Scalable Performance-Energy Trade-off Exploration of Embedded Real-Time Systems on Multiprocessor Platforms

Zhe Ma*, Francky Catthoor†
IMEC
Kapeldreef 75, B-3000 Leuven, Belgium
{mazhe, catthoor}@imec.be

Abstract

Conventional task scheduling on real-time systems with multiple processors is notorious for its computational intractability. This problem becomes even harder when designers also have to consider other constraints such as energy consumptions. Such a multi-objective trade-off exploration is a crucial step to generating cost-efficient real-time embedded systems. Although previous task schedulers have attempted to provide fast heuristics for design space exploration, they cannot handle large systems efficiently. As today's embedded systems become increasingly larger, we need a scalable scheduler to handle this complexity. This paper presents a hierarchical scheduler that combines the graph partition and the task interleaving to tackle the trade-off exploration problem in a scalable way. Our scheduler can employ the existing flattened scheduler and significantly accelerate the design space explorations for large tasks. The speed-up of up to 2 orders of magnitude has been obtained for large task models compared to the conventional flattened scheduler.

1. Introduction

Today's embedded software are complex and contain large portions of static software components such as the multimedia codecs. When mapping these components onto a multiprocessor platform, system designers need a good synthesis tool that can make trade-offs between multiple objectives such as performance and energy consumption. Because most scheduling problems encountered during system synthesis are quite difficult, conventional scheduling algorithms suffer lengthy runtime when scheduling large task models. The divide-and-conquer (DNC) strategy is normally employed on the task models in order to reduce the scheduling complexity. However, conventional DNC algorithms often partition the task model based on the number of tasks, which is not the only reason for the extremely long times of the task-level design space explorations. Moreover, conventional DNC approaches lack a post-DNC step which can partly compensate for the loss of scheduling optimality due to the division. This paper presents a novel approach to deal with very large task models by combining the graph partition method with the interleaving technique.

2. Preliminaries

This section first briefly describes our system model for the trade-off exploration. It then gives an overview of the scalable exploration flow.

An embedded application is represented by a gray-box model, which is essentially a two-level hierarchical task graph [9]. With this specification, an application is represented as a set of *thread frames* (TF) at the high-level. Each TF is a piece of code performing a specific function, which is then partitioned into *thread nodes* (simply referred to as *threads*) - the basic scheduling units at low-level. The purpose of our trade-off exploration is to determine a cost-optimal (e.g. energy consumption, deadline miss rate) thread scheduling on a set of heterogeneous processors. Heterogeneous processors normally execute the same thread at different speeds and with different energy consumptions. These differences form the design space of energy-performance trade-offs. A set of Pareto-optimal trade-off points can be identified by exploring this space. Our design space exploration is performed within each TF, i.e., we identify Pareto-optimal trade-offs inside each individual TF. Hence, this paper is focused on how to efficiently schedule a large TF at the low-level. Handling of multiple TFs at the high-level can be found in the work of [11].

We propose a hierarchical scheduling approach for the scalable trade-off exploration. This approach consists of five steps as illustrated in Fig. 1. Note that the Pareto curves indicated in the overview figure are actually sets of Pareto-

* Also Ph.D. student of ESAT, K.U.Leuven, Belgium
† Also professor of ESAT, K.U.Leuven, Belgium

3-9810801-0-6/DATE06 © 2006 EDAA

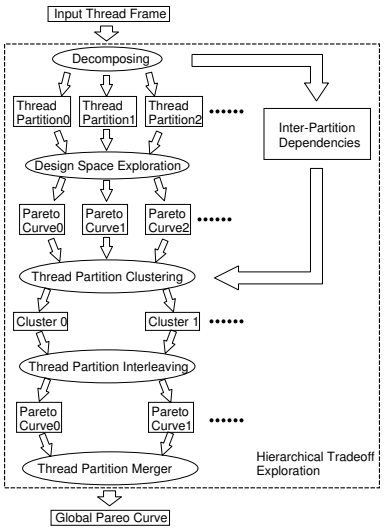

Figure 1. Hierarchical Scheduling Overview

optimal schedules of individual thread partitions (TP). The whole workflow is demonstrated in the following example:

