclusions.

1. Increasing the search depth for *generation* improves both the solution quality and the solution range.
2. There can be one or more threshold *generation* values, at which either the solution quality or the solution range can have a significant improvement. In this example, *generation* 6 has a big improvement in solution

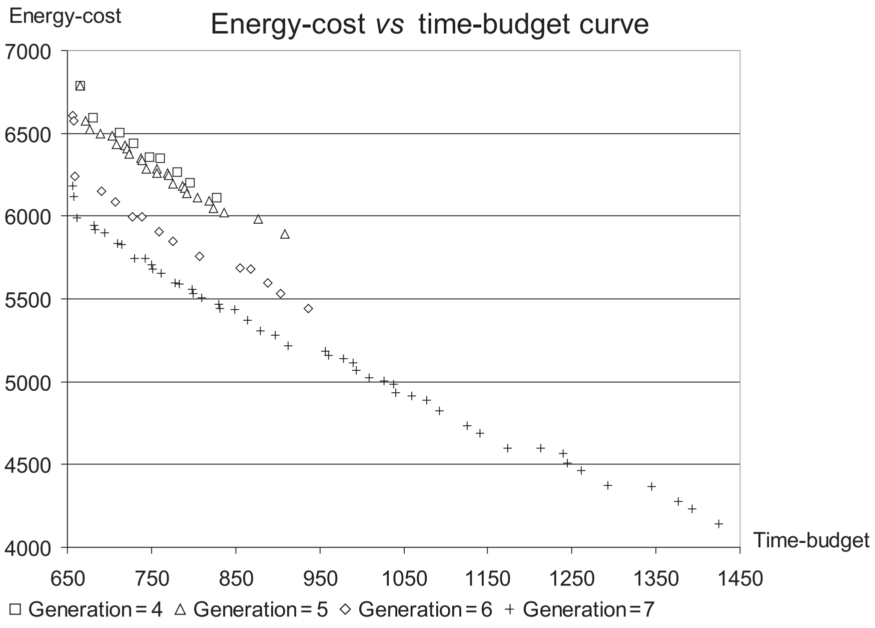

Fig. 4.27. Solution comparison at different *generation* values for Fig. 4.21

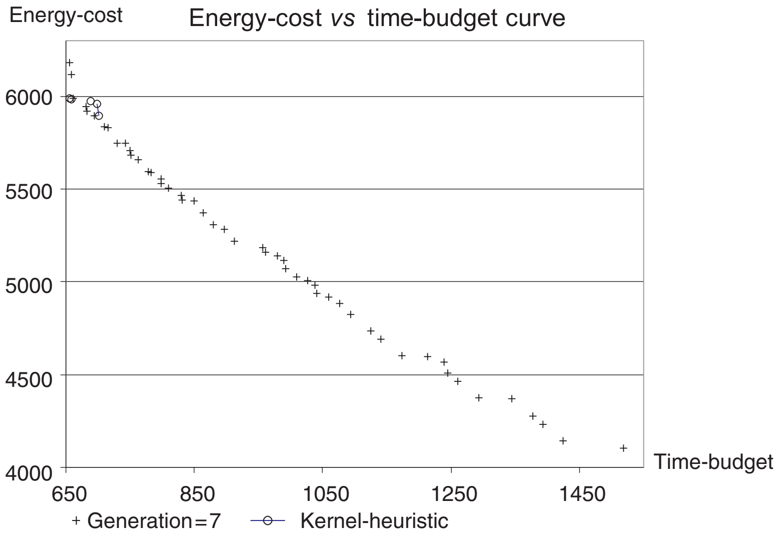

Fig. 4.28. Solution comparison at *generation* 7 and the kernel heuristic for Fig. 4.21

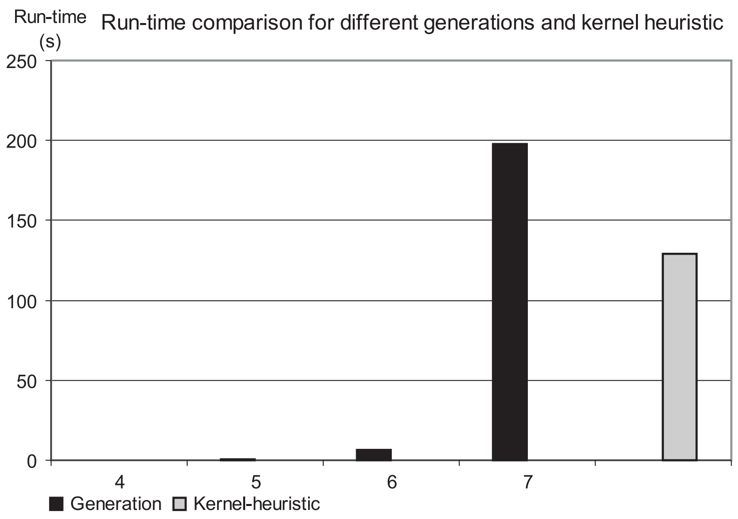

Fig. 4.29. Run time comparison at different *generation* values for Fig. 4.21

Table 4.8. Run time comparison at different *generation* values and the kernel heuristic for Fig. 4.21

G *Generation*

	G=4	G=5	G=6	G=7	Kernel heuristic
Run time (s)	0.2	0.63	6.4	197.53	129.40

quality while *generation* 7 has a big improvement both in solution quality and solution range.

3. Increasing the search depth for *generation* does not necessarily increase the solution count. In this example, *generation* 6 has 15 solutions while *generation* 5 has 26 solutions. But this is not a problem as long as the solution count remains reasonable because the solution range is much more important than the solution count. At run time, the solution count, which can be incorporated in the Pareto point selection, is anyway not higher than about 10.
4. At *generation* 7, the scheduler produces solutions very close to those of the kernel heuristic at the high-performance part while it has a much larger solution range than the kernel heuristic.

We can further conclude that though many pruning techniques are introduced to reduce the search space, this will not reduce the solution quality if the *generation* parameter is set properly.

The algorithm run time comparison shows us that the run time penalty can be quite large due to increasing the *generation*. In addition, at a certain *generation*, e.g., 7 in this example, the algorithm run time can have a jump.

Finally, at *generation* 7, though the solution quality is very close to the kernel heuristic and the solution range is very large, the algorithm run time is comparable to the kernel heuristic.

Next, we study the effect of the *weight* parameter. Figure 4.30 compares the scheduling solutions at *weight* values of 5, 10, 15, and 20, respectively. The detailed solution report is listed in the appendix of Wong [245]. Figure 4.31 compares the scheduling solutions of *weight* 20 and the kernel heuristic. Figure 4.32 and Table 4.9 compare the run time at different *weight* values and the run time of the kernel heuristic.

From the solution comparison, we can make similar conclusions as with the *generation* parameter.

1. Increasing the search depth for *weight* improves both the solution quality and the solution range.
2. At *weight* 20, the scheduler produces the solutions even better than those of the kernel heuristic at the high-performance part while it has a much larger solution range than the kernel heuristic.

Comparing the solutions produced at different *generation* values and the solutions at different *weight* values, we can conclude that *generation* and *weight* can have different effects on the solution quality. For this example (Fig. 4.21), the solutions at *weight* 20 are best both in the quality and in the solution range though its run time is 3.89 times of the run time at *generation* 7.

