KATHOLIEKE UNIVERSITEIT LEUVEN
FAKULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK
AFDELING ESAT - DIVISIE INSYS
Kasteelpark Arenberg 10, B-3001 Leuven, België

PARETO-OPTIMIZATION BASED RUN-TIME TASK SCHEDULING FOR EMBEDDED SYSTEMS

Promotoren:
Prof. F. CATTHOOR
Prof. R. LAUWEREINS

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Peng YANG

September 2004

In samenwerking met

imec *vzw*

Interuniversitair Micro-Elektronica Centrum
Kapeldreef 75
B-3001 Leuven (België)

KATHOLIEKE UNIVERSITEIT LEUVEN
FAKULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT ELEKTROTECHNIEK
AFDELING ESAT - DIVISIE INSYS
Kasteelpark Arenberg 10, B-3001 Leuven, België

PARETO-OPTIMIZATION BASED

RUN-TIME TASK SCHEDULING

FOR EMBEDDED SYSTEMS

Jury :
Prof. G. De Roeck, voorzitter
Prof. F. Catthoor, promotor
Prof. R. Lauwereins, promotor
Prof. H. De Man
Prof. Y. Berbers
Prof. G. Deconinck
Prof. H. Corporaal (T.U. Eindhoven)

U.D.C. : 621.39

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de toegepaste wetenschappen

door

Peng YANG

September 2004

D/2004/7515/81

ISBN 90-5682-541-0

Acknowledgements

When I am writing down these words, I am looking back into the days and nights of the last five years. Obtaining a Ph.D. is hard work, which is no lack of joys and frustrations. Fortunately, I am not a solitary pilgrim: I am guided and accompanied by my mentors and friends, whom I know I can always turn to, for guidance, discussions, exchanging ideas and support. Hence, at the start of this text, I would like to express my full gratitude to all those who helped me during the last five years.

Prof. Francky Catthoor, for bringing me into the world of embedded systems design, for inspiring and motivating me on the way of research as my promotor. Prof. Rudy Lauwereins, for being my co-promotor and guiding me through all my research. Prof. Hugo De Man and Prof. Yolande Berbers, for being members of the reading committee, for following up my research and for providing invaluable feedbacks. Prof. G. Deconinck and Prof. H. Corporaal, for serving in the jury.

I would like to thank my previous and current group leaders, Diederik Verkest and Johan Vounckx, for providing me with so fantastic an environment to work and study. Thanks to my TCM-mates, Chun Wong, Paul Marchal, Stefaan Himpe, Zhe Ma, Chantal Ykman, Patrick David. I will remember the days we spent together. Thanks to my colleagues in IMEC. I owe them a lot for their generous help and support and for the happy after-hours get-togethers. Dirk Desmet, Shashi Kodamballi, Bingfeng Mei, Prahbat Avasare, Frederik Vermeulen, Miguel Miranda... I can not list all the names but I will keep them in my heart. Also I would like to thank my friends in Leuven and Belgium. They have made my stay here more enjoyable and memorable.

Finally, I would like to thank my family. I know I could never have gone so far if it were not for the support and warm encouragement from them. Also, thanks to my girl friend, for the laughs and dreams she has brought to me.

Peng Yang
Leuven, September 2004

Abstract

The rapid evolution and convergence of computing, consumer electronics and communication disciplines are witnessing a trend toward integrating complete and complex systems on a single chip. Technology advances lead to platforms with enormous processing capacity that is however not matched with the required increase in system design productivity. One of the most critical bottlenecks is the very dynamic and concurrent behavior of many current multimedia applications. In today's designs, quite conservative worst case timing characteristics are used to cope with this, leading to a partial waste of resources and energy due to over-dimensioning.

In order to deal with these new dynamic applications where tasks and complex data types are created and deleted at run-time based on non-deterministic events, a novel system design paradigm is needed. Our complete Task Concurrency Management methodology first represents and transforms the system into several concurrent partitions with an in-house gray-box model. Then a two-phase scheduling approach is proposed to allocate, map and order the tasks and sub-tasks of the system onto the multiprocessor platforms. The design-time scheduler explores the design space per task and stores the exploration results, while the run-time scheduler is used to optimize across all tasks the system performance/cost according to the dynamic system context and the pre-computed task information.