Example 1

Consider a design-time exploration problem of mapping a given TF (shown in Fig. 2(a)) onto a three-processor platform consisting of one VLIW processor running at 1.56 V and two RISC processors running at 1.08 V and 1.62 V, respectively. For simplicity, all threads have the same amount of execution times and energy consumptions on each processor, i.e., all threads are the same:

	VLIW	RISC
Energy consumptionμJ	485.438	1040.55
Execution timeμs	177.816	1009.69

Note that although threads' execution times/energy are non-deterministic due to run-time dynamic behaviors, we can capture them by using different *working scenarios*: a thread has a worst case execution time/energy under a specific scenario. Only the profiling data for a RISC running under 3.1 V is given in the table. For the RISC processors running at different voltages, 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 working voltage of the processor, respectively.

The first scheduling experiment has treated the TF as a flattened graph and has used the design-time scheduler described in [10] which can explore the energy-performance trade-off space for a single thread frame. As a comparison, the second experiment has used the hierarchical scheduling approach. That is, it first decomposed the input TF into a set of TPs (see Fig. 2(b)); then it scheduled threads inside each TP using the conventional design-time scheduler; at the end, it interleaved the individual schedules generated from each TP. Due to the dependencies among TPs, only TP1 can be interleaved with TP2, and the same for TP4 with TP5. Fig. 3 uses three sample schedules to illustrate the differences between flattened scheduling, conventional DNC and the hierarchical scheduling with interleaving. Each block in this figure represents a thread from the Fig. 2. The conventional DNC schedule has

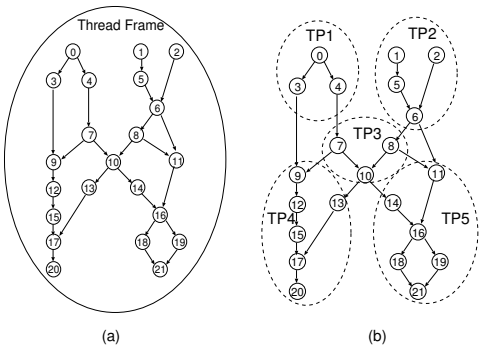

Figure 2. Input TF Example

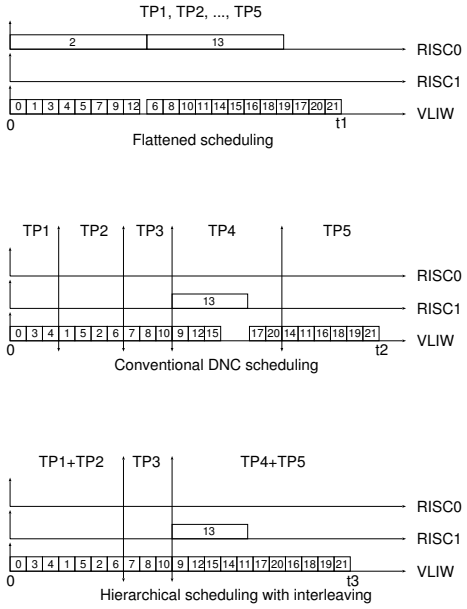

Figure 3. Illustration of different scheduling approaches

4 synchronization points, i.e., each TP must await until the previous TP is done, while the hierarchical schedule only has 2 synchronization points.

The design space exploration results of the two experiments are plotted on Fig. 4, the flattened exploration Pareto curve is very close to the hierarchical exploration Pareto curve. However, hierarchical scheduling has reached this result much faster: the exploration time of the flattened scheduling is about 15.5 seconds, while the hierarchical exploration only takes 0.5 second.

