5 Scalable Design-Time Scheduling

Task scheduling on multiple heterogeneous processors is notorious for its computation intractability. Although previous design-time task schedulers have tackled this intractability with fast heuristics, it remains time-consuming to explore the design space for large input TF. This chapter presents a novel method that combines the graph partition and the TF interleaving technique to tackle the trade-off exploration problem in a scalable way. Based on this method, we have developed a hierarchical scheduler that can employ the existing design-time schedulers and can significantly accelerate the design space explorations for large TF.

Section 5.1 briefly introduces the problem addressed in this chapter. Section 5.2 gives an overview of the scalable design flow. Section 5.3 explains the details of the decomposition process. Section 5.4 shows how to deal with the decomposed partitions. Section 5.5 discusses the novel interleaving technique for the post-decomposition process. Section 5.6 presents our experimental results and Section 5.7 lists the related work. Finally, Section 5.8 concludes this chapter.

5.1 Introduction

Today's embedded software becomes much larger, and contains large portions of static software components such as the multimedia application codecs. When mapping those complex software components onto the hardware platform, system designers need a good synthesis tool that can make trade-offs for multiple objectives such as the energy consumption and the system performance. The first step of such a synthesis tool is typically a platform-independent parallelism analysis and optimization such as the work by Yang and Gerasoulis [254] and Stahl et al. [228]. The basic assumption of most platform-independent task parallelism analysis is an unbounded parallel platform which leads to a very large scale concurrent task model (i.e., a TF) with all inherent parallelism explicitly specified in the software specification. This large-scale model however has unnecessary details when mapping the model on a practical platform with limited processors and other physical constraints such as bandwidth and capacity limitation over the memory hierarchy. Moreover, an overdetailed concurrency model often leads to a extremely long scheduling process. Because most scheduling problems encountered during system-level synthesis are difficult problems in terms of computational complexity, conventional scheduling algorithms suffer from a lengthy execution when scheduling larger TF. A divide-and-conquer (DNC) strategy is normally employed on the TF in order to reduce the scheduling complexity. However, conventional DNC algorithms often partition the TF based on the criterion defined by the number of threads, which is not the only essential reasons that resulted in the long computation time in task-level design space exploration. Moreover, those DNC algorithms neglect the fact that the resulting thread partitions can be partly run in parallel, which can be exploited to increase the exploration quality at a very low cost. In other words, conventional DNC approaches lack a post-DNC step which could partly compensate the final performance-energy trade-off exploration results for the loss of scheduling optimality due to the dividing.

This chapter addresses how to deal with very large task concurrency models as well as how to exploit the concurrency among the partitioned thread partitions. We propose an effective TF partitioning algorithm to split up the large input TF into smaller thread partitions and we schedule each partition separately. Then we use a novel design-time algorithm to generate interleaved schedules based on the separate schedules of TF. We have implemented this combined approach in a hierarchical scheduler that can deal with very complex applications with many tasks in a scalable way. Our hierarchical scheduler can decrease the performance-energy trade-off exploration time by up to 2 orders of magnitude compared to a flattened approach while the exploration results have much lower energy consumptions.

5.2 Motivational Example

The proposed hierarchical scheduling approach mainly consists of five steps, namely the input TF decomposition, the design space exploration of each thread partition, the thread partitions clustering, the thread partition interleaving, and thread partition merger (Fig. 5.1). Note that the Pareto curves indicated in the overview figure are actually sets of Pareto-optimal schedules of individual thread partitions. Why the Pareto-curves are better than the individual optimal schedule should be clear after the earlier discussion in Chapter 1.

In the first step, the input TF is decomposed into a set of thread partitions under certain guidelines. Each thread partition contains a number

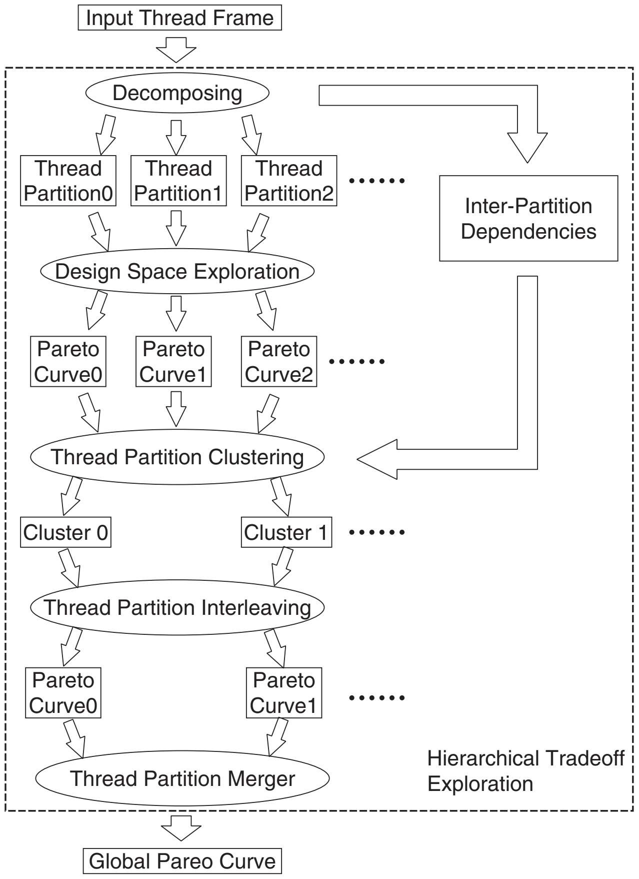


Fig. 5.1. Hierarchical scheduling overview

of thread nodes from the original TF and the corresponding edges among those thread nodes. The edges which are cut during the decomposition are removed from the thread partitions. Instead, new edges are created between thread partitions such that the dependent relationships among thread nodes are preserved.

In the second step, we use a design-time scheduler to explore the design space of each individual thread partition and produce a set of Pareto-optimal schedules for each thread partition. An effective example of that scheduler is the basic TCM design-time scheduler discussed in the previous chapter.

In the third step, we put thread partitions into a sequence of clusters such that partitions in each cluster have no dependencies among them. This is the preparation for the interleaving in the next step.

In the fourth step, which mainly distinguishes our work from conventional DNC approaches, we interleave the Pareto-optimal schedules of those thread partitions that can be run in parallel, i.e., partitions in each cluster.

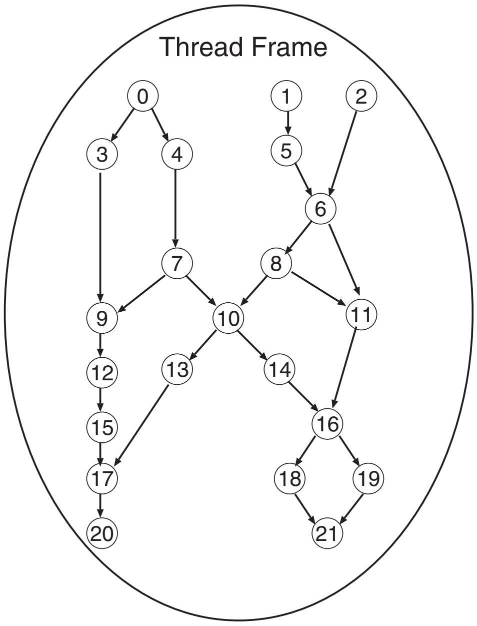


Fig. 5.2. Input thread frame example

Table 5.1. Profiling data for each thread node

	VLIW	RISC
Energy consumption (μJ)	485.438	1040.55
Execution time (μs)	177.816	1009.69

The interleaved schedules form the new Pareto-optimal schedules which are presented in a Pareto curve for each cluster.