In this thesis, we mainly focus on the algorithms that the run-time scheduler applies to make system-level tradeoffs and the implementation of such a scheduler on top of conventional Real-Time Operating Systems. We have proposed two algorithms for the run-time scheduling. The first algorithm is able to explore the pre-computed design tradeoffs fast, while for systems with more than a few processors, the second algorithm can better exploit the multiprocessor feature and result in better scheduling. To support our two-phase scheduling, a module is developed to provide a generic method to integrate the application, the run-time scheduler and the Real-Time Operating System at low overheads.

The effectiveness of our design methodology has been verified by several real-life demonstrators, both in simulation and on real hardware boards. All results prove our methodology can significantly reduce the system cost at low implementation overheads.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

With the semiconductor processing technology entering in the deep sub-micron era, it has long been recognized that the gap between the processing capability and design capability is not decreasing but increasing, which is known as the design productivity gap. As illustrated in Figure 1.1, the design productivity (number of transistors designed by a designer in one month) increases only 21% a year, whereas the design complexity (number of transistors in a typical design) increases 58% a year. The bottleneck here is not only what we can produce now, but also what we can design. A direct result is the number of designer months (number of designers times total time) in a design project is increasing rapidly. The design cost has become the greatest threat to the continuation of the semiconductor roadmap.

The second problem the designers will confront is the extremely high manufacturing nonrecurring engineering (NRE) cost. As the semiconductor industry approaches the 100-nm technology node, the NRE (mask set and probe card) costs are getting close to $1 million for a large integrated circuit (IC). With an average of just 500 wafers produced from each mask set, rapid growth of manufacturing NRE can throttle the initiation of new IC design projects.

The answers to the design productivity gap problem can come from a) novel system-level design specification; b) hardware-software codesign; and c) substantially reuse intellectual property (IP) components [17]. This caused the revolution of System-on-Chip (SoC) design. To solve the problem of the manufacturing NRE cost, we have to produce the same IC in very high volume to reduce the average NRE cost on every chip. This leads us to the motiva-

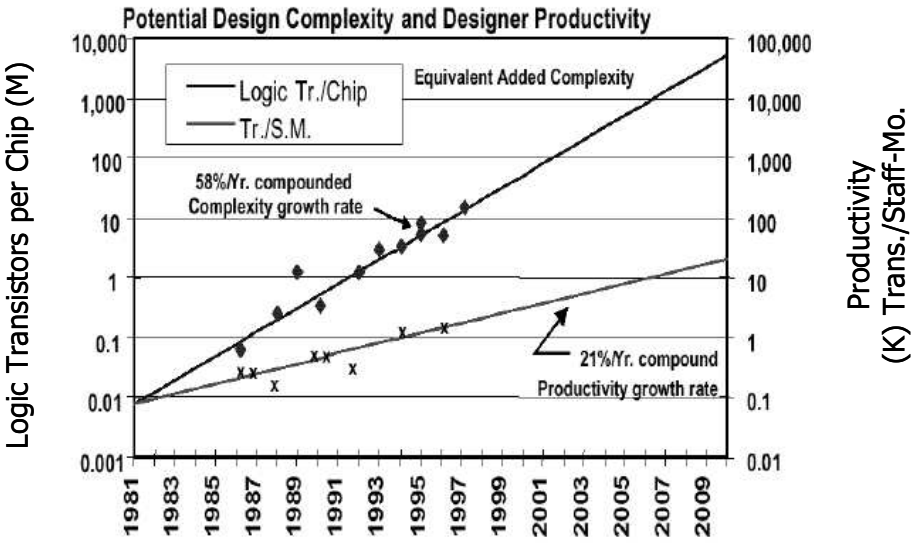


Figure 1.1: Design productivity gap (source: ITRS'99).

tion for a move to platform-based-design (PBD) [108, 37]. The emerging of SoC and PBD has also moved the design complexity from the hardware side to the embedded software side, which is increasing amazingly at a speed of 140% per year and will account for more than 80% of the total design cost according to the ITRS 2003 roadmap in the near future. All these require a novel system-level design methodology for the embedded software, especially to manage the concurrent tasks running on a SoC. As Jan Rabaey said in his keynote speech in DesignCon'04 [131], "It's easy to create concurrency [by putting several components/processors on the same chip], but it's difficult to manage concurrency."