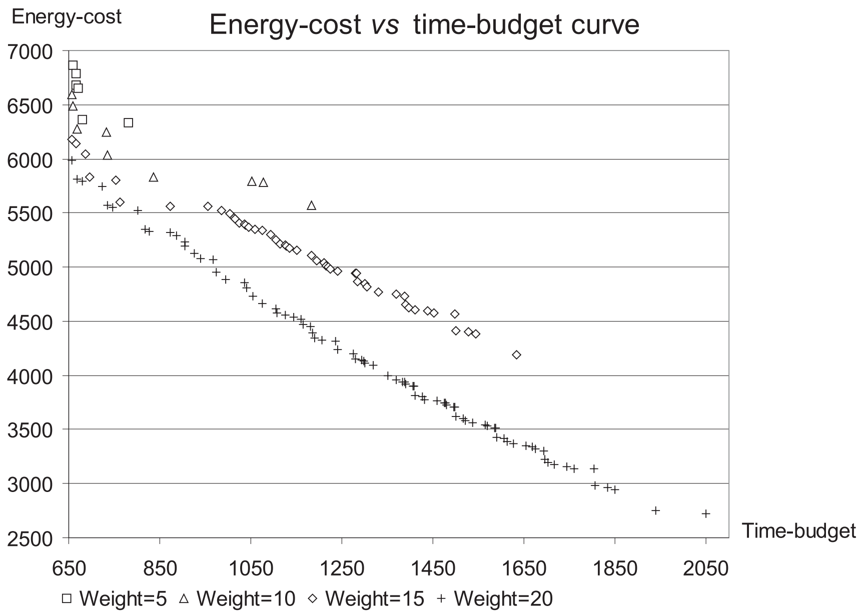

Fig. 4.30. Solution comparison at different *weight* values for Fig. 4.21

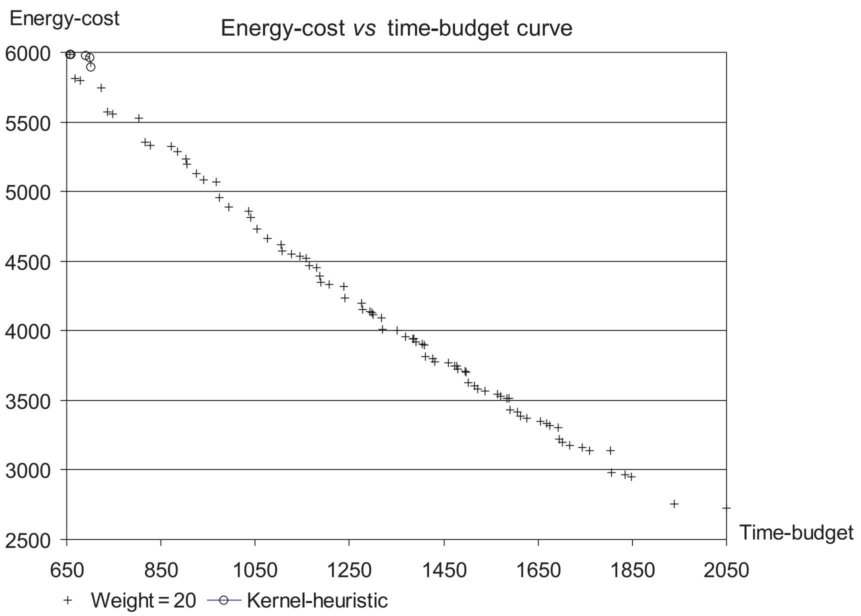

Fig. 4.31. Solution comparison at *weight* 20 and the kernel heuristic for Fig. 4.21

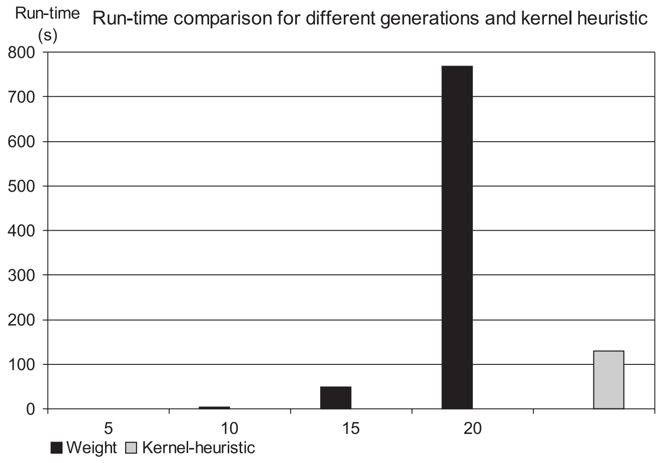

Fig. 4.32. Run time comparison at different *weight* values and the kernel heuristic for Fig. 4.21

Comparing the algorithm run time, we can conclude that the algorithm run time increases fast with the *weight* values. However, even at *weight* 20, where the solution quality becomes better than the kernel heuristic and where the slowest solution is more than 3 times the fastest solution, the algorithm run time is acceptable.

Table 4.9. Run time comparison at different *weight* values and the kernel heuristic for Fig. 4.21

W *Weight*

	W=5	W=10	W=15	W=20	Kernel heuristic
Run time (s)	0.27	3.89	49.47	768.9	129.40

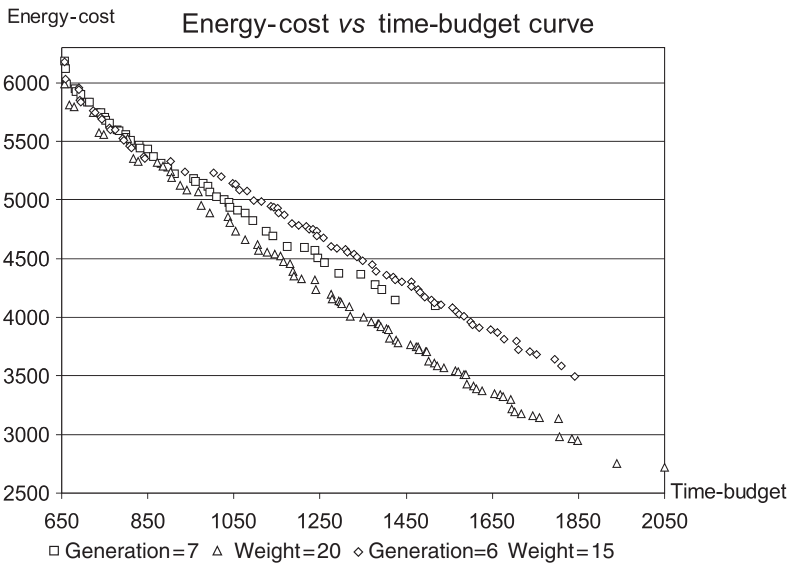

Fig. 4.33. Solution comparison at *generation* 7, *weight* 20, and the combination of *generation* 6 and *weight* 15 for Fig. 4.21

Finally, we study the effect of combining exhaustive search for *generation* and *weight* simultaneously. Figure 4.33 compares the solutions at *generation* 7, *weight* 20, and the combination of *generation* 6 and *weight* 15. The detailed solutions at this *generation* and *weight* combination are listed in the appendix of Wong [245]. Figure 4.34 and Table 4.10 compare the run time at *generation* 7, *weight* 20, *generation* 6 and *weight* 15, and the run time of the kernel heuristic.

The solution comparison clearly shows that the algorithm at *weight* 20 is the best both in solution quality and solution range among these three algorithm configurations. However, at the high-performance part, these three algorithms have similar solution quality.

Since the node selection policies by *generation* and by *weight* are different, i.e., they will select different nodes for exhaustive search, they have different impact on the scheduling solution quality. In this example, though the algorithm run time configured at *weight* 20 and the algorithm run time configured *generation* 6 and *weight* 15 are very close, the solution quality at *weight* 20 is much higher than at *generation* 6 and *weight* 15. In other words, when we are

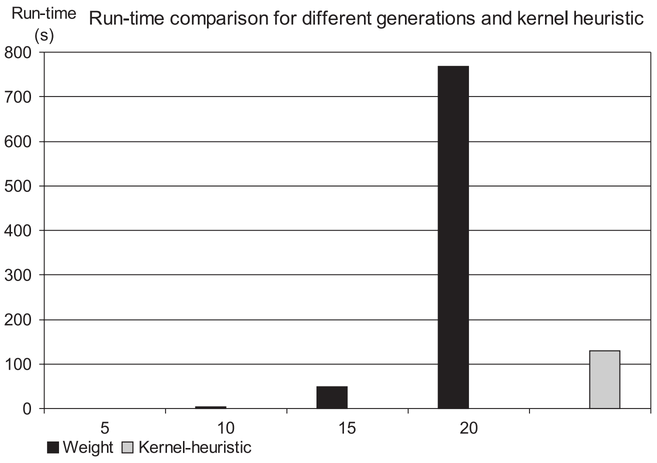

Fig. 4.34. Run time comparison at *generation* 7, *weight* 20, *generation* 6 and *weight* 15, and the kernel heuristic for Fig. 4.21

Table 4.10. Run time comparison at *generation* 7, *weight* 20, *generation* 6 and *weight* 15, and the kernel heuristic for Fig. 4.21

W *Weight*
G *Generation*

	G=7	W=20	G=6 & W=15	Kernel heuristic
Run time (s)	197.53	768.9	739.2	129.40

not satisfied with the solution quality at *weight* 15, and we want to improve the solution quality, we should increase the search depth for *weight* instead of the search depth for *generation*. By increasing the search depth for *weight*, we get higher solution quality with almost the same algorithm run time.

The same argument holds for the *generation* search depth. That is, at *generation* 6, if we need to improve the solution quality, we should increase the search depth for *generation* instead of the search depth for *weight*. Then we get higher solution quality with an even smaller algorithm run time.

So we can conclude that given the same amount of algorithm run time, the combination of exhaustive search for *generation* and *weight* is not necessarily better than the single "incremental" search for either *generation* or *weight*.

Experiment with Fig. 4.35

Having done the first experiment on an artificial example, we are now going to repeat the experiment with a real-life example (Fig. 4.35). This graph is extracted from the scalable mesh decoding algorithm of MPEG-21. The

