Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems *

Peng Yang† Francky Catthoor‡

IMEC, Kapeldreef 75, B3001 Leuven, Belgium, {yangp, catthoor}@imec.be

ABSTRACT

Pareto-set-based optimization can be found in several different areas of embedded system design. One example is task scheduling, where different task mapping and ordering choices for a target platform will lead to different performance/cost tradeoffs. To explore this design space at run-time, a fast and effective heuristic is needed. We have modeled the problem as the well known Multiple Choice Knapsack Problem(MCKP) and have developed a fast greedy heuristic for the run-time task scheduling. To show the effectiveness of our algorithm, examples from randomly generated task graphs and realistic applications are studied. Compared to the optimal dynamic programming solver, the heuristic is more than ten times faster while the result is less than 5% away from the optimum. Moreover, due to its iterative feature, the algorithm is well suitable to be used as an on-line algorithm.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems a nd embedded systems*; D.4.1 [**Operating Systems**]: Processing Management

General Terms

Algorithms, Performance, Design

Keywords

Pareto optimization, scheduling, embedded system, low-power

1. INTRODUCTION

The merging of computers, consumer and communication disciplines gives rise to very fast growing markets for personal communication, multimedia and broadband networks. At the same time, technology advances lead to platforms with enormous processing capacity, which is however not matched with the current system design productivity. The future embedded system design should be based on platforms, which have sufficient computation and memory resources, consume extremely low energy and are flexible enough to cope with the dynamic behavior of future multimedia applications.

For this kind of design, one of the most critical bottlenecks is the very dynamic and concurrent behavior of many of these new applications, which are fully specified in software oriented languages (like Java, UML, SDL, C++) and still need to be executed in real-time cost/energy-sensitive way on the usually heterogeneous SoC platforms. An effective methodology is required to map this system specification onto an embedded multiprocessor platform, and normally this leads to a tradeoff among different system design objectives.

Pareto optimization is a good way to represent and explore these tradeoffs. The Pareto-optimal concept comes from multiobjective optimization problems, where more than two conflicting optimization objectives exist (e.g., for a given IC processing technology, one can not reduce the delay and power of a digital circuit simultaneously). A solution is said Pareto-optimal when it is optimal in at least one optimization objective direction. Such a set of Pareto-optimal solution is normally called a Pareto set and can be conveniently represented by a Pareto curve. In real-world applications, Pareto optimal set can be the result of many different factors, such as architecture (CPU and/or memory) mapping, quality of service control(QoS), as long as more than one optimization objective exists. A systematic way is presented on how to generate the Pareto curves [6] when the designer explores the design space. Traditional design methodology extract that Pareto optimal set and select one point from that set, both at design time. This approach can be used for static systems. However, for modern dynamic systems, we can explore the design space at design time but defer the selection step till run time, to better tune the system to the changing environment.

The Pareto optimal set can come from many design parameters[6]. In this paper we will focus on on-line task scheduling in the Dynamic Voltage Scaling(DVS) context[8] because it is well known to most of the embedded system design researchers. However, the methodology and algorithm are not limited to DVS and can be readily applied to any other similar problem where a multiobjective tradeoff exists.

The rest of the paper is structured as follows. Section 2 uses a motivational example to illustrate our task scheduling method. In Section 3, related work is discussed. Following the problem formulation and algorithm description in Sec-

*This work was partly supported by the Flemish government in the scope of the MEDEA and MESA projects.

†also Ph.D. student of K.U.Leuven-ESAT

‡also professor of K.U.Leuven-ESAT

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

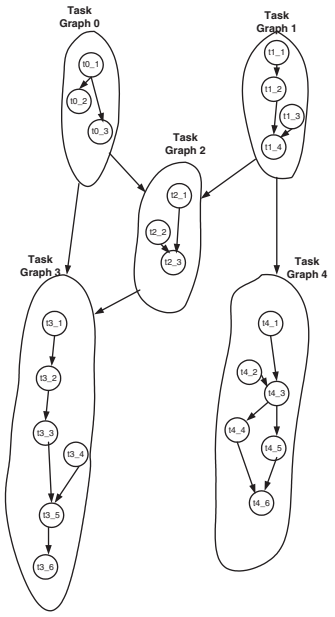

Figure 1: The task graph of the motivational example.

tion 4, Section 5 gives our experimental results on both randomly generated and real-life applications. Finally Section 6 concludes the paper.