1.1.1 The System-on-Chip Era

The complexity of systems is surging due to the exponentially increasing transistor count enabled by smaller feature sizes and spurred by consumer demands for increasing functionality, lower cost, and shorter time-to-market. To design such complex systems, tradeoffs must be made between all aspects of value or quality, and all aspects of cost.

An SoC is a complex IC that integrates the major functional elements of a complete end-product into a single chip or chipset [108]. In general, SoC design incorporates one or more programmable processor cores (homogeneously or

heterogeneously, even reconfigurable), on-chip memory, and accelerating function units implemented in hardware. It also interfaces to peripheral devices and/or the real world. SoC designs encompass both hardware and software components. Because SoC designs can interface to the real world, they often incorporate analogue components, and can, in the future, also include opto/microelectronic mechanical system components. These components are connected with one or more links, either bus, crossbar, or network-on-chip. Figure 1.2 shows an example of such a device.

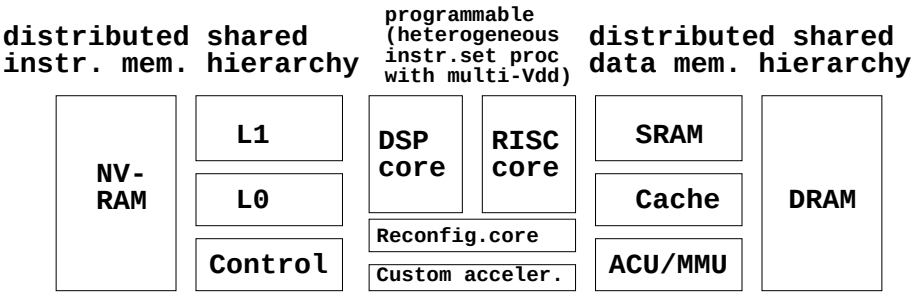


Figure 1.2: A typical System-on-Chip device.

One issue that is worth noting is the trend of putting more than one processing elements into SoCs, either heterogeneously or homogeneously. Adding reconfigurable processing units is also an option. This can be explained by the daunting challenge of managing the SoC power, especially for low-power, wireless, multimedia applications, which needs application-, OS- and architecture-level optimizations including parallelism, and adaptive voltage and frequency scaling. The upper curve of Figure 1.3 gives the intrinsic computation efficiency available from silicon, while the lower one shows the computation efficiency of a single traditional, instruction based processor. From that figure, it is clear that traditional single processor solution is becoming extremely inefficient. An example is the Pentium from Intel. While we are passing 3GHz CPU frequency, it consumes more than 100W and is definitely not the solution to embedded and portable devices, which require processing performance around 2GOPS (Giga Operations Per Second) at average power consumption as low as 0.1W (ITRS 2003 roadmap). However, if we put 4 Pentium cores on an SoC, to provide the same computation power, each core has only to work at one forth of the original frequency. Hence the total power consumption can be reduced dramatically (theoretically the power is reduced by 64 times and the energy is reduced by 16 times, as explained later, if full voltage scaling is allowed[1]). Dif-

[1]In reality, the possible V_{dd} variation is limited by the processing technology. However, techniques such as multi-V_t, multi-T_{ox}, multi-V_{dd} can be applied simultaneous in a single core