In the last step, we simply merge all Pareto curves from each cluster and thus constitute the global performance-energy trade-off Pareto curve for the entire input TF.

Example 5.1 Consider a design-time exploration problem of mapping a given TF onto a three-processor platform (shown in Fig. 5.2). We assume that these three processors are one VLIW processor running at 1.56 V and two scalar RISC processors running at 1.08 and 1.62 V. For simplicity, we let all threads have the same amount of execution time and energy consumption on each processor, namely all threads are the same. The execution times and energy consumptions are given in Table 5.1. Only the profiling data for the reference RISC processor, namely a RISC running under 3.1 V, is given in the table. This is because for the other RISC processors, the profiling data can be derived from the reference data based on $f \propto V_{dd}^3$, where f and V_{dd} denote the frequency and the V_{dd} of the processor, respectively.

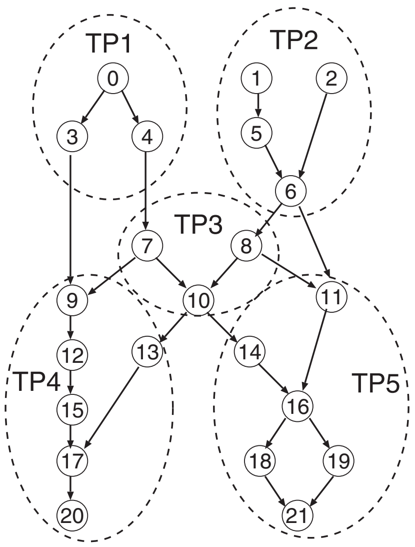


Fig. 5.3. Decomposed input thread frame

The first scheduling experiment has treated the TF as a flattened graph and has used the basic TCM design-time scheduler from the previous chapter which can explore the energy-performance trade-off space for a single TF. As a comparison, the second experiment has used the hierarchical scheduling approach. That is, it first decomposed the input TF into a set of thread partitions (see Fig. 5.3); then it scheduled threads inside each thread partition using the conventional TCM scheduler; at the end, it interleaved the individual schedules generated from each thread partition. To respect the dependencies among thread partitions, only the thread partitions without dependencies are allowed to be interleaved. Therefore, TP1 can be interleaved with TP2, and also TP4 with TP5. Figure 5.4 illustrates the differences between flattened scheduling, conventional DNC, and the hierarchical scheduling with interleaving.

All the data are measured on a Linux PC running at 1.7 GHz. The design space exploration results of the two experiments are plotted in Fig. 5.5. It is clearly shown that the flattened exploration Pareto curve is very close to the hierarchical exploration Pareto curve. In fact, the flattened scheduling has only found a schedule 5% faster than the fastest schedule explored by hierarchical scheduling with interleaving. Although the two results are very close, hierarchical scheduling has reached this result in much fewer efforts. In our experiments, the flattened scheduling time (15.5 s) is about 30 times longer than the time of the hierarchical scheduling with interleaving (0.5 s).

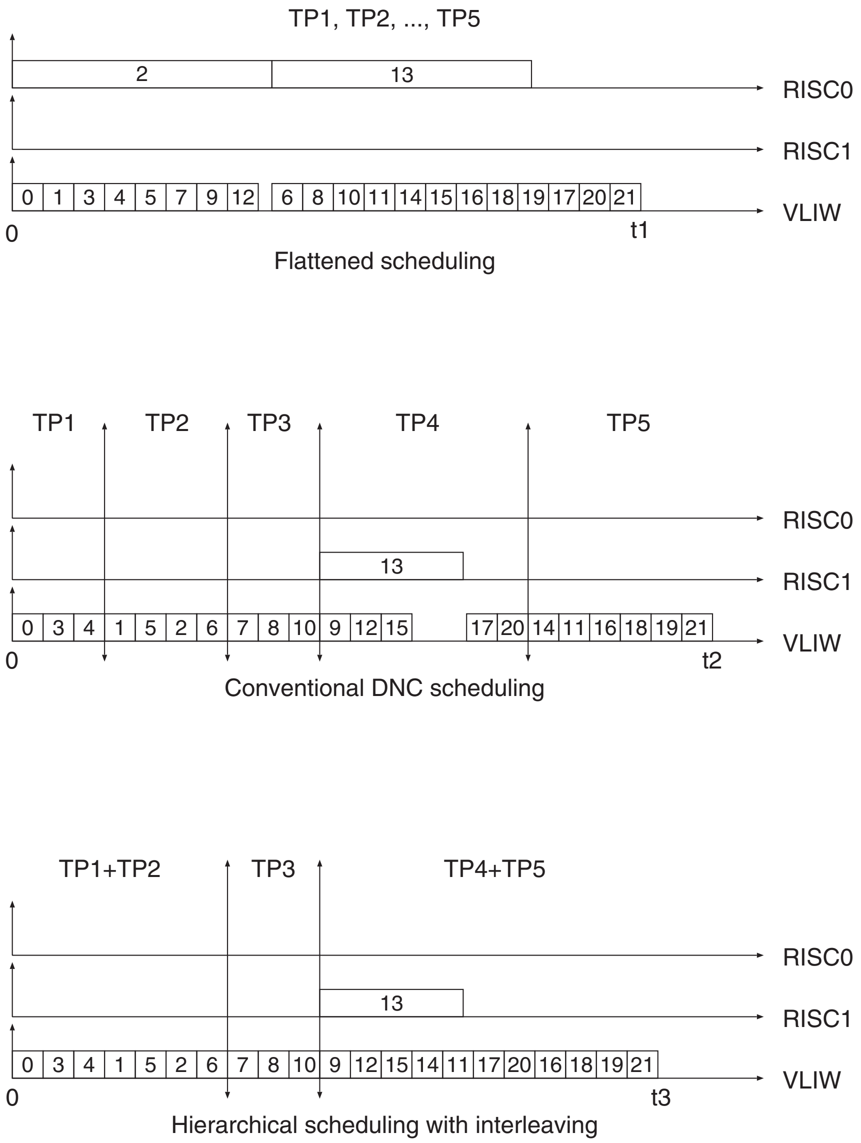


Fig. 5.4. Illustration of flattened and hierarchical scheduling

5.3 Thread Frame Decomposition

The TF decomposition step can create multiple thread partitions from the input TF in order to break down the scheduling efforts. In this section, we first provide a formal description of the problem. Then we discuss what properties of the TF are influential on making a decomposition decision. We finish with an effective TF decomposition algorithm.