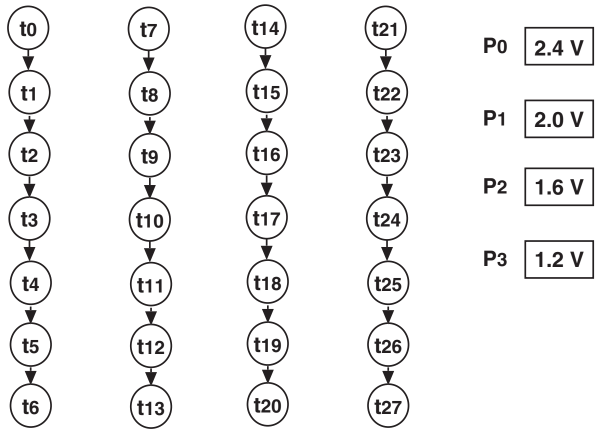

Fig. 4.35. A real-life graph of MPEG-21 with a 4-processor platform

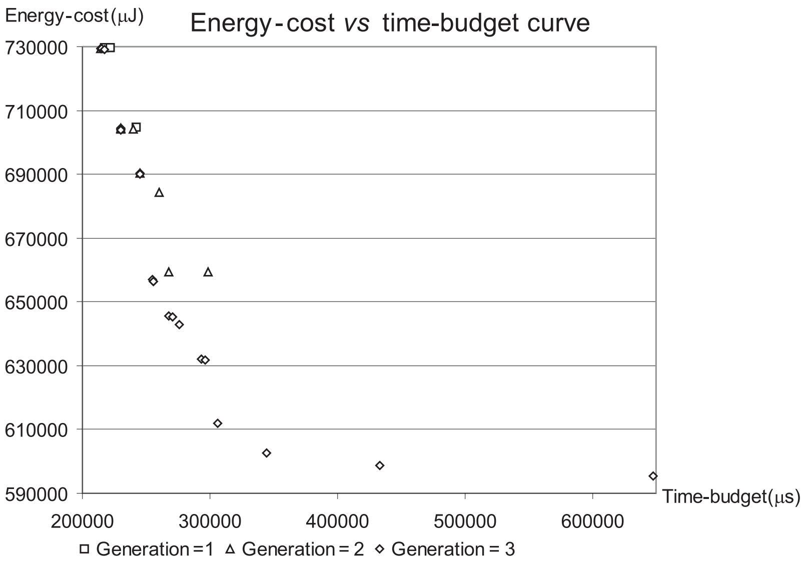

Fig. 4.36. Solution comparison at different *generation* values for Fig. 4.35

execution time and energy data of this example are listed in the appendix of Wong [245].

We first study the effect of the *generation* parameter. Figure 4.36 compares the scheduling solutions at *generation* 1, 2, and 3, respectively. The detailed solution report is listed in the appendix of Wong [245].

To check the solution quality, we have applied the kernel heuristic on this example. The kernel heuristic finds only one solution shown in Table 4.11.

Table 4.11. Scheduling solution produced by the kernel heuristic for Fig. 4.35

Time-budget(μs)	Energy-cost(μJ)
222108	729613

Table 4.12. Run time comparison at different *generation* values and the kernel heuristic for Fig. 4.35

G *Generation*

	G=1	G=2	G=3	Kernel heuristic
Run time (s)	0.47	89.52	3996.6	0.03

Table 4.12 compares the run time at different *generation* values and the run time of the kernel heuristic.

The solution comparison reinforces our conclusions obtained from the previous experiment. That is, increasing the search depth for *generation* improves both the solution quality and the solution range. However, this example also has differences from the previous example.

1. At the low time-budget end, the solutions produced at different *generation* values almost merge together. In other words, the solution quality difference becomes more pronounced at the high time-budget part.
2. The solution counts are not very large for all these three *generation* parameters at 1, 2, and 3, but they are still sufficient.

The only solution produced by the kernel heuristic is among the three solutions produced at *generation* 1. At *generation* 2 and 3, the scheduler produces solutions of better optimality than this solution. We can conclude that the kernel heuristic is not good at solving this type of graph.

Comparing the algorithm run time, we find that the kernel heuristic is very fast. The reason is that its scheduling policy prunes the search space very heavily for this example, so it becomes very fast. As a result, it produces only one solution and that solution is less optimal than solutions found at *generation* 2 and 3. Remember that in the previous example (Fig. 4.21), the kernel heuristic takes comparable run time with the algorithm configured at *generation* 7, which is pretty large. We can conclude that this exhaustive search enhanced scheduler is less sensitive to different problem characteristics. In other words, this scheduler has a more predictable and consistent behavior with different problems. Finally, we can conclude that up to *generation* 3, the algorithm run time is acceptable.

Next, we study the effect of the *weight* parameter. Figure 4.37 compares the scheduling solutions at *weight* values of 3, 6, 9, and 10, respectively. The detailed solution report is listed in the appendix of Wong [245].

From the solution comparison, we can derive similar conclusions as with the *generation* parameter.

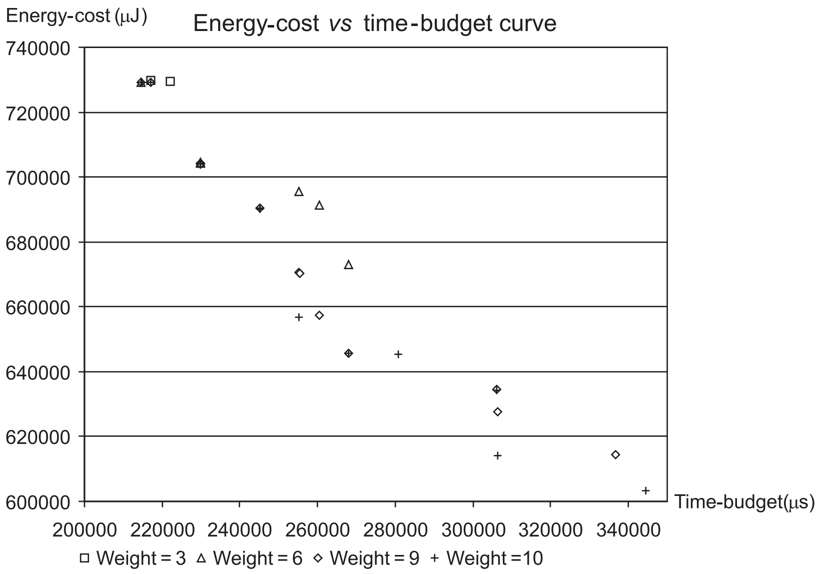

Fig. 4.37. Solution comparison at different *weight* values for Fig. 4.35

Table 4.13. Run time comparison at different *weight* values and the kernel heuristic for Fig. 4.35

W *Weight*

	W=3	W=6	W=9	W=10	Kernel heuristic
Run time (s)	0.34	6.33	836.87	3109.67	0.03

1. Increasing the search depth for *weight* improves both the solution quality and the solution range.
2. At the low time-budget end, the solutions produced at different *weight* values merge almost together. In other words, the solution quality difference becomes more pronounced at the high time-budget part.
3. The solution counts are not very large but still sufficient for all these four *weight* parameters at 3, 6, 9, and 10.

Comparing with the only solution produced by the kernel heuristic, we find again this single solution is among the two solutions produced at *generation* 3. At *generation* 6, 9, and 10, the scheduler produces solutions of better optimality than this solution.

Table 4.13 compares the run time at different *weight* values and the run time of the kernel heuristic.

The algorithm run time at these *weight* values is comparable with the algorithm run time at *generation* 1, 2, and 3. Therefore, the algorithm run time is acceptable.

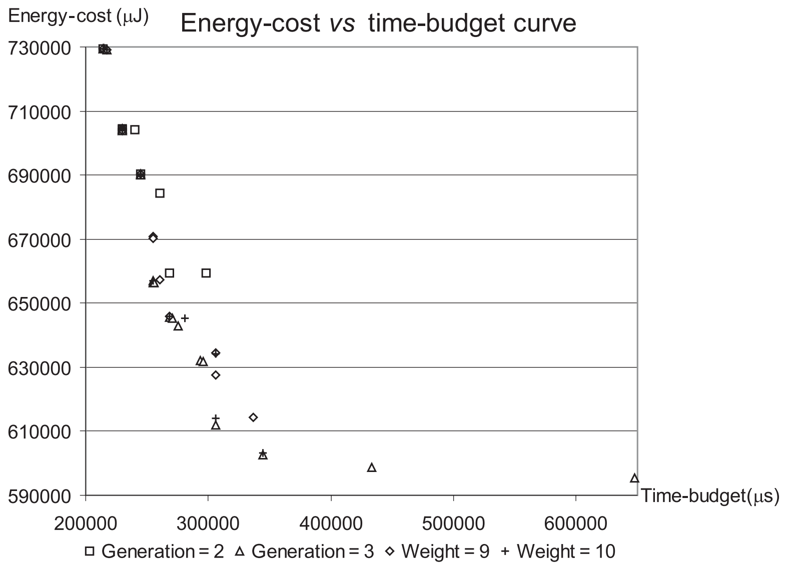

Fig. 4.38. Solution comparison at *generation* 2, 3, *weight* 9, 10 for Fig. 4.35

Comparing the solutions produced at different *generation* values and the solutions at different *weight* values, shown in Fig. 4.38, we have the following conclusions.

1. Solutions produced by these four scheduler configurations are very close to each other at the low time-budget end.
2. At *generation* 3, the scheduler produces the largest solution range while the solutions are of the highest optimality.
3. The algorithm run time at *generation* 3 and *weight* 10 is comparable.

If we remember that with the previous example (Fig. 4.21), the search depth for *weight* plays a more important role regarding the solution quality and solution range, we can further conclude that how to set the search depth for *generation* and *weight* is problem specific. In other words, given the same amount of algorithm run time, for some problems, the *generation* is more effective in producing high-quality Pareto curves (regarding both the solution quality and solution range), while for other problems, the *weight* become more effective. This is a natural outcome since the graph topology and the node execution time distribution can be very different from one graph to another. To get a high-quality Pareto curve, we need to experiment with the configuration of *generation* and *weight* parameters.