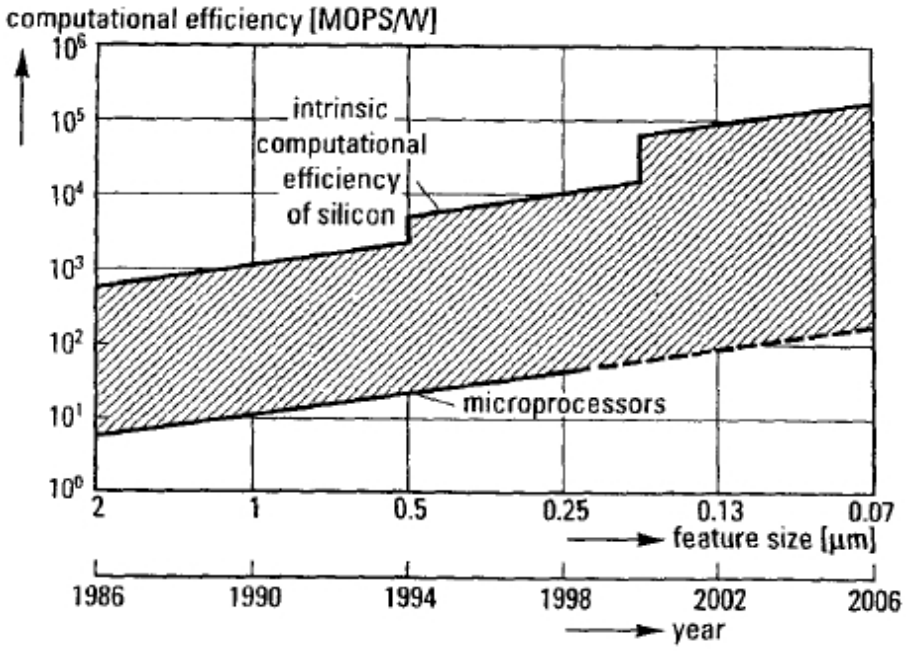


Figure 1.3: Computation efficiency vs. minimum feature length and time [30]. MOPS is Million Operations per Second.

ferent kinds of processors (microprocessor, DSP, ASIP, ...) have different kinds of application domain and different performance/power ratio. For example, for signal processing, a DSP is more efficient with respect to power dissipation and chip area, while a microprocessor is more efficient in handling control-flow specific code. It is possible to explore different levels of parallelism intrinsic in an application and to distribute them to different (kinds of) processors so that the highest power efficiency is achieved. That explains the move to heterogeneous multiprocessor platforms.

The required shift for SoC design is the result of two industrial trends: the development of application-oriented IC integration platforms for rapid design of SoC devices and derivatives, and the wide availability of reusable virtual components [24]. The most obvious way to combine flexibility and cost efficiency is to take the best from both. By their nature, software implementations on programmable cores are preferred to realize maximum flexibility. Tasks which run inefficiently in software, have to be mapped on co-processors (or specific pro-

to achieve ultra low power/energy consumption at a reduced performance level. Multi-V_t and multi-T_{ox} are mainly useful for reducing the static leakage power.

cessor cores) for cost reasons: signal-processing tasks are better implemented on DSP cores or media processor cores than on microprocessor cores, while the opposite is true for control tasks. These considerations lead to the concept of a multi-core "silicon system platform". The move to platform based SoC provides several advantages: high production volumes, the same chip can be used to several related application domains; high flexibility, it is easy to change the functions of the product by simply upgrading the software of the programmable cores; good performance and power numbers; and fast designs, the time to market is significantly shortened by reusing existing HW/SW modules.

1.1.2 Platform Based Design

Design problems are pushing IC and system companies away from full-custom design methods, toward designs that they can assemble quickly from predesigned and precharacterized components. This places priority on design reuse, correct component assembly, and fast, reliable, efficient compilation from specifications to implementations [140].

The platform concept has been around for years, but multiple definitions make its interpretation confusing. According to the VSIA working group, a platform is "An integrated and managed set of common features, upon which a set of products or product family can be built. A platform is a virtual component." In general, a platform is an abstraction that covers several possible lower-level refinements. Every platform gives a perspective from which to map higher abstraction layers into the platform and one from which to define the class of lower-level abstractions that the platform implies. A good example is the motherboard of a personal computer (PC). It defines and characterizes the architecture and capabilities of a PC, whereas it keeps enough flexibility. A designer can choose different components, such as processors, graphic cards, size of memories, to derive different products for different users or for different needs, as long as these components share the same interface supported by that motherboard. At another level, the motherboard itself is derived from a platform at a higher level, which decides, for example, the kind of processor to support (Intel or AMD) or the number of peripheral connection slots. For embedded software, the platform is a fixed microarchitecture that minimizes mask-making cost but is flexible enough to work for a set of applications so that production volume remains high over an extended chip lifetime.