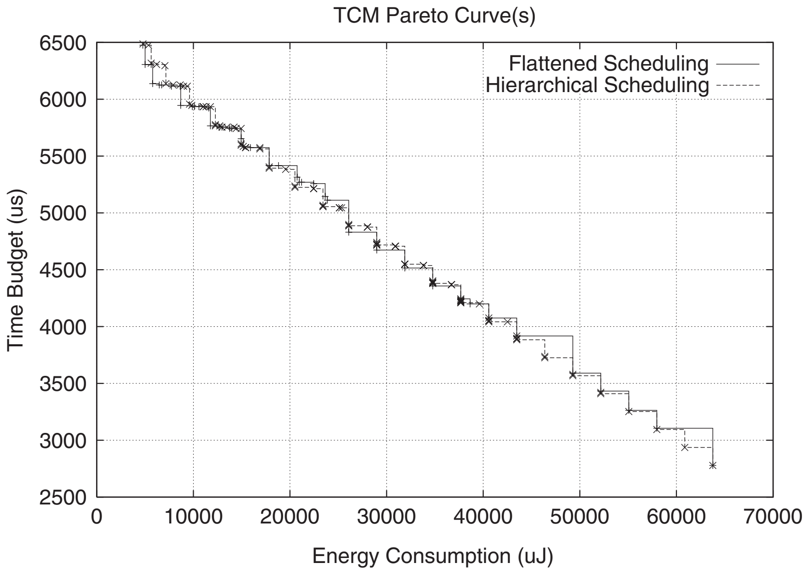


Fig. 5.5. Design space exploration results comparison

5.3.1 Problem Formulation

We consider an input TF as a Directed Acyclic Graph (DAG) $T(V, E)$ where the vertices $V = \{v_0, v_1, ...\}$ represent the set of threads and the edges E represents the control- and data-dependencies among threads.

Before the formal definition of the TF decomposition, we need three auxiliary definitions.

Definition 5.2 (Graph Dependency) *For two thread frames $G_0(V_0, E_0)$ and $G_1(V_1, E_1)$, a Graph Dependency (G_0, G_1) is a constraint between thread frame $G_0(V_0, E_0)$ and thread frame $G_1(V_1, E_1))$ such that the activation of threads $\in V_1$ cannot start before the completion of all threads $\in V_0$.*

Definition 5.3 (Thread Frame Partition) *A Thread Frame Partition (TFP) is a function from a thread frame (V, E) to N sub-frames (referred to as* thread partitions*) $\{(V_0, E_0), (V_1, E_1), \ldots, (V_{N-1}, E_{N-1})\}$ and a set of Graph Dependencies GD, such that,*

1. $V_0 \cup V_1 \cdots = V$
2. $\forall V_i, V_j, V_i \cap V_j = \phi$
3. $\forall v_i, v_j \in V_k,$
   $(v_i, v_j) \in E \Rightarrow (v_i, v_j) \in E_k$
4. $\forall (v_i, v_j) \in E, v_i \in E_k, v_j \in E_l,$
   $k \neq l \Rightarrow (G_k, G_l) \in GD$

Definition 5.4 (Complexity Estimator)

A Complexity Estimator (CE) is a function from a DAG (V, E) to a real number R. The function value of input DAG (V, E) is an estimation of the scheduling time of (V, E).

A TF decomposition problem is formally defined as follows,

Definition 5.5 (Thread Frame Decomposition Problem) *A Thread Frame Decomposition Problem is a Thread Frame Partition on the input thread frame (V, E), such that: $\forall(V_i, E_i), CE((V_i, E_i)) \leq Threshold$, where $Threshold$ is a pre-determined value set by the designer.*

5.3.2 Decomposition Guidelines

While the basic intention of TF decomposition is to keep the scheduling complexity under control, we must also consider how to provide larger room of freedom for the later step of merging. Therefore, two important concerns exist regarding the decision-making of decomposition. First, how to estimate the scheduling time of a certain thread partition and thus ensure that the resulting subgraphs would not lead to a heavy computation effort. Second, how to establish the inter-thread partition dependencies such that those dependencies will give fewer constraints to utilize the parallel processors. To address these two concerns, we present two guidelines, namely the horizontal decomposition and the look-ahead decomposition, on the appropriate decompositions which lead to fast scheduling as well as better scheduling results.

Horizontal Decomposition

Most DNC algorithms only relate the problem's complexity to the size of the problem. The time to solve scheduling problems, however, does not merely depend on the size of the problem. In fact, the structure of a thread partition can influence the scheduling time significantly. Hence, in addition to the sizes of thread partitions, we also consider the parallelism inside each thread partition.

We define the maximum number of parallel threads in a thread partition as its width, i.e., if a graph is traversed by a breadth-first search, the maximum number of vertices that are traversed within one search step is called the width of this graph. For example, the graph depicted in Fig. 5.2 has a width of 4 (node 11, 13, 14, and 15).

In general, the more parallel threads a thread partition has, the longer the design-time scheduling will be. This is because that parallel threads have no ordering constraints between them, and thereby lead to a larger exploration space when allocating and ordering the threads onto heterogeneous processors. For instance, when scheduling two parallel threads onto two different processors, one has six possible schedules. On contrast, scheduling two sequential threads on the same two processors only has four possible schedules.

Therefore, we use the maximum width of a thread partition to predict its scheduling time during the TF decomposition. That is, we first decide a maximum width threshold value based on experiments, and then ensure that each thread partition has a maximum width less or equal to the maximum width threshold.

As the result of the maximum width control, our decomposition scheme tends to split a wide TF into a number of horizontal (parallel) thread partitions. Although this horizontal decomposition seems to bring a penalty in the sense that less concurrency would be available when conducting the exploration for each partition, the interleaving technique that we are to present later would effectively exploit the concurrency across the boundaries of partitions.

Example 5.6 Consider a scheduling experiment on a large set of random TF. We decompose the TF using two options. The first option, OPT1, let us decompose the input TF into thread partitions with the maximum thread number of 10 and the maximum width of 5; the second option, OPT2, allows the same maximum thread number, but increases the maximum width to 10. We conducted design-time scheduling on these thread partitions and measure the scheduling time of each thread partition. This experiment employed random input TF with different sizes, namely 50 threads, 75 threads, and 100 threads.

The average scheduling times for TF with different options shown in Fig. 5.6.