2. MOTIVATIONAL EXAMPLE

The advantages of our novel approach can be shown by an example generated by Task Graph For Free (TGFF)[5]. Fig. 1 shows an application which requires the cooperation of five task graphs, denoted from 0 to 4. The details of those task graphs can be found in Tab. 1, in which the first

	TG0	TG1	TG2	TG3	TG4
Number of nodes	8	47	12	29	21
Number of arcs	9	59	13	35	26
Exec. time(us)	284	1258	371	826	661
En. consum.(uJ)	222	1211	331	814	671

Table 1: Task graphs generated by TGFF.

row shows the number of nodes in each task graph, the second row is the number of arcs, and the third and fourth rows give the execution time and energy consumption respectively when the task graph is executed completely on a 3.0V processor. We assume the application is frame based, i.e. every time frame the application will be executed once to process the current input data. This is a reasonable abstraction for the multimedia processing or communication (e.g. mp3 or image decoding). Depending on the content of the input data, not all 5 task graphs of the application will be needed for that specific time frame. In our motivational example we simply assume that every time frame each task graph is selected randomly. The last assumption is that the application has to finish before a deadline, which is normally the period of the time frame.

In this paper, we investigate the single-processor task scheduling. To handle the worst case, which happens when all 5 task graphs are selected, the CPU has to be powerful enough to

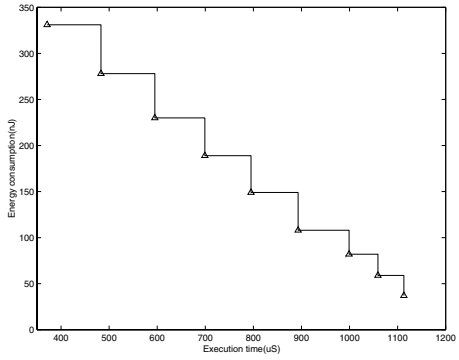

Figure 2: The Pareto curve of task graph 2.

complete the application before the deadline. On the other hand, most of the time only a few of the 5 task graphs are active and hence DVS can be applied to save energy. The online inter-task DVS algorithm[19], which is called so because the voltage scheduling is done dynamically at the boundary of task graphs, monitors the application execution. Whenever it sees some slack time available, it scales the working voltage accordingly. For instance, with a deadline of 3.4ms, a CPU working at 3.0V is just good enough to finish all the task graphs in time. Every task graph will take exactly the execution time and energy given in Tab. 1. Now suppose at the end of task graph 0, the inter-task DVS scheduler notices that task graph 3 will not be active for the current time frame. Therefore a slack of 0.826ms is available. We still have to run task graph 1, 2 and 4, which will normally take 2.29 ms and consume 2213uJ, while the time available is 3.4ms subtracted by 0.284ms, i.e. 3.12ms. Using the conventional equation[7], even if the inter-task DVS reduces the CPU working voltage by 2.29/3.12=0.73, the application can still be completed in time, but the energy consumption is now 2213*0.73*0.73=1179uJ, i.e. almost half of the original value.

The inter-task DVS can save much energy, but it assumes a continuous variable voltage. That requires special circuit design and processing technology and a DC-DC converter which is not very energy efficient(80-95%, depending on the output voltage[4]). Besides, inter-task DVS cannot see the internal contents of the task graph, and is not able to explore the slack time coming from that.

Our scheduler behaves quite differently from the inter-task DVS. Firstly, we use only a limited set of discrete voltages (two for the first example, namely 3.0V and 1.0V). Secondly, the internal content of the task graph is divided into scheduling nodes and scheduled at that granularity. Thirdly, we schedule it in two phases. At design time, the task graph is scheduled at the scheduling node granularity, i.e. each node is assigned to one of the discrete voltages and ordered in time, generating a set of Pareto points with different execution time and energy consumption. The Pareto curve we extract for task graph 2 is given as an example in Fig. 2. At run time, knowing the active task graphs of the current time frame and their Pareto curves, the run-time scheduler is able to select one Pareto point (which represents a specific voltage assignment and ordering for that task graph) from each active curve and can combine them together to get the complete scheduling, taking into account the time constraints.