Finally, we study the effect of combining exhaustive search for *generation* and *weight* simultaneously. Figure 4.39 compares the solutions at *generation* 2, *weight* 9, and the combination of *generation* 2 and *weight* 9. The detailed

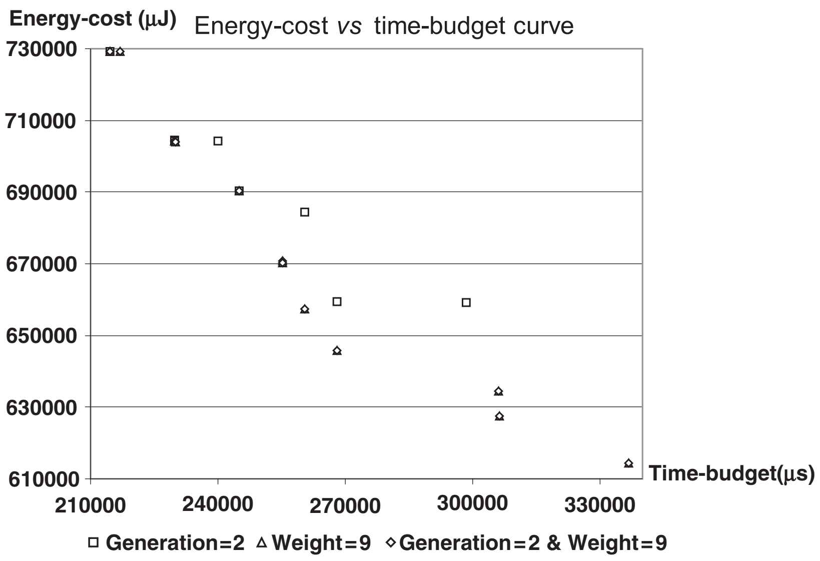

Fig. 4.39. Solution comparison at *generation* 2, *weight* 9, and the combination of *generation* 2 and *weight* 9 for Fig. 4.35

Table 4.14. Run time comparison at *generation* 2, *weight* 9, *generation* 2 and *weight* 9, and the kernel heuristic for Fig. 4.35

W *Weight*
G *Generation*

	G=2	W=9	G=2 & W=9	Kernel heuristic
Run time (s)	89.52	836.87	831.19	0.03

solutions at this *generation* and *weight* configuration are listed in the appendix of Wong [245].

Table 4.14 compares the run time at *generation* 2, *weight* 9, *generation* 2 and *weight* 9, and the run time of the kernel heuristic.

The solution comparison shows that using the configuration of *generation* 2 and *weight* 9, the scheduler produces exactly the same solutions as configured at *weight* 9. The algorithm run time of these two configurations is also very close. By analyzing the graph, it is easy to understand this result. The graph in Fig. 4.35 is a duplicated graph, i.e., the four parallel execution paths are identical. Moreover, the first 6 nodes of each execution path are also identical. And these six nodes are heavier in *weight* than the last node on the execution path. When the search depth for *generation* is set to 2, the first 2 nodes of each execution path will undergo the exhaustive processor assignment. When the search depth for *weight* is set to 9, the first 2 nodes of each execution path and the third node of the first execution path will undergo the exhaustive processor assignment. When the scheduler is configured at *generation* 2 and *weight* 9, the nodes selected by *weight* 9, which undergo exhaustive assignment, contain the nodes selected by *generation* 2. Therefore, the scheduler behaves just as configured at *weight* 9.

This experiment reinforces our conclusion obtained from the previous example. That is, given the same amount of algorithm run time, the combined exhaustive search for *generation* and *weight* is usually not better than the single exhaustive search for either *generation* or *weight*. To produce high-quality Pareto curves, we need to be aware of the topology and node execution time distribution of the input graph. Through experimenting with different configurations of the scheduler, we can find good configuration parameters for a specific problem, which means high-quality Pareto curve and a low algorithm run time.

Finally, it is good to stress that the example of Fig. 4.35 is representative for the graphs which we have extracted from real-life applications. Therefore, the scheduler configuration methods discussed above can be instructive when experimenting with those graphs.

Summary

In this section, we have applied the pruning techniques of Sections 4.3.7 and 4.3.8, and we have extended the scheduling algorithm with exhaustive search. The exhaustive search is applied to nodes according to either its *generation* or *weight*. Using the combined exhaustive search for both *generation* and *weight* usually is not a good solution. The designer can also be in the control of which node should go under exhaustive assignment search.

By using the pruning techniques, we have reduced the algorithm run time, so the scheduler can work with all the graphs we have, either artificially generated or extracted from real-life applications. By introducing the exhaustive search, we have improved both the solution range and the solution optimality. The reason is that the exhaustive search for *generation* and *weight* targets the main problems accurately, so it solves the problem effectively.

4.4 Backward Search Algorithm

In Section 4.3.9, though we have improved both the solution quality and the solution range by exhaustive search on some "important" nodes, the scheduler can still potentially increase the solution range further.

The example in Fig. 4.21 uses two processors, one at 3 V, the other one at 1 V. The cheapest energy-cost solution is at time-budget 2050, where the energy-cost is 2721.89.

Assume that all the nodes are assigned on the 1 V processor, then the energy-cost will simply be the sum of the energy-cost of each node on this

1 V processor. That will be 933.444. Since only one processor is available, no matter how much "concurrency" or "parallelism" exists between the nodes, the platform cannot exploit that. Therefore, all the nodes will be executed sequentially on this only available processor. Moreover, the sequential execution of these nodes can be ordered without any slack in between. And many orderings exist for a no slack schedule. However, their time-budget will be the same, i.e., the sum of all the node execution times on this 1 V processor. That will be 2547.

This example shows clearly that especially on the energy-cost dimension exists a large space for improvement.

The reason that the scheduler stays far away from the energy-cheapest solution is that the node selection policy and the processor selection policy, generically called scheduling policy, do not support the cheap node-to-processor assignment. As discussed in previous sections, all the scheduling algorithms developed so far favor the fastest solution. They all put the performance as their primary importance. From all the extensions made so far to the kernel heuristic, it is obvious that they do not work effectively for increasing the solution range far enough. In other words, we need an algorithm, which favors the energy-cheapest solution. Luckily, it is always straightforward to know the energy cheapest processor for each node. Therefore the scheduler can force such a node-to-processor assignment on each node. Then using a simple algorithm to order these node-to-processor assignments, e.g., the kernel heuristic in Section 4.3.1, we can find a high optimality ordering and hence the final schedule.

This will be used as the starting point of a backward search phase. In addition, the backward search will move some nodes from their energy cheapest processor to the more energy-costly processor, then order those resulted node-processor couples to produce the final schedules with reduced the time-budget. The basic idea is illustrated by Fig. 4.40.

Be aware that due to the heterogeneity of processors, i.e., moving a node from an energy-cheaper processor to a more energy-expensive processor

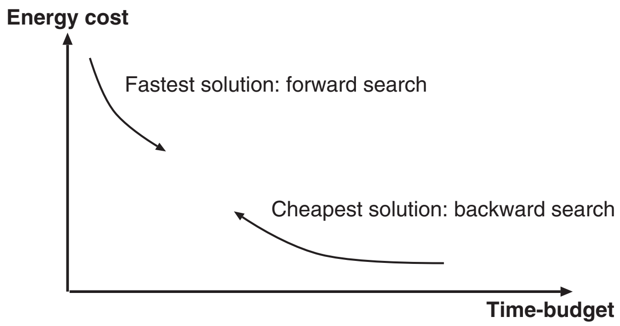
Energy cost
Fastest solution: forward search
Cheapest solution: backward search
Time-budget

Fig. 4.40. Backward search

does not necessarily reduce the execution time of this node. Within the same processor type, to be energy-costly is equivalent to increasing the V_{dd} of a processor, Then to be energy-expensive is equivalent to execution time reduction. However, between different processor types, this does not hold anymore. It is simply because one type of processor can be very inefficient at handling a certain node. No matter how high the energy-cost is, it does not handle the node as well as another type. Even in such a case, sometimes it is still beneficial to pay the energy-cost because the node moved away from one processor makes room for other nodes to execute earlier on this processor. Therefore, the total time-budget can be reduced.

In summary, trying to move the node from its energy-cheapest processor to a more energy-expensive processor can potentially reduce the time-budget. In other words, it can potentially increase the performance.

When it comes to how to move the node, many options exist. One main reason for a slow schedule is that the nodes, which can be executed in parallel, are sequentially squeezed on one processor. Therefore, we can try to reduce the time-budget by distributing some nodes, which were originally assigned to one processor, and which can be executed in parallel, to different processors.

Refining the above idea, we identify the following steps.

Step 1: Initial Phase Start with the energy-cheapest node-to-processor binding, i.e., every node assigned to its energy-cheapest processor. To avoid misunderstanding, and to make the text easy to read, we have to make the following definition.

Definition 4.1 (Binding) *Given a graph and a platform, a binding is the aggregation of every node-to-processor assignment.*

On certain platforms, a node can have two or more energy-cheapest processors. Those processors can be either identical processors, or different types of processors. Energy-wise speaking, as long as all the nodes are assigned to their energy-cheapest processors, no matter how the nodes are distributed among the processors, the energy-cost will always remain the same. In other words, the energy-cost of a schedule is only decided by its binding, not by how the nodes are ordered on each processor. However, under the same energy-cost, the time-budget of the final schedule can be different. Two reasons exist for that. First, if the node-to-processor binding results in an ill-balanced workload among the processors, e.g., one processor has many nodes assigned to it while another is almost idle, and if the graph allows parallel execution on those processors, which is now destroyed by the workload distribution, the final schedule will have a larger time-budget than it can reach under a better workload distribution. Second, given the same workload distribution, if the node execution is badly ordered, due to the precedence constraints, some nodes will be unnecessarily delayed. Therefore, the time-budget of the final schedule is also larger than a better ordering can make.