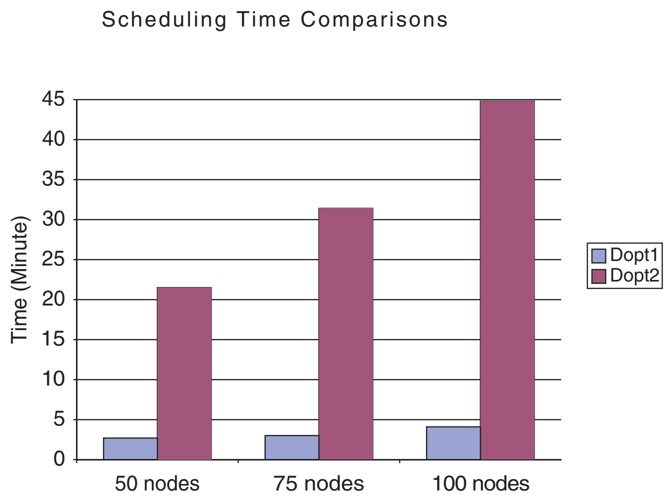


Fig. 5.6. Scheduling times comparison

The figure clearly shows that not only the number of threads in a TF can affect the scheduling time, the different widths also lead to significantly different scheduling times. This observation indicates that reducing the width of a thread frame would result in a faster scheduling process.

Dependency-Aware Decomposition

In addition to decomposing the original TF to thread partitions that have lower scheduling costs, we need to consider the scheduling quality after decomposition. In other words, we want to have the scheduling results of the decomposed TF as good as the ones that we can obtain by scheduling the original TF in the flattened way. In order to explore more parallelism from the original TF, we have developed an interleaving technique that can exploit the parallelism from different thread partitions (Section 5.5 will give the explanations of interleaving technique).

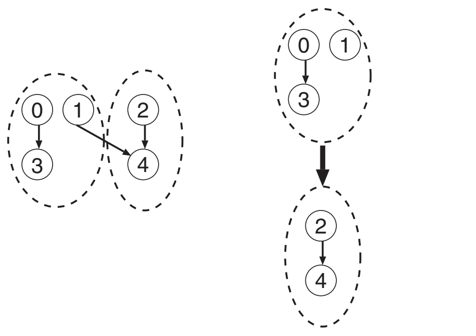
The interleaving technique can extract the parallelism from different thread partitions at a low computation cost. However, it can only be applied to a set of concurrent thread partitions. That is, if two thread partitions have a dependency in-between, they will not be able to be interleaved. Therefore, we must create thread partitions as concurrent as possible to boost the final scheduling quality.

From previous discussion about the scheduling time estimation, we can derive the maximum width a thread partition can have. This does not mean that we should partition the input TF in a greedy way, i.e., decomposing into thread partitions as wide as the maximum width allows. Because partitioning as many threads as possible to a thread partition could lead to additional inter-thread partition dependencies.

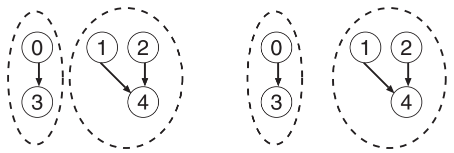
Example 5.7 Suppose we have a maximum width of 2 for decomposing an input TF illustrated on the left side of Fig. 5.7(a), a greedy decomposition is to split the TF in the way shown on the right side of Fig. 5.7(a). This straightforward decomposition however leads to an inter-thread partition dependency between the two thread partitions. This dependency is an additional constraint that postpones the start time of thread 2 to the finish time of thread 3, even though there is no such dependency in the original TF.

Instead of the greedy decomposition, we could check if any two threads have a common immediate successor by looking one step ahead. That is, we try to group the threads who share a same immediate successor in the same thread partition, as long as the the parallelism inside the thread partition does not go beyond the maximum width. In this way, we have a decomposition result illustrated in Fig. 5.7(b). Now that the two thread partitions are in parallel, the later interleaving step can further extract the thread concurrency across the thread partition's boundaries.

To reduce the inter-thread partition dependencies, we have proposed the a look-ahead dependency-aware decomposition (LADAD). The basic



(a) Without Look-ahead and the sequential result



(b) With Look-ahead and the parallel result

Fig. 5.7. Non-lookahead vs lookahead decomposition

idea of LADAD is that if there are N candidate threads to be decomposed ($N >$ Maximum Width), we do not treat those candidate threads as unrelated ones. Instead, we look one step ahead on each candidate thread by checking if any two candidate threads share a common successor thread. If two candidate threads share a common successor, we call them *relative threads*. In doing so, we actually create an undirected graph called look-ahead graph (LAG), as formally defined below.

Definition 5.8 (Look-ahead Graph) *A Look-ahead Graph is an undirected graph $LAG(V, E)$, where,*

1. $V = \{v_0, v_1, \dots\}$ *is the set of all candidate threads*
2. $\forall v_i, v_j \in V$,
   $(v_i, v_j) \in E \Longleftrightarrow v_i, v_j$ *are relative threads*

An established LAG may have a number of disjointed subgraphs. Each subgraph consists of the candidate threads that share common successor threads. For each disjointed subgraph of a LAG, we can now apply the greedy partition. Because candidate threads that share common successors stay in the same subgraph, greedy partition is much less likely to allocate them to different thread partitions (the partition heuristics presented in the next subsection can further reduce the chance that relative threads are allocated to different thread partitions).

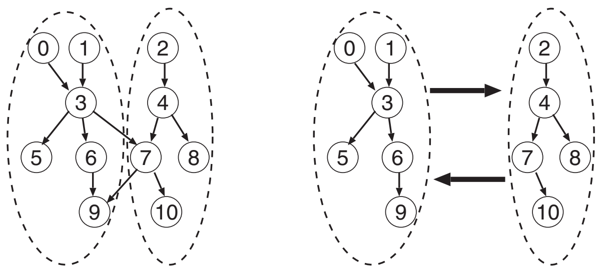
5.3.3 Valid Decomposition

During the TF decomposition, a set of graph dependencies are created among the thread partitions. These graph dependencies are additional constraints over the original control dependencies specified by the edges of those thread partitions. According to the Definition 5.2, a graph dependency between the thread frame $TG1$ and $TG2$ constraints the activation of all threads inside $TG2$ to the completion of $TG1$. The graph dependencies between thread partitions are built up by following Definition 5.3. After the TF decomposition, each thread partition is regarded as an independent unit by the scheduler, the graph dependencies are crucial to preserve the control dependencies in the original TF.

If we model the decomposed N thread partitions with a graph $G(V, E)$, where $V = \{v_i \mid 1 \leq i \leq N\}$ denotes all thread partitions and $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ denotes all graph dependencies, it is obvious that a valid G must be a DAG. However, a straightforward TF decomposition by only following the decomposition guidelines discussed so far could lead to an invalid non-DAG for the thread partitions.

Example 5.9 Consider a TF decomposition with maximum thread partition width of 2 on the input TF depicted in Fig. 5.8(a). Because of the look-ahead guided decomposition, thread 0 and thread 1 are partitioned to the same thread partition, while thread 2 starts a new thread partition. Then the initial thread partitions are expanded with the constraint that the maximum TF width is less or equal to 2. Due to the thread dependency between thread 3 and thread 7, a graph dependency is created from the thread partition 1 to the thread partition 2; while the thread dependency between thread 7 and thread 9 leads to an opposite graph dependency. The two thread partitions are then in a deadlock as illustrated in Fig. 5.8(b). The TF decomposition is then an invalid decomposition.