Our approach benefits from the fact that we know which task graph is selected for the current frame when we schedule the application. This is realistic for real-life multi-media applications by inserting extra code to extract the necessary information, at the start of each frame (see e.g. [21]). If it is impossible to do so, the run-time scheduling decision should be re-evaluated as soon as that info becomes available during the task execution. But even then, our run-time heuristic is fast enough to accommodate that situation. This is also one of the major reasons why the heuristic should be fast, and preferably also scalable in terms of speed versus solution quality.

We have simulated the scheduling of the above problem for 1000 frames. In each frame the task graphs are selected randomly and the time constraint is always 3.4ms. The results are shown in Tab. 2, in which our two-Vdd scheduler

	no DVS	inter.	PC_2	PC_3	optimal
en. cons.(uJ)	1620	1068	956	823	705
en. saving	0	34%	41%	49%	56%

Table 2: Energy consumption of the motivation example with different schedulers.

is denoted as PC_2. For comparison, we also list the energy number for an "optimal" scheduler which uses a continuous optimal DVS strategy. This is not achievable in practice because it requires the full knowledge of the future run-time behavior of the tasks. In this simple example, compared to the state-of-the-art inter-task DVS schedulers (see related work section), our approach saves 7% more energy, which is quite good taking into account we have only two discrete voltages instead of a continuously changeable one. When a third voltage, 1.5V, is available (PC_3), 15% more energy can be saved compared to the inter-task case. This result also comes close (within 7%) to the theoretical (unachievable) optimal value.

These results show clearly the potential advantage of our method. In section 4 we will present a run-time scheduling heuristic on how to achieve it.

3. RELATED WORK

Task scheduling has been investigated overwhelmingly in the last decades. When a set of concurrent tasks - i.e. tasks that can overlap in time - have to be executed on one or more processors, a predefined method, called scheduling algorithm, must be applied to decide the order in which those tasks are executed. For a multiprocessor system, another procedure, assignment, is also needed to determine on which processor one task will be executed. A good overview of scheduling algorithms can be found in [17]. In our paper, the terminology *task scheduling* is used for both the task ordering and the processor assignment.

Generally speaking the scheduling algorithms can be classified as off-line and on-line algorithms. We focus here only on the latter because it provides better opportunity for run-time optimization. These algorithms are usually priority-based and are derived from Liu and Layland's classical paper [11], in which the priority of a task is either set statically, e.g. the Rate Monotonic (RM), or dynamically, e.g. the Earliest Deadline First (EDF). Nowadays, RM and EDF are still the most used algorithms.

Recently, Dynamic Voltage Scaling(DVS) is getting popular ([8, 16] for a good survey). Traditionally, the CPU works at a fixed supply voltage, even for a light workload, when high CPU speed is unnecessary and can be traded for a lower energy/power consumption by a reduction of the supply voltage. This simple observation leads to many algorithms.

Shin and Choi [19] extended the normal fixed-priority scheduler to implement an on-line inter-task DVS for real-time application. They slow down the processor when there is only one eligible running task and completely shut it down when that last task ends earlier than expected. Their work is extended in [20] by adding a two-phase strategy: an off-line algorithm decides the basic voltage for a task set with a priority-based scheduler, while the on-line algorithm refines the off-line decision, using the approach in [19].

To fully utilize the slack time in the on-line DVS, some new approaches are proposed recently. They can be classified as intra-task DVS, because they benefit from looking inside the boundary of a task. Lee and Sakurai [9] partition a task into several pieces, called *time-slots*, then dynamically control the supply voltage on a "time-slot by time-slot" basis, with the feedback from the software. In [10], a combination of hierarchical Finite State Machine and Synchronous Data Flow (SDF) actors is used to model the system. Scheduling is done at the SDF actor level. Whenever one actor finishes, the scheduler is recalled to see whether the current execution path has more slack time available, either because the previous actor finishes earlier or because an execution path different from the worst case is selected. Other approaches include checking the current code execution against data generated at compile time to find the opportunity for voltage scaling[2, 18]. To take full advantage of DVS, some authors further suggest a handshake between the content or application and the OS[3, 15]. Our approach also has a compile-time stage. However, different to the above ones, it computes and stores a partial scheduling which is used in the run-time phase.