However, the first special case can be handled by carefully distributing the nodes among the processors. When every node has only one energy-cheapest processor, and actually, this is the normal case for most experiments in our study, this preprocessing step can be skipped. The second special case is guaranteed by the node selection policy of the our scheduling algorithm. Though all the algorithms we have discussed so far handles the node selection and the processor selection in separate steps, if we apply the node selection policy to node-processor couples, we get good ordering solutions as well. We have demonstrated this by the experiments in this section.

Step 2: Generating Time-budget Intervals using the scheduling heuristic, we get the least time-budget solution. This solution is also the most energy expensive solution. Then from the cheapest energy-cost to this most expensive energy-cost, we can divide this space into a number of intervals. The number of intervals can be set by the user. The boundaries of these intervals form a number of threshold values. When adapting the cheapest node-to-processor binding incrementally, we raise the energy-cost step by step, and the adapted bindings target these threshold values.

Step 3: Adapting Bindings Starting with the energy-cheapest binding, starting with the source nodes of the graph, when two or more nodes independent from each other exist, check their processor assignments. We identify the following three cases.

If the number of independent nodes is less than or equal to the processor number and all these independent nodes are assigned to different processors, i.e., they will be potentially executed in parallel, we skip these nodes and go on to check their child nodes. Why "potentially"? Because due to the execution time of their parents nodes, it is possible that they do not overlap in their execution time. Anyhow, the physical location, i.e., their different processor assignments allow them to be executed in parallel.

If the number of independent nodes is larger than the processor number and all the processors have one or more node assigned to it, we skip these nodes and go on to check their child nodes. In this case, though the graph provides more parallel execution possibility, the platform cannot utilize the parallelism. However, since no processor is idle, the platform has used the parallelism at its best. From this example, we distinguish the concept of "parallelism" from two origins, one from the graph, the other from the platform. When the graph is rather "sequential", i.e., only a few nodes are independent from each other, it is no use to execute the graph on a platform consisting of many processors. On the contrary, if the graph is very "parallel," it is better to use a large number of processors to execute the graph. In summary, the match between the "parallelism" of the graph and the "parallelism" of the platform provides the fast execution of the graph. Increasing the processor number without knowing the graph topology is a waste of resources.

Finally, when one or more idle processors exist and two or more nodes are assigned to the same processor, move one of the nodes occupying the same processor to an idle processor. Currently, we select the energy-cheapest move among all the possible moves. Finely tuning the move policy can be a future extension to this algorithm. If the energy increment reaches one of the energy-cost threshold values, the adaptation is recorded. Otherwise, we continue to find idle processors and move one of the nodes occupying the same processor to an idle processor. Each time we move one node until the energy increment reaches the next energy-cost threshold value.

When all the possible moves are exhausted or when all the energy-cost threshold values are met, the binding adaptation finishes. Then the scheduling algorithm will be applied on those bindings to find the final order of the node execution. Since the node and processor are bound together, we can also say the node execution order is the execution order of the node-processor couple.

Discussion 1 When adapting the bindings, we are basically checking the nodes assigned to one processor, which can be executed in parallel. Currently, we check the nodes *generation* by *generation*. That is to say, only the nodes of the same *generation* are checked to see whether more "parallelism," which is destroyed by the processor assignment, can be released and utilized. This is illustrated by Fig. 4.41. In this figure, only the "parallelism" of node t_1, t_2, and t_3, or the "parallelism" of node t_5, t_6, t_7, and t_8 are checked to see whether they are fully utilized. However, as illustrated by Fig. 4.42, "parallelism" also exist, e.g., in node t_1, t_6, t_7, and t_8. These four nodes do not have any dependency between each other. The same thing is also true for other node groups, e.g., node t_2, t_3, t_4, and t_5.

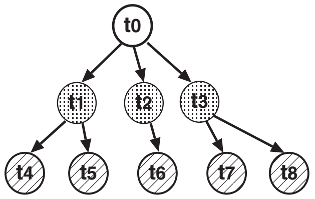

Fig. 4.41. Parallelism checking within the same *generation*

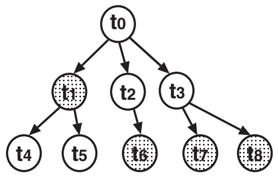

Fig. 4.42. Cross-*generation* parallelism checking

For a realistic size graph, the number of such kind of node groups can be very large. If we explore all the possibilities, the algorithm run time is not affordable. However, as we discussed before, at the assignment stage, we do not know the exact node execution order. Though we assign two independent nodes to different processors, probably in the final schedule, they do not overlap in execution time. That is to say, viewed from the time axis, they are executed sequentially on different spatial locations, i.e., on different processors. So much potential "parallelism" exist, which cannot be utilized in reality. To prune the search space, we just check the nodes of the same *generation*. Tuning the "parallelism" checking policy can be a future extension to this algorithm.

Discussion 2 As discussed earlier, having found a number of node-processor bindings, we need an algorithm to order the execution of assigned node-processor couples for each binding. The scheduling algorithm developed so far does the node selection and processor selection in two steps. The node selection is based on *generation*, *load*, and *weight*. Since the node and processor are bound together here, the processor selection does not exist any more. Therefore, if we apply the node selection policy to the bindings, it can do the ordering for all the assigned node-processor couples.
However, a deeper thought reveals also the following facts. So far, the node *weight* is calibrated on a reference processor. This is not an ideal solution. If we know exactly where the node is going to be executed, we should certainly use the execution time on that processor to help us make scheduling decisions. Exact information is more effective than uncertain information in decision-making. It is only because we lack this information that we use this better-than-nothing measurement. Now that the binding is given, the execution time of each node becomes fixed with a specific binding, we exploit this information by a new parameter called *dynamic weight*. It is dynamic in the sense that it can take different values with different processor assignments. Accordingly, *dynamic load* is also introduced to reflect the fact that we know the node execution time exactly.

We have implemented this backward search extended scheduling algorithm. Two examples are used to study the algorithm. One is Fig. 4.11, the other is Fig. 4.35.

4.4.1 Experiment with Fig. 4.11

For this example, the scheduler is configured at the search depth of three for both *generation* and *weight*. Due to the heavy search space pruning introduced in Section 4.3.9, we have found that the scheduler produces a reasonably large solution range when configured at a search depth bigger than 0, e.g. 2 or 3. And the algorithm run time is very small at such configurations, e.g. 10s to 20s. That is the reason that we set the search depth for both *generation*

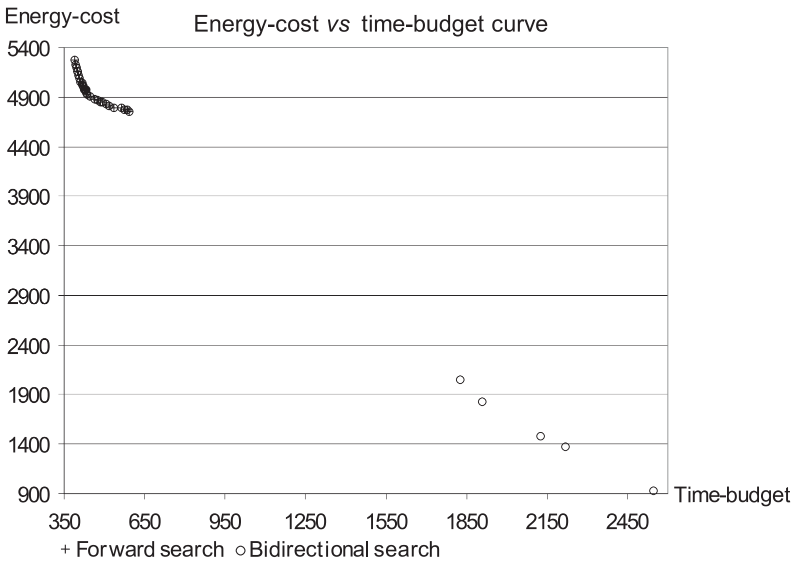

Fig. 4.43. Comparing solutions produced by the forward and bidirectional search for Fig. 4.11

and *weight* at 3. The number of time-budget intervals is the default value 9, i.e., the number of the boundaries of these 9 intervals is 10. The detailed scheduling solutions are listed in the appendix of Wong [245].

The forward search produces 29 solutions while the bidirectional search produces 34 solutions. Checking the solutions, we find that the first 29 solutions produced by the bidirectional search are identical with the 29 solutions produced by the forward search. Figure 4.43 compares the solutions produced by the forward and the bidirectional search. We have three observations from this figure.

1. The number of low time-budget solutions is dominant. The backward search only produces 5 solutions at the lowest energy-cost end, which is a small fraction of the solutions.
2. The solutions found by the forward search are of high density while the solutions found by the backward search are sparsely distributed.
3. A reasonably big gap exists between the solutions found by the forward search and the backward search.

Graph in Fig. 4.11 has 1 source node and 8 *generation*s. The first 3 *generation*s have 1 node each. That is to say, no "parallelism" can be utilized at these 3 *generation*s. Our current policy utilizes the parallelism *generation* by *generation*, i.e., no cross-*generation* is utilized. Therefore, we can already make a rough estimation, namely, starting from the energy-cheapest binding how many bindings we can get through adapting the energy-cheapest binding.