3. Thread Frame Decomposition

We consider an input TF as a Directed Acyclic Graph (DAG) $T(V, E)$ where the vertices $V = \{v_0, ...v_n\}$ repre-

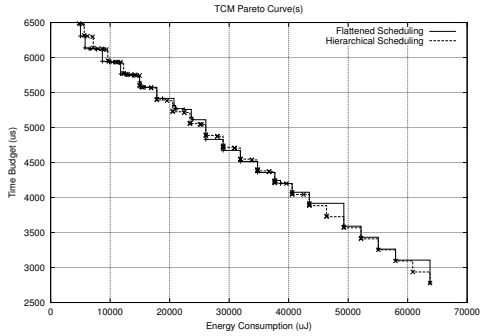

Figure 4. Design Space Exploration Results Comparison

sent the set of threads and the edges E represent the control- and data-dependencies among threads. The TF decomposition splits an input TF into multiple TPs in order to break down the scheduling efforts. Each TP is in fact a smaller TF. In this section, we first discuss what properties of the TFs have influences on making a decomposition decision. We then present an effective TF decomposition algorithm.

3.1. Decomposition Guidelines

Two major issues must be considered for decomposition. First, how to estimate the scheduling time of a certain TP and thus ensure that the resulting partitions would not lead to a long-time scheduling. Second, how to establish the inter-TP dependencies such that those dependencies will give less constraints to utilize the parallel processors. To address these two concerns, we present three decomposition guidelines for the hierarchical scheduler.

3.1.1. Horizontal Decomposition Most DNC algorithms in the task scheduling domain only relate the problem's complexity to the size of the problem. However, the parallelism of a TP can also influence the scheduling time significantly. Because parallel threads have no ordering constraints, a larger exploration space needs to be examined when allocating and ordering the threads onto heterogeneous processors. For instance, to put two sequential threads on two different processors one has 4 possible schedules; while with two parallel threads, one has 8 different schedules. Scheduling results based on extensive set of random TFs have shown that a TP with more parallel threads has longer design-time scheduling in general. We define the maximum number of parallel threads in a partition as its width, i.e., if the task graph of a partition is traversed by a breadth-first search, the maximum number of threads that are traversed within one search step is called the width of this partition. For example, the task graph depicted in Fig. 2(a) has a width of 4 (node 11, 13, 14 and 15). Then we can use the maximum width of a TP to control its scheduling complexity during the thread frame decomposition. That is, we decide a maximum width threshold value based on experiments, and ensure that each TP has a maximum width less or equal to the threshold. As a result of the maximum width control, our decomposition scheme tends to split a wide TF into horizontal (parallel) TPs. Although this horizontal decomposition seems to cause a optimality penalty in the sense that less concurrency would be available when conducting the exploration for each partition, the interleaving technique on the later stage of the hierarchical scheduler would effectively exploit the concurrency across the boundaries of partitions.

3.1.2. Look-ahead Decomposition In order to improve the results of the hierarchical scheduling, we have developed a partition interleaving technique that can exploit the parallelism from different TPs (See Section 5 for the explanations of interleaving). Because interleaving can only be applied to a set of parallel TPs, we need to construct parallel TPs to use the interleaving after decomposition.

Partitioning two dependent threads to different partitions can create unnecessary inter-partition dependencies (illustrated in Fig. 5) and hence lead to less parallel partitions. Therefore, we need to take the thread dependencies into account during the decomposition. One way to consider those dependencies is to put threads with the same successor thread in one partition. To do that, we developed the look-ahead decomposition (LAD) method. The basic idea of LAD is that if N candidate threads should be decomposed ($N > Maximum\ Width$), we look one step ahead on each candidate thread to check if any two candidates share a common successor thread. If that happens, we call them *relative threads*. The candidate threads are distributed into different groups in such a way that all threads inside a group are relative threads. Then we can partition threads inside each group. Because candidate threads that share common successors stay in the same group, our partition will not allocate them to different TPs unless the group size is larger than the maximum width.