The invalid decomposition is caused by the fact that the original TF is not a tree and hence a thread may have more than one predecessor. As illustrated in Fig. 5.8(a), thread 9 has two predecessors, since one of its predecessor,



(a) Thread Frame Decomposing (b) Graph Dependency Deadlock

Fig. 5.8. Invalid thread frame decomposition

namely thread 7, has been decomposed to another thread partition. As a consequence, the left-side thread partition has an inbound graph dependency from the right-side thread partition; unfortunately, this situation also takes place on thread 7, which leads to a graph dependency from left side to right side and hence causes the deadlock condition.

It is hard to detect the deadlock graph dependencies during an arbitrary TF decomposing. However, we can effectively avoid such deadlocks by controlling the inbound graph dependencies. Before controlling the dependencies, we label all threads of the original TF with generation numbers. A thread's generation number is 0, if it has no predecessor; Otherwise, its generation number is the maximum one of all predecessors' generation numbers plus 1. For a thread partition, we call the set of threads with the smallest generation number as the *entry threads* (note that the entry threads may have predecessors outside the thread partition). If we let that all inbound graph dependencies only occur at the entry threads, we can then guarantee that the decomposition on the original thread frame does not have graph dependencies in deadlock.

Proposition 5.10 *If a set of thread partitions only have inbound graph dependencies at each thread partition's entry thread(s), then there are no graph dependencies in deadlock.*

Proof Suppose a chain of graph dependencies $(s_0, s_1), ...(s_N, s_0)$ is in deadlock, and all inbound graph dependencies only occur at the entry threads. Let us refer the threads where the graph dependencies start as outbound threads, and denote outbound threads on that graph dependency chain as $t_0, t_1, ...t_N$, where t_i stays in the partition s_i. The outbound threads' generation numbers are $g_0, g_1, ...g_N$, respectively. Because a graph dependency only occurs between an outbound thread and an entry thread of another partition, for any graph dependency (s_i, s_j), the generation number g_i of the outbound thread t_i is smaller than that of the entry thread of the partition s_j, and thus smaller than the generation number of any thread in the partition s_j. Hence, it is clear that $g0 < g1 < ... < gN$. However, $g0 < gN$ contradicts the existence of $(sN, s0)$. Therefore, such a deadlock chain cannot exist.

In order to ensure a valid decomposition on the original TF, we have developed a two-step decomposition method. By following this two-step method, the decomposition process can guarantee that all inbound graph dependencies only occur at the beginning of a thread partition and thus effectively avoid the invalid decomposition.

The first step is called thread partition initialization. It can create thread partitions from the set of threads with the same generation numbers, i.e., it creates thread partitions in a single generation layer.

The second step is called thread partition expansion. This step tries to expand the initial thread partition generation by generation, until either no further successors are available, or all successors have at least one predecessor not partitioned to the current thread partition.

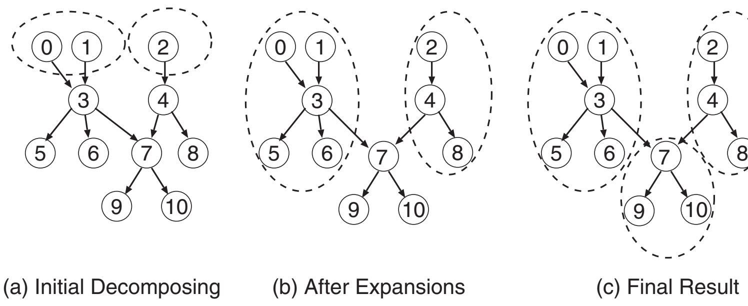


Fig. 5.9. Two-step thread frame decomposition

Example 5.11 Consider applying the two-step decomposition method on the problem given in Example 5.9. The first step can create two initial thread partitions, as illustrated in Fig. 5.9(a). These two thread partitions are then expanded in the second step, generating the intermediate result depicted in Fig. 5.9(b). Since unpartitioned threads, i.e., thread 7, 9 and 10, are still present in the original TF, we repeat the first step of our decomposition method and then carry out the second step. The final result of the decomposition is shown in Fig. 5.9(c). It is clear that there is no graph dependency in deadlock for the final result.

5.3.4 Thread Frame Decomposition Algorithm

The whole decomposition process is listed in Algorithm 1. The topological sorting at line 4 labels each thread with an unique generation number. Then the first set of initial thread partitions is created from threads at generation 0 with consideration of dependency-aware look-ahead mechanism (line 5). Each initial thread partition is expanded as large as possible under the constraint of maximum width (line 7). If any threads in the original TF are not partitioned after the expansions, new initial thread partitions are created from the earliest generation level (line 8).

Algorithm 1 Thread Frame Decomposition

```
1: INPUT: Original Thread Frame
2: OUTPUT: Thread Partitions, Partition Dependencies
3: Initialize thread frame data-structures
4: Sort of all threads topologically
5: Create initial thread partitions from generation 0
6: while Input thread frame is not fully decomposed do
7:     Expand each initial thread partition
8:     Initialize new thread partitions from undecomposed part of the input thread
       frame
9: end while
```

5.4 Thread Partition Clustering

After the decomposition, the input TF is broken into multiple thread partitions which are then passed to the TCM design-time scheduler. The design-time schedules of each thread partition are explored by the TCM design-time scheduler independently from other thread partitions. Therefore, this design-time schedules exploration can be easily executed on different computers and thus dramatically speed up the overall scheduling time.

The TCM design-time schedulers can generate the Pareto-optimal schedules on the performance-energy trade-off space for each thread partition. All Pareto-optimal schedules of a thread partition is referred as its Pareto-curve.

Traditional hierarchical thread scheduling techniques often stop after decomposing the input TF and schedule each thread partition individually. In contrast, our hierarchical scheduling approach moves one step forward by exploiting the parallelism among thread partitions, i.e., we will generate a more parallel global schedule than just run over the schedules of all thread partitions in a row. An interleaving phase is necessary to achieve that parallelism. Before the interleaving phase, we must first cluster the thread partitions.

The thread partition clustering mainly consists of two components:

1. Identifying the thread partitions clusters such that thread partitions within a cluster can be run in parallel.
2. Generating a new Pareto curve for each thread partition cluster based on merging and pruning the Pareto curves of all thread partitions.

5.4.1 Identifying Thread Partition Clusters

The output from the decomposition process include a set of thread partitions and the corresponding graph dependencies. Since we are going to extract the parallelism across the thread partitions' boundaries, it is necessary to assign thread partitions to different clusters such that each cluster only has thread partitions in parallel. Then we can interleave all thread partitions in a cluster.

If we construct a graph $G(V, E)$, where $V = \{v_i \mid 1 \leq i \leq N\}$ denotes all thread partitions and $E = \{(v_i, v_j) \mid v_i, v_j \in V\}$ denotes all graph dependencies, G must be a DAG according to Proposition 5.10. Thus it is a straightforward step to label the cluster numbers to each partition by a topological sorting on G, as illustrated in Algorithm 2.