The Knapsack Problem and its generalization, Multiple Choice Knapsack Problem (MCKP), which forms the underlying base for our technique, have been studied for decades due to their importance in operation and optimization problems. Martello and Toth[12] discuss the knapsack-like problem in detail. A good exact algorithm is given in [14]. Good heuristics are being looked for in many different contexts[1]. Mejia-Alvarez *et al* applied it to DVS-like problems and a heuristic is constructed from the linear relaxation of the original MCKP[13].

4. RUN-TIME SCHEDULING ALGORITHM

Section 2 shows the effectiveness of our two-phase, Pareto-curve-based scheduling methodology. The key step of this method is the run-time scheduler. Given a set of Pareto curves and a deadline, the run-time scheduler has to select one and only one point from each active Pareto curve and combine them into the final scheduling. It has to be done fast because that will allow a more frequent (re)evaluation of the run-time scheduling decision or the handling of more tasks in a single shot. Both will result in still more energy saving. The quality of the solution is also important because it affects the amount of energy saved.

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

4.1 Problem Specification

The run-time scheduling problem can be formulated as a constrained minimization problem:

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

$$subject\ to \quad \sum_{i=1}^{k} \sum_{j=1}^{N_i} t_{ij} x_{ij} \leq D, \tag{2}$$

$$\sum_{j=1}^{N_i} x_{ij} = 1, i = 1, \ldots, k, \tag{3}$$

$$x_{ij}\ is\ 0\ or\ 1, i = 1, \ldots, k,\ j = 1, \ldots, N_i. \tag{4}$$

where k is the number of Pareto curves, each of which has N_i Pareto points. e_{ij} and t_{ij} are the energy consumption and execution time of the jth point on the ith curve respectively. x_{ij} is 1 if the jth point on the ith curve is selected; otherwise it is 0. The total number of points is denoted by n, $n = \sum_{i=1}^{k} N_i$.

The minimization problem can be transformed into a different form[13]. Taking into account that each Pareto curve is an ordered set, we can substitute e_{ij} with s_{ij} as

$$s_{ij} = (e_{i0} - e_{ij}), s_{ij} \geq 0. \tag{5}$$

Thus Eqn. 1 becomes a maximization problem:

$$maximize: z^{'} = \sum_{i=1}^{k} \sum_{j=1}^{N_i} s_{ij} x_{ij} \tag{6}$$

With the same set of constraints, this is a classical Multiple Choice Knapsack Problem (MCKP) and it is known as NP hard[12].

When of limited size, MCKP can be solved optimally in pseudo-polynomial time through dynamic programming. However, the worst case computation complexity of DP is still exponential, which is not acceptable as an on-line algorithm for big problem size.

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

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

4.2 The Greedy Heuristic

We have developed a fast and effective greedy heuristic with the above considerations. Alg. 1 consists of two stages, the initialization (line 2 to 16) and the iteration stage (line 18 to 41). Every point i of our Pareto curve m is denoted by two basic parameters, $t_{m,i}$ and $e_{m,i}$, standing for the execution time and energy consumption if that point is selected by the scheduler(the corresponding concepts in MCKP are weight and profit). D is the deadline. In the initialization stage, we compute the changes of t and e if we move to the right (from point i to $i+1$) or to the left (to point $i-1$) and the corresponding slopes (line 5 to 12). Here a superscript "+" means the rightward direction and "-" means the leftward direction. The initial solution is found at line 13 and 14: a portion of the deadline(s_m) is assigned to a curve proportional to the execution time of its leftmost point. Therefore it guarantees a valid initial solution can always be found for that curve. When finding the initial solution we use an on-the-fly strategy. The difference between the time assigned to curve m and the actual execution time of its initial solution will be accumulated in the variable $slack$ and added to the available time of the following curves.