3.1.3. Partition Dependency Control During the decomposition, the dependencies that are broken by the TPs' boundaries give rise to a set inter-partition dependencies, which is referred as *graph dependencies*. These graph dependencies are additional constraints over the original dependencies specified by the edges inside those TPs. A graph dependency between thread partitions $TP1$ and $TP2$ makes the activation of any threads inside $TP2$ later than the completion of the last thread in $TP1$. After the thread frame decomposition, each TP is regarded as an independent unit by the scheduler. This makes the graph dependencies crucial to preserve the control- and date-dependencies in the

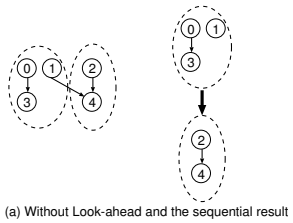

Figure 5. Non-lookahead vs. Lookahead Decomposition

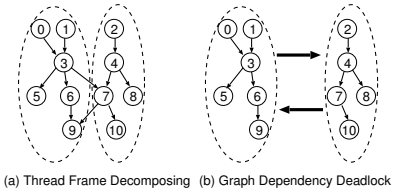

Figure 6. Invalid TF Decomposition

original TF. If we model N decomposed TPs with a graph $G(V, E)$, where $V = \{v_i \mid 1 \leq i \leq N\}$ denotes all TPs 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, an arbitrary TF decomposition by only following the decomposition guidelines discussed so far could lead to an invalid non-DAG for the TPs, as illustrated in Fig. 6.

It is hard to detect the deadlock graph dependencies during an arbitrary TF decomposition. 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 TP, 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 TP). If we let that all inbound graph dependencies only occur at the entry threads, we can then guarantee that the decomposition on the original TF does not have graph dependencies in deadlock. Our decomposer controls the graph dependencies by using a two-step decomposition method. The first step is called *thread partition initialization* (TPI). It creates initial TPs from the set of threads with the same generation numbers, i.e. it creates TPs in a single generation layer. The second step, *thread partition expansion* (TPE), then tries to expand the given initial TP to the next generation. The new partition is fed back to the second step and the expansion is continued until either no further successors are available, or all successors have at least one predecessor not partitioned to the current TP.

3.2. 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 TPs are created from threads at generation 0 with consideration of LAD mechanism (line 5). Each initial TP 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 TPs are created from the earliest generation level (line 8).

Algorithm 1 Thread Frame Decomposition

```
1: INPUT: Input TF
2: OUTPUT: TPs, Partition Dependencies
3: Initialize TF data-structures
4: Topological sorting on all threads of input TF
5: TPI at generation 0
6: while Input TF is not fully decomposed do
7:     TPE on each initial TPs
8:     TPI from undecomposed part of the input TF
9: end while
```

4. Clustering Thread Partitions

After the decomposition, the input TF is broken into multiple TPs which are then passed to the design-time scheduler [10]. The design-time schedules of each thread partition are explored by the design-time scheduler running independently from other TPs. Therefore, multiple instances of schedulers can run simultaneously to speed up the overall scheduling. Note that our hierarchical scheduler does not depend on specific design-time schedulers, any scheduler that can carry trade-off explorations would be able to be employed by the hierarchical scheduler. This design-time schedulers can generate the Pareto-optimal schedules on the performance-energy trade-off space for each TP. All Pareto-optimal schedules of a TP is referred to as its Pareto curve.

Traditional hierarchical thread scheduling techniques stop after decomposing the input TF and only schedule each thread partition individually. In contrast, our hierarchical scheduling approach takes 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 TPs in a row.