5.4.2 Generating New Pareto Curves

After the thread partition clustering, we have a sequence of clusters. Each cluster has one or more thread partitions inside. If a cluster contains more than one thread partition, we need to prune the Pareto-optimal schedules from each individual thread partitions. Because each thread partition has a

Algorithm 2 Thread Partition Clustering

```
1: INPUT: Thread Partition Graph $G(V, E)$
2: OUTPUT: $Cluster(C_0, C_1, ...C_N)$
3: $L \leftarrow 0$
4: $C_0 = C_1 = ... = C_N = \phi$
5: while $V \neq \phi$ do
6:     $C_L \leftarrow$ all $v \in V$ without predecessors
7:     $E \leftarrow E - \{all\ edges\ that\ starts\ from\ v \in C\}$
8:     $Cluster \leftarrow Cluster + \{C_L\}$
9:     $L \leftarrow L + 1$
10: end while
```

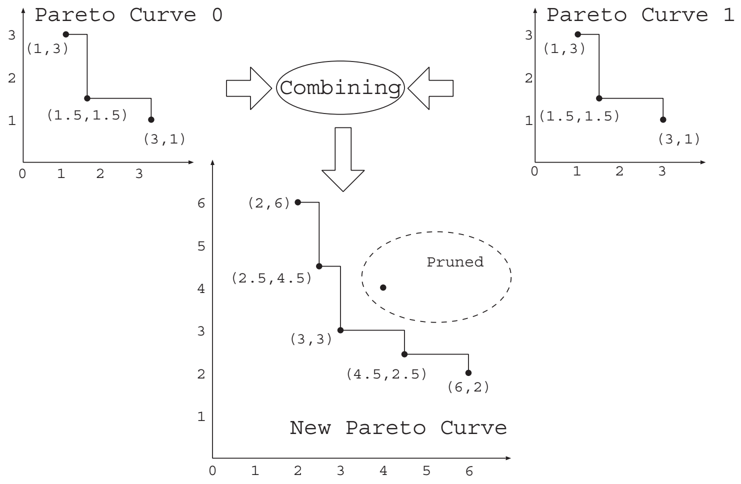


Fig. 5.10. Pareto curve pruning

number of Pareto-optimal schedules, the total number of combinations of Pareto-optimal schedules would be too large to examine. In fact, many combinations are not Pareto-optimal and need to be pruned. Therefore, we need to build a new Pareto curve for all thread partitions in this cluster. This pruning is illustrated on Fig. 5.10, note that the energy figure of each combination is the summation of energy figures of each individual thread partitions, and the time figure is the summation of all individual time figures.

5.5 Thread Partition Interleaving

5.5.1 Motivation

Each thread partition cluster established in the clustering step has a pruned Pareto curve consisting of a sequence of Pareto-optimal schedules. Those Pareto-optimal schedules represent the sequential combinations of

Pareto-optimal schedules of individual thread partitions within this cluster. The sequential combination implies that although the thread partitions do not have dependencies within a cluster, they have to be run in a sequential manner. In the meantime, an interesting observation of Pareto-optimal schedules of individual thread partitions is that they have significant slacks inside. This is mainly due to two reasons: (a) when decomposing the original thread frame, we have limited thread parallelism within a thread partition to reduce the scheduling time. This limited parallelism leads to insufficient utilization of the processors, and (b) different execution times and energy consumptions caused by running the same thread on heterogeneous processors let the power-optimizing design-time scheduling strategies under-utilize the parallelism in favor of the energy efficiency.

Because of the slacks inside the schedules, the sequential execution is not able to fully utilize the available parallelism of the underlying hardware and hence provides an inferior overall Pareto curve for the whole TF. In order to generate better Pareto curves, we propose to apply a new technique to exploit the inter-thread-graph concurrency, namely the INTERLEAVING.

5.5.2 Preliminaries

The basic idea of interleaving is to build a new global schedule based on shifting the Pareto-optimal schedules of individual thread partitions. During this shift, both the processor allocation of a thread and the sequence of threads that belong to one thread partition are kept unchanged. For a clear problem definition, we provide the problem formulation below.

In our problem formulation, a thread partition's schedule S_k for c processors is a list $(M_1, ..., M_c)$, where $M_i = ((b_j, e_j), ...)$ is the list of threads scheduled on the ith processor and (b_j, e_j) denotes that the first thread on this processor is s_j with the start time of b_j and the execution time of e_j. For k schedules $(S_1, ..., S_k)$ of the thread partitions $((V_1, E_1), ..., (V_k, E_k))$ on a platform with c processors, the interleaving problem can be formally expressed as:

$$\forall s_i \in \left(\bigcup_{q=1}^{k} V_q\right), \qquad \text{Minimize } [(x_i + e_i)_{\max}]$$

such that:

1. $\forall e(s_i, s_j, t) \in \{E_1, E_2, ..., E_k\} \qquad x_i + e_i + t \le x_j;$
2. $\forall a \in \{1, 2, ...c\}\ \forall (x_i, e_i), (x_j, e_j) \in M_{1a} \cup ... \cup M_{ka}$
   $(x_i < x_j \Rightarrow x_i + e_i \le x_j \ \wedge\ x_i > x_j \Rightarrow x_j + e_j \le x_i).$

5.5.3 Interleaving Technique

Scheduling threads with nonuniform execution times on multiple processors is well known for its intractability [76]. In fact, Hoogeveen et al. [94]

have proved that even for three processors, scheduling threads with fixed processor allocations is a NP-hard problem. Our interleaving problem is more restrictive than the thread scheduling problem with fixed processor allocations in the sense that interleaving has to consider the precedence constraints. The NP-hard nature of interleaving problem makes it a difficult problem to solve. Still, for not too many threads, an exact algorithm can be applied. We have developed a branch-and-bound algorithm for the interleaving problem (Algorithm 3). This algorithm starts with an initial state S formed by all threads' starting times. All starting times are not decided and thus the partial global schedule is empty. The algorithm first selects threads without predecessors or with predecessors whose starting times are already decided, and then decides the starting times for these threads. Each time when the starting time of a thread is decided, this thread is inserted to the partial global schedule. The algorithm checks if the partial global schedule is valid after each insertion. A partial schedule is valid only if no precedence constraints are violated in all the threads that have been inserted. Once a thread's starting time is decided, its successors become candidates for further thread selection to grow the partial global schedule. This growing is continued until the global schedule is completed. A partial global schedule is completed once all threads are inserted. For each completed global schedule, the algorithm measures its *makespan*, i.e., the difference between the starting time of the first thread and the finishing time of the last thread. If the makespan is shorter than the shortest one from all explored completed global schedule, this completed global schedule is then recorded. After that, the algorithm does backtracking in an attempt to explore better schedules. The searching stops when all possible schedules are explored.