In [108], PBD is defined as "an organized method to reduce the time required and risk involved in designing and verifying a complex SoC, by heavy reuse of combinations of hardware and software. Rather than looking at IP reuse in a block by block manner, platform-based design aggregates groups of components

into a reusable platform architecture."

Typically, PBD is either a derivative design with added functionality, or a convergence design where previously separate functions are integrated. In the vision of [140], ICs used for embedded systems will be developed as instances of a particular architecture platform. That is, rather than assembling them from a collection of independently developed blocks of silicon functionality, designers will derive them from a specific family of microarchitectures – possibly oriented toward a particular class of problems. The system developer can then modify the ICs by extending or reducing them (see Figure 1.4). Good examples of this

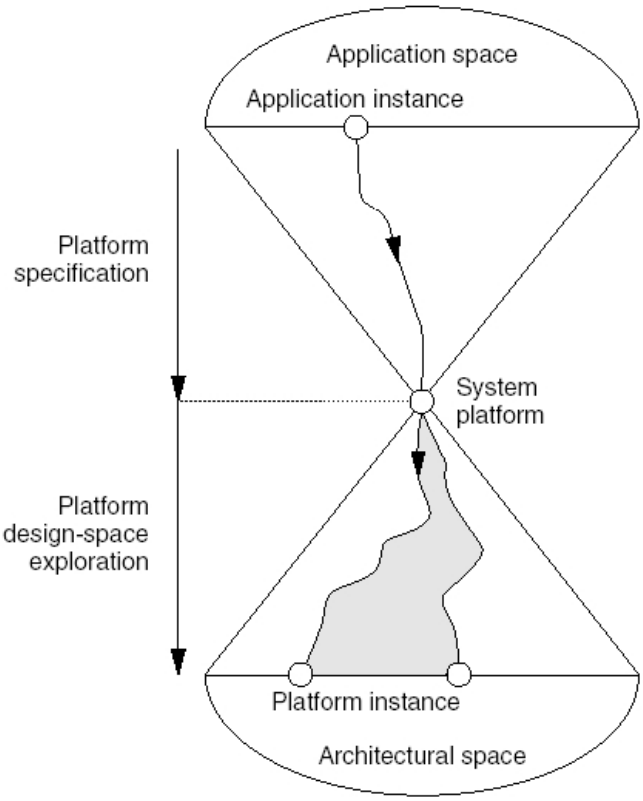


Figure 1.4: System platform layer and design flow. The system platform effectively decouples the application development process (the upper triangle) from the architecture implementation process (the lower triangle).

kind of platform are OMAP of TI, Nexperia of Philips, PrimeXsys of ARM and Virtex-II Pro of Xilinx [108].

PBD is the extension of core-based design. It creates highly reusable *groups* of cores to form a complete hardware "platform," further simplifying the SoC design process. With highly programmable platforms that include one or more programmable processor(s) and/or reconfigurable logic, derivative designs may be created without fabricating a new SoC. Platform customization for a particular SoC derivative then becomes a constrained form of design space exploration: the basic communications architecture and platform processor choices are fixed, and the design team is restricted to choosing certain customization parameters and optional IP from a library. PBD also entails HW-SW partitioning, which decides the mapping of key processing tasks into either HW or SW, and which has major impact on system performance, energy consumption, on-chip communications bandwidth consumption, and other system figures of merit. Multiprocessor systems require "SW-SW" partitioning and codesign, i.e., assignment of SW tasks to various processor options. While perhaps 80-95% of these decisions can be made *a priori*, particularly with platform-based or derivative SoCs, such codesign decisions are usually made for a small number of functions that have critical impact.

To map a specific application on a given platform, since a large portion of the design has already been fixed, the main effort of the design is on the embedded software part, which is becoming increasingly complex and will account for 80% of the design cost (ITRS 2003 roadmap).