Table 4.15. Run time comparison between the forward search and the bidirectional search for Fig. 4.11

	Forward search	Bidirectional search
Run time (s)	22.02	22.34

The number should be 5 due to the lack of "parallelism" in the first 3 *generation*s. This explains why the backward search finds only 5 extra solutions on the energy-cheap side.

The number of energy-cost intervals is set at 9, which is the default value. The cheapest energy-cost is 933.444, the highest energy-cost is 5271.36. Therefore, the step size of the energy-cost is 481.99. Our solutions (in the appendix of Wong [245]), follow this energy-cost increment scheme. We cannot expect that they follow this scheme strictly because we are treating a discrete mathematic problem, i.e., we can neither split a node into arbitrary granularities nor change the processor supply voltage continuously. This explains why these 5 solutions produced by the backward search are distributed at much larger intervals than solutions produced by the forward search. It also explains that why the backward search stops at time-budget 1827, which leaves a big gap from time-budget 594, which is the slowest solution found by the forward search.

Finally, Table 4.15 compares the algorithm run time of the forward search and the bidirectional search. Apparently, for this example, the algorithm run time difference is negligible.

4.4.2 Experiment with Fig. 4.35

Next, we proceed with the example in Fig. 4.35. We compare the forward search and the bidirectional search algorithm both configured at *weight* 9 because the forward search produces high-quality solutions at this configuration. The number of energy-cost intervals is 9, the default value. The detailed solutions of the bidirectional are listed in the appendix of Wong [245].

The forward search produces 13 solutions for this example while the bidirectional search doubles the solution number, namely 26. We find that the first 13 solutions of the bidirectional search are identical with the those of the forward search. Figure 4.44 compares the solutions produced by the forward search and the bidirectional search.

Contrary to the previous experiment, this time, the backward search produces a reasonable number of solutions. These solutions are distributed evenly on the energy-cost axis and no significant gap exists between the forward search and the backward search. In summary, the backward search works much better for this example than the example in Fig. 4.11.

Two reasons lead to this difference. One is the topology between the graphs. The graph in Fig. 4.35 has more "parallelism" to utilize within each

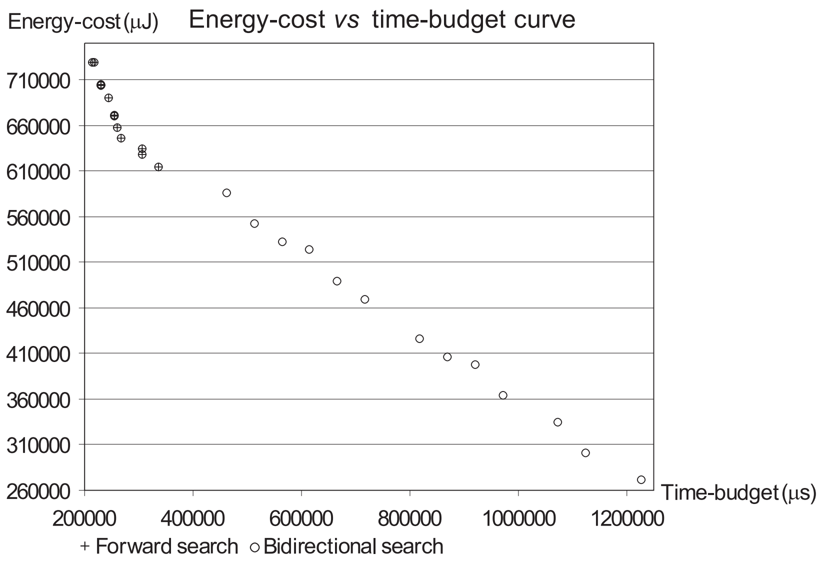

Fig. 4.44. Comparing solutions produced by the forward and bidirectional search for Fig. 4.35

generation than the graph in Fig. 4.35. The graph in Fig. 4.35 is a duplicated graph, i.e., the four parallel execution paths are identical. It has "parallelism" in all these 7 *generation*s.

The other reason is the node execution time distribution. For each boundary of the energy-cost intervals, the scheduler tries to find two bindings, one with a lower energy-cost, the other with a higher energy-cost. However, because the node execution time is a discrete value, we cannot guarantee two bindings for each boundary. For example, if a binding has exactly the same energy-cost with one of the boundary value, the scheduler will only take this binding. Sometimes, the scheduler can only find one binding closest to the boundary value. The next energy-cost higher binding will be the closest binding to the next higher boundary value, or even the closest binding to the next next higher boundary value. In the latter case, no binding exists for the skipped boundary value. In this example, the scheduler finds two bindings for some of the boundary values. After ordering those bindings and checking the optimality, 13 solutions remain.

Table 4.16 compares the algorithm run time of the forward search and the bidirectional search. Again, for this example, the algorithm run time difference is negligible. Actually, the run time is even smaller for the bidirectional search. But apparently, this is due to the inaccurate measurement on Unix. Unix is a time-sharing operating system. We cannot dedicate the server for running this algorithm only. Every time when measuring the time, the workload of the server is not the same. In addition, the resolution is only

Table 4.16. Run time comparison between the forward search and the bidirectional search for Fig. 4.35

	Forward search	Bidirectional search
Run time (s)	832.43	828.53

0.01 s, which is pretty low. However, from this rough measurement, what we can be sure about is the algorithm run time difference between the forward and the bidirectional search will be negligible. In other words, the backward search does not increase the run time significantly.

4.4.3 Summary

In summary, the backward search works well for graphs, which has its "parallelism" distributed evenly among its *generation*s. For graphs, like Fig. 4.11, we need further extensions to cover the gap between the forward search and the backward search. This is the topic of the next section.

4.5 Subplatform Scheduling

The experiments in Section 4.4 show that the bidirectional search does not cover the gap between the forward and backward search. Moreover, if we look at the experiment result in Fig. 4.44, we can see that the energy-cost and time-budget trade-off curve goes almost linearly. This is again not what we expect because the energy-cost reduces quadratically with the supply voltage while the execution-time increases linearly with the supply voltage. Of course, our scheduling problem is a discrete mathematic problem. It is discrete because the node execution time can only take a few values. And the parallel execution introduces even more irregularity in this problem. However, it is good to think this way and check whether we can improve further.

By checking the results in the middle range of Fig. 4.44, we can identify that some nodes are assigned to high V_{dd} processor unnecessarily. It seems that the scheduler builds up some bindings only for the sake of increasing energy-cost. It does not consider whether moving nodes to high energy-cost processors is the most optimal move. Figure 4.45 illustrates such a situation. Suppose node t_7 is only dependent on node t_4 and t_8 is only dependent on t_5, we can move them to the idle time-slots as illustrated. Such a change reduces the energy-cost while keeping the time-budget as before.

This example shows us that we can get energy-cheaper schedules by restricting the processor selection. Since the forward search is a fast algorithm, we can derive all the possible subsets of the given platform, then apply the forward search on each of the derived platforms, and finally select the

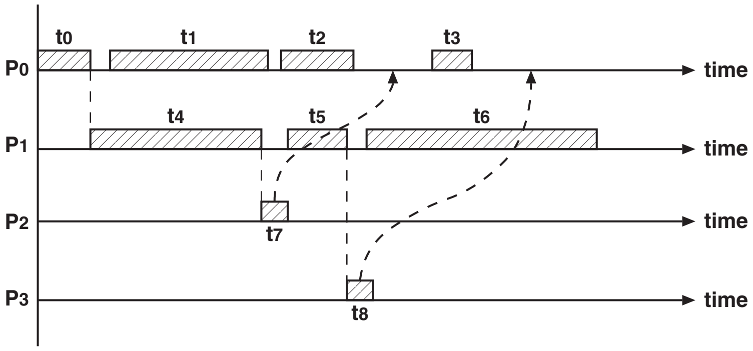

Fig. 4.45. Illustration of unnecessary high V_{dd} processor assignments

Table 4.17. Sub-platform numbers of typical processor numbers

Processor Num	Sub-platform Num
2	3
3	7
4	15
5	31
6	63
7	127
8	255

optimal solutions out of the intermediate solutions. Since the derived platforms are subsets of the given platform, the name "sub-platform" scheduling has been selected for this heuristic.

Generally speaking, for a given platform of n processors, the number of possible sub-platforms is the sum of C_n^1, C_n^2, $\ldots$, C_n^{n-1}, C_n^n, i.e., $\sum_{i=1}^n C_n^i$. By mathematical induction, we can prove that $\sum_{i=1}^n C_n^i = 2^n - 1$. Table 4.17 shows the sub-platform numbers for typical processor numbers. Given these numbers, we can conclude that the algorithm run time should still be affordable for typical cases.

To make easy comparisons, we have used the same two examples in Section 4.4 to study the sub-platform scheduling.

4.5.1 Experiment with Fig. 4.11

We still configure the search depth for both *generation* and *weight* at 3. The detailed solutions produced by sub-platform scheduling are listed in the appendix of Wong [245]. The sub-platform scheduling produces 64 solutions, which almost doubles 34, i.e., the solution count of the bidirectional search, Fig. 4.46 compares the solutions produced by the bidirectional search and the sub-platform scheduling.