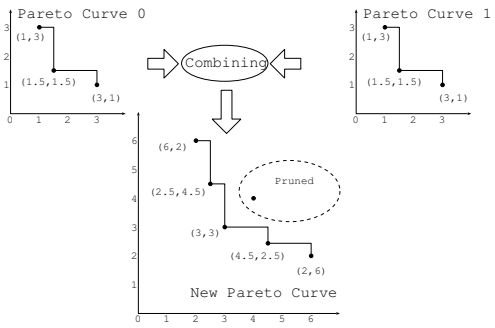

Figure 7. Merger of Pareto Curves

An interleaving phase is necessary to achieve that parallelism. Before the interleaving phase, we must a) cluster the TPs such that all partitions inside a cluster have no dependencies between each other, and b) combine the Pareto curves inside each cluster and prune the combinations to extract a Pareto curve for each cluster. One combination is formed by picking one Pareto point from each curve in the cluster and adding them up, i.e., the combination's energy figure is the sum of all selected Pareto points' energy figures and its timing figure is the sum of all timing figures. Note that such combinations imply that no overlapping among TPs exists at this stage; overlapping only takes place after applying the interleaving technique introduced in the next section. Once all combinations are generated, we only select the Pareto optimal combinations as illustrated by Fig. 7.

5. Interleaving Thread Partitions

Each cluster has a set of Pareto-optimal schedules after pruning. Those Pareto-optimal schedules represent the sequential combinations of Pareto-optimal schedules of individual TPs within this cluster. That is, each Pareto optimal schedule of the cluster represents a one-by-one running of all TPs inside. This means that although the TPs do not have dependencies within a cluster, they have to be run in a sequential manner. Meanwhile, an interesting observation of Pareto-optimal schedules of individual TPs is that they have significant processor idle times inside. These idle times, referred as slacks, are created mainly due to two reasons: a) when decomposing the original TF, we have limited thread parallelism within a TP to reduce the scheduling time. This limited parallelism leads to the 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-utilizes the parallelism in favor of the energy efficiency. In order to reclaim the slacks and thereby generate better Pareto curves, we propose to apply a new technique to exploit the thread concurrency across partitions, namely the *interleaving*.

The idea of interleaving is to build a new global schedule based on shifting the schedules of individual TPs. During this shifting, both the processor allocation of a thread and the sequence of threads that belong to one TP are unchanged. For a clear problem definition, we provide the problem formulation as follows. A TP's schedule S_k for c processors is a list $(M_1, ..., M_c)$, where $M_i = ((b_j, et_j), ...)$ is the list of threads scheduled on the ith processor and (b_j, et_j) denotes that the first thread on this processor is s_j with the start time of b_j and the execution time of et_j. For k schedules $(S_1, ..., S_k)$ of the TPs $((V_1, E_1), ..., (V_k, E_k))$ on a platform with c processors, the interleaving problem can be formally expressed as:

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

such that:

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

Scheduling threads with non-uniform execution times on multiple processors is notorious for its intractability [6]. In fact, Hoogeveen *et al*[7] have proved that even for three processors, scheduling threads with fixed processor allocations is a NP-hard problem. We have used an existing interleaving heuristic based on the first-come-first-served principle [12] to generate interleaved schedules.

6. Experimental Results and Discussions

We have used a software tool called TGFF [4] to generate random TFs for experiments. TGFF generates random TFs according to the specified options such as the thread number. In addition to generating the random TFs, TGFF can be modified to generate a random configuration of multiprocessor platform.

To measure optimality of Pareto curves, we first need to calculate the lower bounds. The lower bound of energy consumption for scheduling a TF is calculated by allocating each thread to the processor with minimal energy consumption and then summarizing all threads' energy figures. The lower bound of execution time is calculated by allocating threads to their fastest processors and summarizing all individual execution times; then the summarized execution time is divided by the number of processors on the given platform. Note that a lower bound of time may not be reached by any feasible schedule at all. But no feasible schedule could have an execution time shorter than a lower bound.