Algorithm 1 The greedy heuristic algorithm.

```
 1: INITIALIZATION
 2: step 0:
 3: slack=0;
 4: for all curve m do
 5:   for all point i on curve m do
 6:     δe+_{m,i} = e_{m,i} − e_{m,i+1};
 7:     δe−_{m,i} = e_{m,i−1} − e_{m,i};
 8:     δt+_{m,i} = t_{m,i+1} − t_{m,i};
 9:     δt−_{m,i} = t_{m,i} − t_{m,i−1};
10:     slope+_{m,i} = δe+_{m,i}/δt+_{m,i};
11:     slope−_{m,i} = δe−_{m,i}/δt−_{m,i};
12:   end for
13:   s_m = t_{m,0} D / Σ_{l=0}^{k−1} t_{l,0};
14:   search for maximal j with t_{m,j} ≤ (s_m + slack);
15:   update slack;
16: end for
17: ITERATIVE IMPROVEMENT
18: step 1:
19: sort slope+ descendingly and slope− ascendingly;
20: for all curve m in slope+ do
21:   for all curve n in slope− and m ≠ n do
22:     if slope+_m ≤ slope−_n then
23:       goto step 2;
24:     end if
25:     if δe+_m > δe−_n and δt+_m < δt−_n + slack then
26:       change solution of curve m from i to i + 1;
27:       change solution of curve n from j to j − 1;
28:       update slack;
29:       goto step 1;
30:     end if
31:   end for
32: end for
33: step 2:
34: sort slope+ descendingly;
35: for all curve m in slope+ do
36:   if δt+_m < slack then
37:     change solution of curve m from i to i + 1;
38:     update slack;
39:     goto step 2;
40:   end if
41: end for
```

After the initialization, we explore the chances of finer tuning the solution in two steps, $step1$ and $step2$. $step1$ checks the possibility of moving the operating point on one

curve to the right and the operating point on another curve to the left in pair. At line 19, all curves are sorted according to the slopes of their current solutions, $slope^+$ descendingly and $slope^-$ ascendingly. Then the algorithm will try to find two curves m and n, which satisfy the time constraint and reduce the energy consumption most, if the solution of m is changed from i to $i+1$ and the solution of n from j to $j-1$. When no such kind of tuning is possible, the algorithm will enter the next step. $step2$ does the final tuning by finding any curve m which can still satisfy the time constraint if we move its current solution form i to $i+1$.

Assuming k curves and l points are present on each curve, the complexity of the initialization step is $O(k \log l)$ because for every curve we have only to do an ordered search(line 14). The complexity of the iterative stage is also very low. In $step1$ every iteration takes maximally $O(k^2)$ operations, while in $step2$ $O(k)$ operations. The heuristic ends when no improvement is possible, but we can interrupt the iteration at any moment to finish the run-time scheduling in a predefined time slot. In that case the algorithm just returns the best available solution. This capability is very important for a real-time system where bounded and deterministic service is always desirable. The performance of our greedy heuristic is illustrated in Section 5.

5. EXPERIMENTAL RESULTS

We have implemented the greedy algorithm in C and tested it with both randomly generated and real-life applications.

The first test set we have used is the task graphs generated by TGFF. For each task graph, a Genetic Algorithm is used to extract the Pareto curve, on an architecture like the one we used in section 2. Finally the heuristic is applied to find the on-line task scheduling within a given deadline. A dynamic programming(DP) optimal algorithm[14] is used in this step to check the speed and quality of our heuristic.

We have generated three task sets with TGFF, containing 5, 10, and 20 task graphs respectively. For every task graph, we have extracted two Pareto curves, one with 5 points and the other with 9 points. The former is just a subset of the latter. The points are distributed almost uniformly, in the sense of execution time, between the lowest and highest possible values. Different deadlines are then tried on the same task set and same Pareto curves and the results are summed in Tab. 3 and Tab. 4.

no. of cv.	av. sp. up	max. sp. up	av. error	max. error	av. init. sp. up	max. init. sp. up	av. init. error	max. init. error
5	14.9	24.0	1.2%	5.2%	44.0	58.7	4.1%	9.1%
10	8.8	13.2	1.0%	2.9%	42.9	53.3	6.8%	13.4%
20	3.9	7.3	1.0%	2.0%	24.0	50.2	4.5%	8.7%

Table 3: The performance of the greedy algorithm compared to DP, 5 points per curve.

no. of cv.	av. sp. up	max. sp. up	av. error	max. error	av. init. sp. up	max. init. sp. up	av. init. error	max. init. error
5	15.4	24.9	0.6%	3.5%	46.0	65.1	3.4%	10.3%
10	8.4	14.5	0.8%	2.1%	34.5	55.6	4.1%	8.7%
20	4.3	7.7	0.9%	1.9%	26.2	43.4	3.5%	7.0%

Table 4: The performance of the greedy algorithm compared to DP, 9 points per curve.