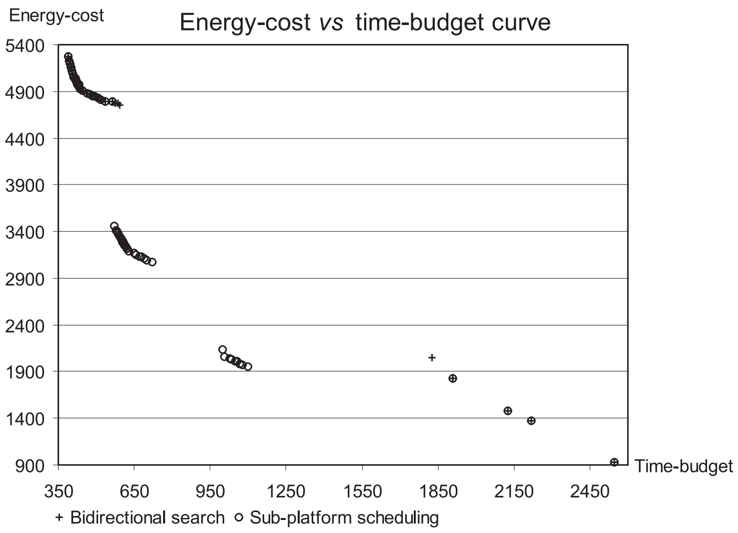

Fig. 4.46. Comparing solutions produced by the bidirectional search and the sub-platform scheduling at search depth 3 for *generation* and 3 for *weight* for Fig. 4.11

Table 4.18. Algorithm run time of the bidirectional search and the sub-platform scheduling at *generation* 3 and *weight* 3 for Fig. 4.11

	Bidirectional search	Sub-platform scheduling
Run time (s)	22.34	44.91

At the low time-budget part, the sub-platform scheduling produces identical solutions with the bidirectional search. In general, the sub-platform scheduling produces solutions in four groups. At time-budget 564.353 and 719.8, there are cliffs in the energy-cost axis. These big jumps in energy-cost say that at those time-budgets reducing the time-budget will incur a high-energy penalty. Also at energy-cost 1949.63 is a cliff in the time-budget axis, which says the energy-cost is not sensitive to the time-budget. In other words, when we reduce the time-budget, the energy-cost remains flat.

Such a solution distribution is good enough for normal use. Because when the energy-cost has a big jump, the time-budget does not jump. Vice versa, when the time-budget has a big jump, the energy-cost remains flat. As long as the jump does not happen in both dimensions simultaneously, or in other words, at least one dimension is continuously changing, the trade-off curve is usable.

Table 4.18 compares the algorithm run time of the bidirectional search and the sub-platform scheduling. Apparently, the run time overhead introduced by sub-platform scheduling is affordable. Especially considering the gain in solution space, this overhead is worthwhile.

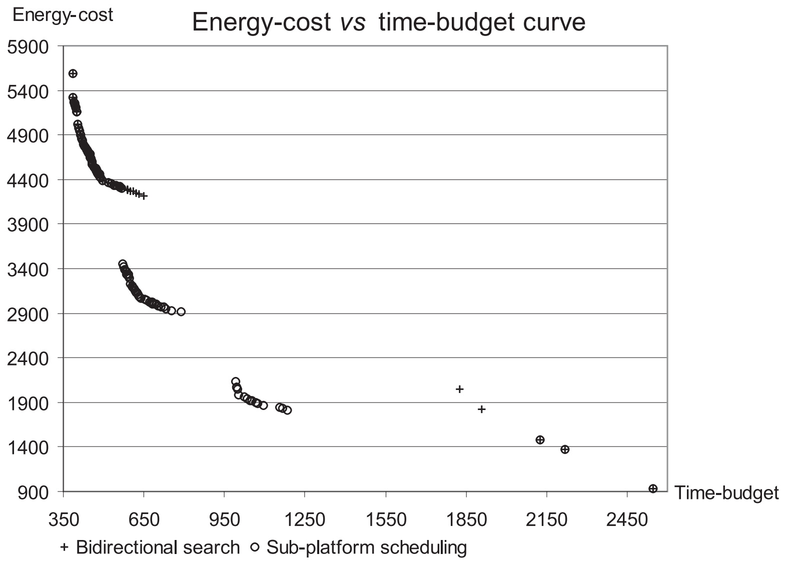

Fig. 4.47. Comparing solutions produced by the bidirectional search and the sub-platform scheduling at search depth 3 for *generation* and 6 for *weight* for Fig. 4.11

Table 4.19. Algorithm run time of the bidirectional search and the sub-platform scheduling at *generation* 3 and *weight* 6 for Fig. 4.11

	Bidirectional search	Sub-platform scheduling
Run time (s)	1104.87	1558.69

It is clear that, for this example, to get a better solution distribution, i.e., to reduce the gaps on either the time-budget dimension or the energy-cost dimension, we have to increase the search depth for *generation* and/or *weight*.

Figure 4.47 compares the solutions of the bidirectional search and the sub-platform scheduling when they are configured at *generation* 3 and *weight* 6. The detailed solutions are listed in the appendix of Wong [245]. Compared with Fig. 4.46, the solutions are distributed more smoothly at the algorithm run time penalty shown in Table 4.19.

Increasing the search depth for *generation*, we get even smoother solution distribution for this example. Figure 4.48 compares the solutions of the bidirectional search and the sub-platform scheduling when they are configured at *generation* 6 and *weight* 3. The detailed solutions are listed in the appendix of Wong [245]. As expected, we pay an even higher run time price for the improved solution quality. Table 4.20 shows the algorithm run time.

Figure 4.49 illustrates how the solution quality and distribution improve when the search depth for *generation* and *weight* increase. The solutions are

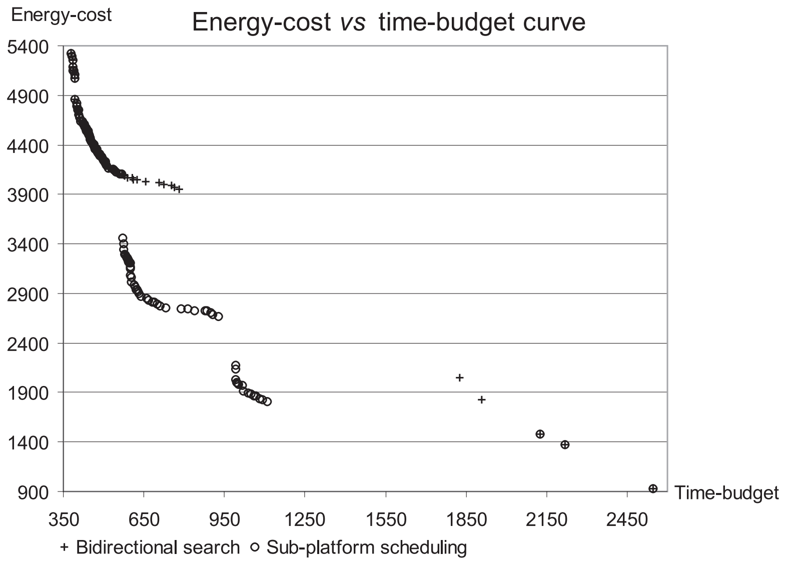

Fig. 4.48. Comparing solutions produced by the bidirectional search and the sub-platform scheduling at search depth 6 for *generation* and 3 for *weight* for Fig. 4.11

Table 4.20. Algorithm run time of the bidirectional search and the sub-platform scheduling at *generation* 6 and *weight* 3 for Fig. 4.11

	Bidirectional search	Sub-platform scheduling
Run time (s)	2436.63	2964.44

produced by the sub-platform scheduling. The scheduler is configured at *generation* 3 and *weight* 3, *generation* 3 and *weight* 6, and *generation* 6 and emph-weight 3.

We have tried to increase the search depth even further, e.g., *generation* 6 and *weight* 6. However, the solution quality and distribution does not improve significantly. When configured at *generation* 6 and *weight* 6, the scheduler produces two more solutions than configured at *generation* 6 and *weight* 3. Most of the solutions produced with these two configurations are identical. Considering the scheduler already has 119 solutions at *generation* 6 and *weight* 3, this improvement is negligible.

For discrete mathematic problems, the gap is due to the discrete nature of the problem itself. When a scheduling solution does not exist in the gap, no matter which algorithm we use, we simply cannot create something, which does not exist.

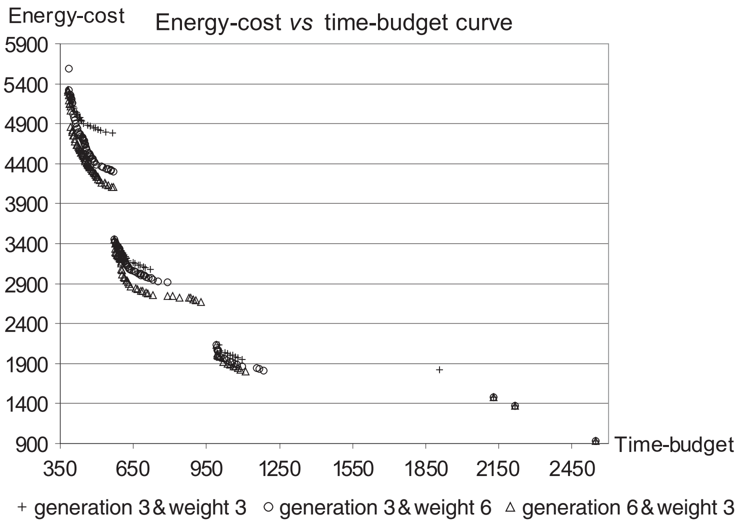

Fig. 4.49. Comparing solutions produced by the sub-platform scheduling at different search depths for Fig. 4.11

4.5.2 Experiment with Fig. 4.35

Applying the sub-platform scheduling to the example in Fig. 4.35, we get 38 solutions (listed in appendix of Wong [245]). The scheduler is configured at *weight* 9, which is the same as in the experiment of Section 4.4.2. Figure 4.50 compares the solutions produced by the bidirectional search and the sub-platform scheduling.