Algorithm 3 Branch-and-bound algorithm for interleaving

```
1: BnB()
2: INPUT: $status$; $upper\_bound$
3: OUTPUT: makespan
4: if $makespan\ of\ status > upper\_bound$ then
5:    return $makespan\ of\ status$
6: end if
7: if all threads are scheduled then
8:    print $status$
9:    return $makespan\ of\ status$
10: else
11:    $schedulable\_threads \leftarrow precedence\_free\ threads$
12:    for all thread $i$ in $schedulable\_threads$ do
13:       $new\_status \leftarrow status$
14:       schedule thread $i$ and update $new\_status$
15:       $makespan \leftarrow \mathbf{BnB}(\mathbf{new\_status},\ \mathbf{upper\_bound})$
16:       if $makespan < upper\_bound$ then
17:          $upper\_bound \leftarrow makespan$
18:       end if
19:    end for
20:    return $upper\_bound$
21: end if
```

It is still interesting though to have fast algorithms that can handle more threads. Therefore an effective heuristic algorithm has been developed to interleave multiple TF. This heuristic must be fast to construct a valid schedule so that the designer can evaluate multiple schedules which have been provided by preceding individual thread scheduler.

We have developed a fast interleaving heuristic based on the list scheduling algorithm [98]. This heuristic uses the first-come-first-served principle to keep a list for each processor and from all given schedules, allocate each thread to the list of the processor where it is mapped in its own thread partition's schedule. All threads allocated to a list are sorted according to their starting times (ST) in the original schedules. An earlier thread in the original schedule is put closer to the top in the list than a thread with a later starting time. For each processor, the algorithm then scans the list from top to bottom. Once a scanned thread has all of its predecessors completed, it will be added to the ready list and scheduled onto the current processor. To alleviate the greedy behavior of this heuristic, we have also adapted the threads' order in the ready list using a look-ahead mechanism, i.e., we modify the order such that a thread is put to an earlier position if it has successors with larger accumulated execution times. Please note that the successors include both immediate successors and the successors of immediate successors. The entire heuristic algorithm is presented in Algorithm 4. Once the interleaved schedule is generated, we can use it to steer the code generation by using the code merging technique presented by Marchal et al. [148]. The resulting code can then be executed on the multiprocessor platform.

Algorithm 4 Interleaving heuristic

```
1: INPUT: Schedules of N partitions: $S_1, ...S_N$
2: OUTPUT: $Global\_Schedule$
3: $timer \leftarrow 0$
4: $unsched\_threads \leftarrow$ threads from all partitions
5: while $unsched\_threads > 0$ do
6:   for all processor $i$ do
7:     for all schedule $S_j$ do
8:       if $S_j$ has threads on processor $i$ then
9:         add the threads to the ready list on the processor $i$
10:      end if
11:    end for
12:    for all threads on the ready list $i$ do
13:      $priority \leftarrow ST + accu.\ exec.\ time\ of\ successors$
14:    end for
15:    $ST \leftarrow ST$ of the highest priority thread
16:    if $timer < ST$ then
17:      $timer \leftarrow EST$
18:    end if
19:    schedule the top priority thread $T$ starting at $timer$
20:    $processor\_schedule_i \leftarrow processor\_schedule_i + T$
21:    inform this thread's start time to its successors
22:    $unsched\_threads \leftarrow unsched\_threads - 1$
23:  end for
24: end while
25: $Global\_Schedule \leftarrow \{processor\_schedule_1\} + ... + \{processor\_schedule_N\}$
```

5.6 Experimental Results and Discussions

In this section, we first give a brief introduction to the implemented hierarchical scheduler. Then we present the hierarchical scheduling experiments on a large set of random TF generated by TGFF [70].

5.6.1 Experimental Setup

We have implemented the whole hierarchical scheduler in three main components, namely the Thread Frame Decomposer (TFD), the Thread Partition Design-time Scheduler (TPDS), and the Thread Partition Merger (TPM). The TFD module is implemented in Python code, its main functionality is to parse the input description file of TF and generate description files for each resulting thread partition. The TPDS module is a Python wrapper around the executable of the existing design-time scheduler [245], which does a trade-off exploration for each thread partition. The TPM module is also implemented in Python code. It can parse the output files of TPDS and identify thread partition clusters. For each thread partition cluster, the TPM then invokes the executable of our interleaver (implemented in C code) to generate a Pareto curve. The TPM then merges all Pareto curves into a global Pareto curve.

5.6.2 Experiments with Random Thread Frames

To evaluate the effectiveness of the hierarchical scheduling, we need a large set of random TF. We have used a software tool called TGFF [70] to generate the TF for evaluations. TGFF can generate random TF according to the specified options such as the thread number. In addition to generating the random TF, TGFF can be configured to also generate a random configuration for a multiprocessor platform. A sample option file for TGFF is illustrated in Fig. 5.11. This option file can make TGFF generate a TF with 50 thread nodes as well as a platform with 6 processors running at random working voltages, as illustrated in Figs. 5.12 and 5.13, respectively.

In order to measure the optimality of Pareto curves, we first need to calculate the lower bounds. The lower bound of energy consumption for scheduling a TF on a given platform can be calculated by allocating each thread to the processor with minimal energy consumption and then the sum of all threads' energy numbers is the lower bound. The lower bound of execution time is calculated by allocating threads to their fastest processors and then the sum of all individual execution times is divided by the number of processors on the given platform. The result is the lower bound of execution time. Please note that this lower bound of time may not be reached by any feasible schedule at all. But it is clear that no feasible schedule could have an execution time shorter than the lower bound.

The optimality measurement of the scheduling results is then carried out by applying the metric of Pareto-optimality. That is, we calculate the

```
seed 10
################
#thread graph
################

tg_cnt 1 #nr of thread frames

thread_cnt 50 1 #base nr of threads in a graph

period_mul 1 #multiple of base nr of threads

################
#processor speed
###############

thread_type_cnt 30

table_cnt 6 #processor nr

tg_write
pe_write
###############
#processor power
###############
table_label Power
thread_type_cnt 6
table_cnt 1
type_attrib working_power 20 10

pe_write
eps_write
```

Fig. 5.11. Sample TGFF option file

time difference between a result's length and the lower bound of time consumption as well as the energy difference between a result's energy consumption and the lower bound of the energy consumption. The product of a result's time difference and its energy difference is used to measure its Pareto optimality. The optimality of a Pareto curve is then measured by the mean value of all Pareto points' products (as illustrated in Fig. 5.14).

A large number of random TF are generated for three categories, the first category has 50 TN in each TF, the TN numbers of the second and third categories are 75 and 100. Thread frames from each category are decomposed with the maximum partition width of 5 and with the maximum thread number of 10. We have conducted scheduling experiments for platforms with 6 and 8 processors, respectively. The design-time scheduler of [244] is used as

```
@TASK_GRAPH 0 {
    TASK t0_0   TYPE 22
    ......

    ARC a0_0   FROM t0_0   TO t0_1   TYPE 8
    ......
}
@PE 0 {
# type version exec_time
  0    0       104
  ......
  29   0       86
}
@PE 1 {
# type version exec_time
  0    0       108
  ......
  29   0       91
}
  ......
  ......
  ......
@PE 5 {
# type version exec_time
  0    0       113
  ......
  29   0       127
}
@Power 0 {
# type version working_power
  0    0       25
  ......
  5    0       20
}
```

Fig. 5.12. TGFF output file

the reference flattened scheduler for the comparisons. The results are listed in Table 5.15; note that the lower value in the metric of Pareto-optimality represents a better Pareto curve in the sense that it gives faster schedules at lower-energy consumptions.