The performance of our heuristic can be evaluated in two ways: the execution time and the quality of the result. Tab. 3 and Tab. 4 give the overview of the result. In the tables, the first column is the number of curves; the second column is the average speedup of execution time of the greedy heuristic against the DP solver; the third column is the maximum speedup; the fourth column gives the average error between the heuristic and DP solution and the fifth column is the maximum error. The next four columns are just the same but for the initial solution given by $step0$ of Alg. 1.

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

For the on-line scheduling stage, the time spent on the scheduler itself will not contribute to the application functionality execution. So it has to be minimized or bounded, even though we can have a separate CPU to run the scheduler in some architectures. Our heuristic provides the capability of improving the initial solution iteratively until the time slot assigned to the scheduler depletes. This is especially important for big problem sizes, when the scheduler could not run to its end and still has to find a solution in a short time slot. Tab. 5 shows the iterative improvement of our heuristic. The result is for the 9 points per curve,

iter. no.	time(cycles)	solution(nJ)
0	11554	39366
1	36909	39102
2	48201	38857
3	59389	38695
4	70700	38640
5	81939	38556
6	93502	38538
7	103381	38526
8	113225	38463
9	119312	38443

Table 5: The iterative improvement of the heuristic for a 20 curves, 9 points case.

20 curves case because it is the worst case in our experiment when the execution time is considered. The optimal result is 37836nJ and it takes the DP 232k processor cycles to find it. With the heuristic, to find the final solution 38443nJ, it takes 119k cycles, which may be too long. However, the final solution is only 1.6% from the optimal one and we are usually already satisfied with solutions which are not that good but can be found rather fast. If we assume we have 50k(100k) cycles available for the scheduler, which is 0.25ms(0.5ms) on a 200MHz processor, the result we can find is 38857nJ(38538nJ) and it is only 2.7%(1.9%) away from the optimal solution. Even the initial solution is acceptable in this case, which can be found in less than 12k cycles. Given the fact that the run-time scheduler is triggered by external events (e.g. user related) at the frequency of tens of ms, this result is quite good.

We have also tried our heuristic on some real-life applications. One example is the quality of service(QoS) adjustment algorithm of a 3D image rendering application. Every

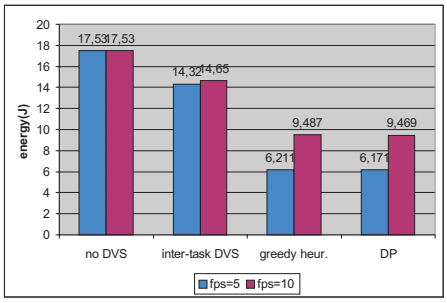

Figure 3: The energy consumption of QoS adjustment algorithm for 1000 frames.

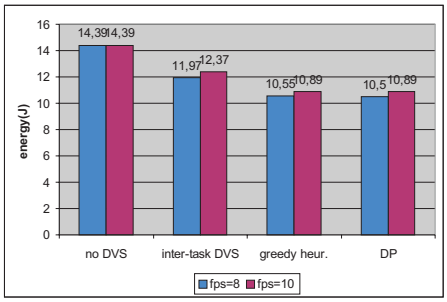

Figure 4: The energy consumption of the VTC decoder for 1365 frames.

time frame, depending on the number of visible objects and which kind of objects they are, the QoS controller will adjust the number of vertices assigned to each object, in order to provide the best quality at a fixed computation power. Fig. 3 illustrates the energy consumption of QoS adjustment algorithm for 1000 frames, with a frame rate of 5fps(frame per second) or 10fps. From this figure it is obvious that our run-time scheduler can achieve a very high energy saving (65% for 5fps and 46% for 10fps). The inter-task DVS does not work very well here because the number of task graphs and the execution time of each task graph varies dramatically in this application. Having to assume the worst case for the unscheduled task graphs, the inter-task DVS scheduler has a limited chance to scale the voltage. Another observation is that the difference between the greedy heuristic and the DP is very small. This is because most of the frames the heuristic and DP will give the same solution, due to the limited problem size.