The following conclusions can be derived from this comparison.

1. The solution count increases from 26 to 38, which is approximately 50%.
2. Both in the low time-budget part and in the energy-cost cheap part, the solutions produced by these two schedulers are very close. Actually, they come up with identical solutions at many places.
3. In the middle range, sub-platform scheduling improves the solution quality significantly.
4. The overall solution distribution of sub-platform scheduling is as good as the bidirectional search.

Table 4.21 compares the algorithm run time between bidirectional search and sub-platform scheduling. Compared with the bidirectional search, the run time of sub-platform scheduling increases 52.1%. However, considering the improvement in solution quality, the penalty is worthwhile.

4.5.3 Summary

Through these two experiments, we can derive the following conclusions about the sub-platform scheduling.

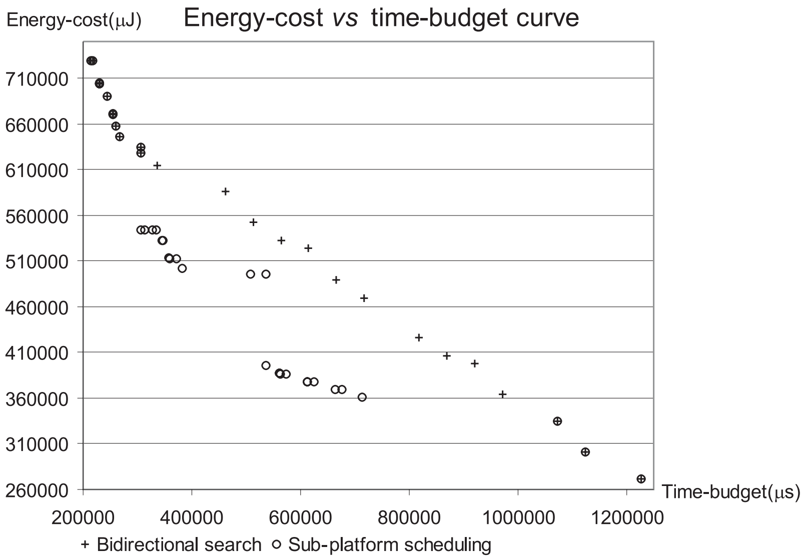

Fig. 4.50. Comparing solutions produced by the bidirectional search and the sub-platform scheduling for Fig. 4.35

Table 4.21. Run time comparison between the bidirectional search and the sub-platform scheduling for Fig. 4.35

	Bidirectional search	Sub-platform scheduling
Run time (s)	828.53	1206.38

1. It helps covering the gap in time-budget dimension and energy-cost dimension.
2. It improves the scheduling solution quality especially in the middle range.
3. It produces a better solution distribution, which means it produces more solutions at the places, where energy-cost changes sharply with the time-budget.
4. It introduces a reasonable algorithm run time overhead, which is affordable and worthwhile.

4.6 Handling Timing-Constraints

In our target domain of embedded systems, it is not rare to have timing constraints between two nodes. We have classified the timing constraints in the following four types from experiments with real-life applications. Assume t_1 and t_2 are two nodes in a graph. They can be directly connected, indirectly connected or independent from each other.

1. Minimum timing constraint from the finish of t_1 to the start of t_2. This happens when, for example, t_2 waits for the data from t_1 to start execution, t_1 only produces the data at the end of its execution, and the data produced by t_1 can only become available to t_2 after the minimum timing constraint due to some kind of hardware delay.
2. Minimum timing constraint from the start of t_1 to the finish of t_2. This happens, when for example, t_2 waits for the data from t_1 to finish execution, t_1 produces data as soon as it starts execution, but the data can only become available to t_2 after the minimum timing constraint due to some hardware delay.
3. Maximum timing constraint from the finish of t_1 to the start of t_2. This happens when, for example, t_2 needs data from t_1 to start execution, t_1 only produces the data at the end of its execution, and the data will go corrupted after the maximum timing constraint.
4. Maximum timing constraint from the start of t_1 to the finish of t_2. This happens, when, for example, t_2 needs data from t_1 to finish execution, t_1 produces the data at the start of its execution, and the data will go corrupted after the maximum timing constraint.

Figure 4.51 illustrates these four types of timing constraints. This section extends the scheduling algorithm to handle the timing constraints. The basic idea is to use timing-constraints to prune the solutions before checking its optimality. The order of the timing-constraint pruning and the optimality checking is important. Because we can only talk about the optimality after the solution is valid in timing constraints. This means for every solution found by the scheduler, the scheduler will check the solution's validity first regardless of its optimality. That is, checking the solutions against all the timing constraints, which means an overhead of the algorithm run time. There are two consequences of such a pruning step. One is that at some time-budgets, we will not find any solution at all due to the timing constraints. The other is that some solutions, which are non-optimal without timing-constraints, become optimal.

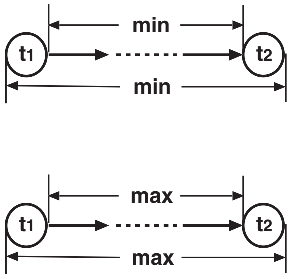

Fig. 4.51. Four types of timing constraints between two nodes

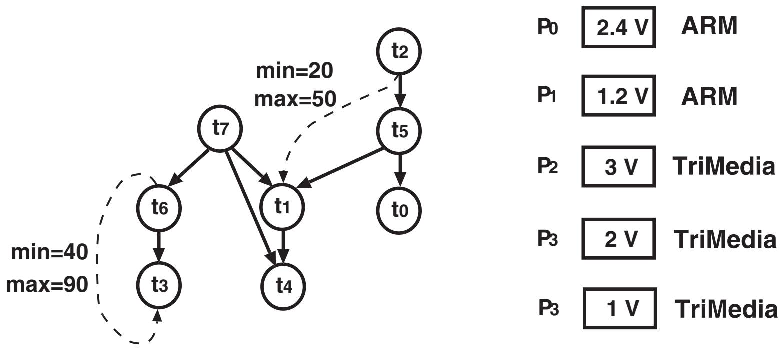

Fig. 4.52. An example task graph with timing constraints

A small example of 8 nodes with 5 processors and 4 timing constraints as shown in Fig. 4.52, is used to illustrate how this pruning step works. Timing constraints are represented with dash lines in the graph. Both the scheduler without and with timing constraints are configured at search depth 1 for both *generation* and *weight*. Table 4.22 lists the solutions produced by these two schedulers. This table shows that due to timing constraints, some scheduling solutions become invalid, e.g., at time-budget 64, 117, 133, 134, 162, and 210. Those scheduling solutions violate the timing constraints, so they are ruled out by the pruning step. At other places, e.g., at time-budget 72, the original optimal solution is replaced by an originally non-optimal solution. Like discussed before, the optimal solution violates the timing constraints while the non-optimal solution meets the timing constraints, so it survives the pruning step.

Table 4.23 compares the algorithm run time of the scheduler without and with timing-constraint pruning. As expected, the timing-constraint pruning introduces a run time overhead. This is really a tiny example. When the problem size goes big, checking every outcome solution against a set of timing constraints can be a large overhead.

In summary, handling timing-constraints as a pruning step is a safe way to solve the problem. It will not reduce the search space, hence it will keep the solution quality. However, it may not be very efficient in algorithm run time. An alternative is to treat the timing-constraints in the scheduling algorithm. Apparently, this increases the algorithm complexity and also the run time. With careful algorithm design, especially by using the hierarchical approach of the next chapter, we may expect a reasonable run time. That is a direction for future extension to this scheduling algorithm. However, treating the scheduling and timing constraints separately gives us a chance to switch the pruning step off completely. Therefore, we can save algorithm run time very effectively for those problem without timing-constraints.

Table 4.22. Comparing solutions produced by the algorithms with and without timing-constraint handling

TB: Time-budget
E: Energy-cost
TC: Timing-constraint

TB	E	
	Without TC handling	With TC handling
64	1109	
72	1026	1086
77	613	613
90	568	568
108	490	490
117	487	
133	448	
134	436	
137	376	376
162	158	
210	156	

Table 4.23. Run time comparison between the scheduler without and without timing-constraint handling for Fig. 4.52

TC: Timing-constraint

	Without TC handling	With TC handling
Run time (s)	0.24	0.37

4.7 Summary

This chapter gives a detailed explanation of our basic design-time scheduling algorithm. This algorithm can be applied on individual TFs and generate energy vs performance trade-off points for each TF. The heuristics used in this algorithm work well with small to medium sized TFs (i.e., <100 TNs). For extremely large TFs, a more scalable design-time scheduling technique has been developed and will be discussed in the next chapter.