5.7 Comparison with State of the Art

Scheduling a DAG on multiprocessor platforms with a minimum makespan is notorious for its intractability. The scheduling time increases dramatically with large DAGs, even when using heuristic algorithms. This problem

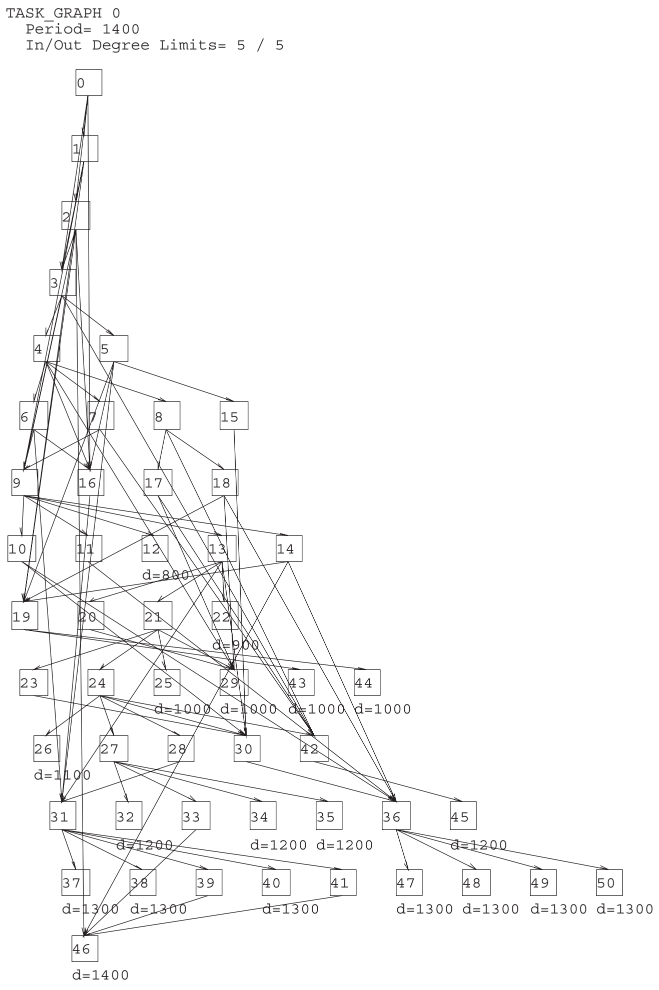


Fig. 5.13. TGFF output thread frame

becomes even worse when the designers have to consider other scheduling criteria such as energy consumptions, in addition to the schedules' lengths. A natural way to speedup the scheduling process is to split up the original DAG and schedule each sub-DAG in parallel.

The general graph decomposition problem using the divide-and-conquer strategy has been investigated for many years, most of the publications, such as [74], have been aimed at solving general graph decomposition problems without considering the constraints introduced by the scheduling process after the decomposition.

Recently, Ahmad and Kwok [2] has combined the decomposition problem and the scheduling problem within an unified hierarchical scheduling

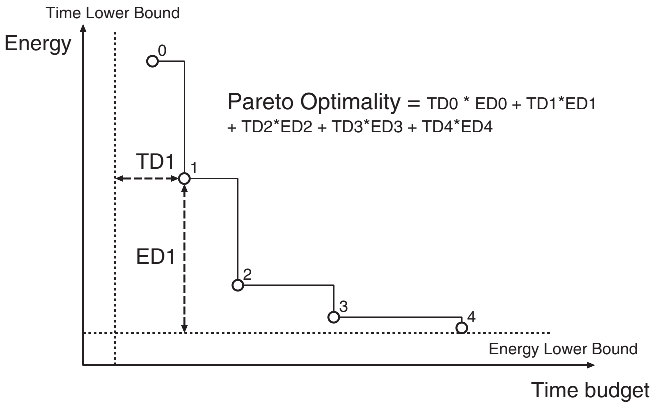

Time Lower Bound
Energy
0
Pareto Optimality = TD0 * ED0 + TD1*ED1 + TD2*ED2 + TD3*ED3 + TD4*ED4
TD1
1
ED1
2
3
4
Energy Lower Bound
Time budget

Fig. 5.14. Pareto optimality metric

		6-processor		8-processor	
		optimality	sched. time	optimality	sched. time
50	Flat	13.9×10^6	$1000\ s$	14.3×10^6	$1000\ s$
threads	Hier	7.5×10^6	$29\ s$	7.7×10^6	$30\ s$
75	Flat	13.3×10^6	$2000\ s$	15×10^6	$2000\ s$
threads	Hier	2.8×10^6	$30\ s$	3.0×10^6	$35\ s$
100	Flat	36×10^6	$4000\ s$	32×10^6	$6000\ s$
threads	Hier	10×10^6	$35\ s$	9×10^6	$37\ s$

Fig. 5.15. Result Pareto optimality comparison: flattened scheduling vs hierarchical scheduling

flow. However, their work has only considered the performance of resulting schedules and hence severely reduced the scheduling exploration space for each subgraph.

General purpose evolutionary algorithms have been widely studied for the multiobjective optimization problems (see [63] for a good survey). They have recently been adapted to in the embedded software synthesis methodologies [68, 199]. These evolutionary algorithms are distinguished from previous scheduling algorithms by their capabilities to explore the trade-off space of the multiobjective optimization, which is an important problem encountered when designing modern embedded systems. The evolutionary algorithms, however, are designed for general-purpose problem-solving and thus inefficient for task scheduling problems. Despite their extremely long scheduling times, they are not robust in terms of optimality. Because they choose starting points randomly, and a bad starting point can dramatically reduce the result's quality.

In contrast, the design-time performance-energy trade-off exploration algorithm presented by Wong [245] was designed specifically for the thread scheduling problem on multiprocessor platforms and therefore is much effective. Because of the intractable nature of scheduling problems, even the heuristic method of this specific exploration algorithm may suffer a lengthy execution time, which makes it unsuitable to work on large TF.

5.8 Summary

This chapter has presented a hierarchical scheduling approach based on the interleaving technique. This hierarchical scheduling approach can deal with large TF efficiently. As a result, the speedup of up to 2 orders of magnitude has been achieved for large TF. Moreover, the hierarchical scheduling approach can speedup the scheduling process in a scalable way by creating concurrent scheduling jobs, where each job can be performed independently on an individual processor. The interleaving technique ensures that the final results from the hierarchical scheduler can reach the lower bound of energy consumption. The hierarchical scheduler does suffer a makespan penalty when compared to a flattened scheduling. Nevertheless, the average overall Pareto optimality of the scheduling results from our hierarchical scheduler is 50% better than the results from the conventional flattened scheduler due to its faster exploration.