Another real-life application we have experimented on is the Visual Texture Coding(VTC) decoder of the MPEG-4 standard. Similar to the QoS example, it is frame based. However, unlike the varying number of objects in QoS, the number of blocks to be decoded is fixed (3 in this experiment) for every frame, though the workload of each block varies from frame to frame. As shown in Fig. 4, this example gives less space for voltage scaling because of its relative high and less varying work load. In spite of that, our heuristic still outperforms the inter-task DVS and provides an energy saving of 27%. Again the results from the heuristic and DP are very close.

6. CONCLUSION

In this paper we have modeled the Pareto-optimization-based run-time task scheduling with Multiple Choice Knapsack Problem and have proposed a greedy heuristic for it. Results on randomly generated and real-life applications prove that our heuristic is fast (speedup of more than 10) and high quality (suboptimality less than 5%). The incremental and scalable feature makes the heuristic well suitable for on-line task scheduling. Further research is continuing to extend the heuristic to handle more than two optimization objectives and to integrate the run-time scheduler into Real-Time Operating Systems.

7. REFERENCES

[1] M. Akbar et al. Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem. In *Int. Conf. Computational Science*, pages 112–117, June 2001.

[2] A. Azevedo et al. Profile-based dynamic voltage scheduling using program checkpoints. In *Proc. Design Automation and Test in Europe*, pages 168–75, 2002.

[3] E.-Y. Chung, L. Benini, and G. De Micheli. Contents Provider-Assisted Dynamic Voltage Scaling for Low Energy Multimedia Applications. In *Proc. Int. Symp. on Low Power Electronic Device*, pages 42–7, Aug. 2002.

[4] A. P. Dancy, R. Amirtharajah, and A. P. Chandrakasan. High-Efficiency Multiple-Output DC-DC Conversion for Low-Voltage Systems. *IEEE Trans. VLSI Syst.*, 8(3):252–63, June 2000.

[5] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *Proc. Int. Work. Hardware/Software Codesign(CODES)*, pages 97–101, 1998.

[6] T. Givargis, F. Vahid, and J. Henkel. System-level Exploration for Pareto-optimal Configurations in Parameterized System-on-a-Chip. *IEEE Trans. VLSI Syst.*, 10(4):579–592, Aug. 2002.

[7] I. Hong et al. Power Optimization of Variable Voltage Core-Based Systems. *IEEE Trans. Computer Aided Design*, 18(12):1702–14, Dec. 1999.

[8] N. K. Jha. Low Power System Scheduling and Synthesis. In *Int. Conf. Computer-Aided Design*, pages 259–63, 2001.

[9] S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *Proc. 38th Design Automation Conf.*, pages 806–9, 2000.

[10] S. Lee, S. Yoo, and K. Choi. An Intra-task Dynamic Voltage Scaling Method for SoC Design with Hierarchical FSM and Synchronous Dataflow Model. In *Proc. Int. Symp. on Low Power Electronic Device*, pages 84–7, Aug. 2002.

[11] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[12] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley and Sons, 1990.

[13] P. Mejia-Alvarez, E. Levner, and D. Mosse. Power-Optimized Scheduling Server for Real-Time Tasks. In *Proc. the 8th IEEE Real-Time and Embedded Technology and Applications Symp.*, 2002.

[14] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, Dept. of Computer Science, University of Copenhagen, Denmark, 1995.

[15] J. Pouwelse, K. Langendoen, and H. Sips. Energy Priority Scheduling for Variable Voltage Processors. In *Proc. Int. Symp. on Low Power Electronic Device*, pages 28–33, 2001.

[16] G. Quan and X. Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. In *Proc. 38th Design Automation Conf.*, 2001.

[17] K. Ramamritham and J. A. Stankovic. Scheduling Algorithms and Operation Systems Support for Real-Time Systems. *Proc. IEEE*, 82(1):55–67, Jan. 1994.

[18] D. Shin, J. Kim, and S. Lee. Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis. In *Proc. 38th Design Automation Conf.*, pages 438–43, 2001.

[19] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proc. 36th Design Automation Conf.*, pages 134–139, 1999.

[20] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *Int. Conf. Computer-Aided Design*, pages 365–8, 2000.

[21] P. Yang et al. Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. In *Proc. Int. Symp. on System Synthesis*, pages 112–9, Oct. 2002.