The optimality measurement of the scheduling results is then performed by applying the metric of Pareto-optimality,

		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

Figure 8. Pareto optimality and Scheduling Time Comparison: Flattened *vs.* Hierarchical Scheduling

that is, we measure the differences between a result's energy/time and the lower bounds of energy/time. The product of a result's time difference and its energy difference is used to evaluate its Pareto optimality. The optimality of a Pareto curve is then evaluated by the mean value of all Pareto points' products. A large number of random TFs are generated with 50, 75 and 100 threads inside. TFs are decomposed with the maximum partition width of 5. We have conducted the scheduling experiments for platforms with 6 and 8 processors, respectively. The design-time scheduler of [10] is used as the reference flattened scheduler for comparisons. A lower value of the metric of Pareto-optimality in experimental results (Fig. 8) represents a better Pareto curve in the sense that it gives faster schedules at lower energy consumptions.

7. Related Work

General graph decomposition methods of the divide-and-conquer strategy have been investigated for many years. Most of them, such as [5], have been aimed at solving general graph decomposition problems without considering the constraints introduced by the scheduling process after the decomposition, i.e., these methods do not consider how to handle the task dependencies. Recently, [1] has combined the decomposition problem and the scheduling problem within an unified hierarchical scheduling flow. However, their work has only considered the performance of resulting schedules and hence severely reduced the scheduling exploration space for each sub-graph. General purpose evolutionary algorithms have been widely studied for multi-objective optimization problems (see [2] for a good survey). They have recently been adapted to the embedded software synthesis methodologies [3, 8]. These evolutionary algorithms distinguish from previous scheduling algorithms by their capabilities to explore the trade-off space of the multi-objective 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. Despite their extremely long scheduling time, they are also not robust in terms of optimality; because they choose starting points randomly, and a bad starting point can dramatically reduce the result's quality.

8. Conclusions

This paper has presented a hierarchical scheduling approach based on the graph partition and the interleaving technique. This approach can deal with large TFs in a scalable way by decomposing a TF into multiple TPs and performing scheduling at each partition independently. The subsequent interleaving technique ensures that parallelism among TPs are exploited in the final schedule. Pareto optimality of the scheduling results from our hierarchical scheduler is 50% to 80% better than that from the reference flattened scheduler on average. Moreover, the scheduling times are reduced by up to 2 orders of magnitude, which enables fast design space explorations for the very large embedded systems.

References

[1] I. Ahmad and et al. On parallelizing the multiprocessor scheduling problem. *IEEE Trans. Parallel Distrib. Syst.*, 10(4) 1999.

[2] K. Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Ltd., England, 2001.

[3] R. Dick and N. Jha. Mocsyn: Multiobjective core-based single-chip system synthesis. In *DATE '99*

[4] R. Dick, D. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *CODES'98*

[5] G. Even, and et al. Divide-and-conquer approximation algorithms via spreading metrics In *IEEE Foundations of Computer Science'95*

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP Completeness*.

[7] J. A. Hoogeveen and et al. Complexity of scheduling multiprocessor tasks with prespecified processor allocations, 1992.

[8] M. T. Schmitz, and et al Co-synthesis of energy-efficient multi-mode embedded systems with consideration of mode execution probabilities. *IEEE TCAD*, 24(2) 2005.

[9] F. Thoen and F. Catthoor. *Modeling, Verification and Exploration of task-level concurrency in real-time embedded systems*. Kluwer Academic Publishers, 1999.

[10] C. Wong, and et al Task concurrency management methodology to schedule the MPEG-4 IM1 player on a highly parallel processor platform. In *CODES'01*

[11] P. Yang, and et al A cost-performance tradeoff aware scheduling method for single chip multiprocessor embedded system. *Design & Test of Computers*, 18(5) September 2001.

[12] Z. Ma, and et al Hierarchical task scheduler for interleaving subtasks on heterogeneous platforms *ASP-DAC'05*