1.1.3 Embedded Software

The trend of future embedded systems is now clearly toward wireless, multimedia, multi-functional and ubiquitous applications. This challenges the existing solutions on performance, power, flexibility and costs, calling for innovations in both architecture and design methodology.

In general, the platforms we discuss in the previous section are characterized by multiple programmable components. Thus, each platform instance derived from the architecture platform maintains enough flexibility to support an application space that guarantees the production volumes necessary for economically viable manufacturing. This also results in fast time-to-market. For this kind of platforms, the key problem is how to transform the applications from highly abstract specifications (even in several different computation models) to embedded software implementations and how to map them to the given platforms.

To cope with the tight constraints on performance and cost typical of most embedded systems, programmers write today's embedded software using low-level programming languages such as C or even assembly language. Because of performance and memory requirements, it typically uses application-dependent, proprietary operating systems, or even no operating systems. When embedded

software was simple, there was little need for a more sophisticated approach. However, with the increased complexity of embedded-systems applications (increasing 140% a year), this rather primitive approach has become the bottleneck of the design productivity.

In many emerging embedded-systems applications in areas such as wireless communication and portable multi-media device, the need to capture specifications at high abstraction levels has led to the use of modeling tools such as the Mathworks' Matlab/Simulink and the UML. After the systems are characterized and evaluated, they are further implemented in languages such as Java, SDL, C++ and SystemC. However, these models and high level implementations do not cover the full embedded-system design spectrum. For example, they lack formal dataflow support and the Finite-State Machine capture. Lower level platform/architecture related details are far less considered at this stage. An effective methodology is needed to partition the system and map it onto an embedded platform, normally with more than one programmable components. During this stage, many different partitionings/mappings can be tried, generating different designs, each with different characteristics such as execution speed, power consumption and memory footprint. This is known as design space exploration.

To make things even worse, these applications are completely different from the conventional embedded softwares, which are small in size, simple in architecture, static in function, typically managed by a simple scheduling policy such as cyclic scheduling. The new challenge is the inherent dynamism of the embedded systems. Dynamism is the system property that enables the system to adapt at run time under the influence of user requirement (e.g. the multimedia quality of service). New abstractions are required for such run-time modification of function and architecture. In nowadays embedded systems, typically more than one task is running simultaneously, processing and manipulating complex data structures. These tasks and data are created and deleted at run time to the request of the applications. From time to time, the embedded systems also have to switch between different modes. For example, it can act as a wireless phone at one moment while as a 3D game engine at another moment. Certainly different modes will cause different execution characteristics on the same device. Besides the concurrent and dynamic features of the new applications, the heterogeneous multiprocessor platform makes the embedded software design even more difficult. No effective concurrent programming model exists for the multiprocessor embedded software programming.

Clearly, a gap exists between the high level specification and the final implementation for embedded software design. Hence, a need exists for a systematic design methodology and system-level tool support. The Task Concurrency Management (TCM) approach developed in IMEC (see Chapter 3) is our an-

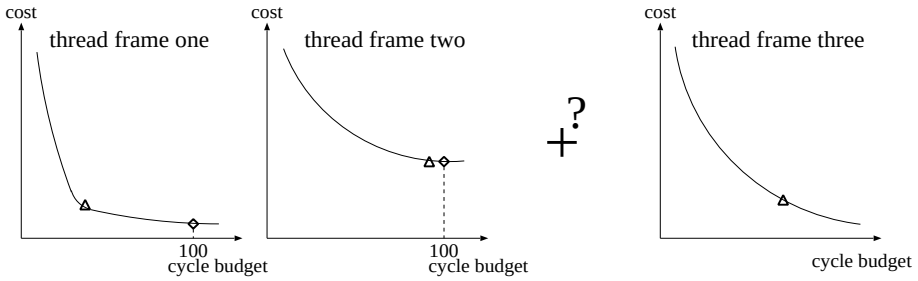
swer to that challenge and our two-phase scheduling methodology is a key component of that approach. The novel scheduling methodology provides design space exploration, run-time task scheduling (to optimize the system behavior) and the integration of the complete application with the RTOS and the hardware below.

1.2 Two-Phase Task Scheduling: Why and How

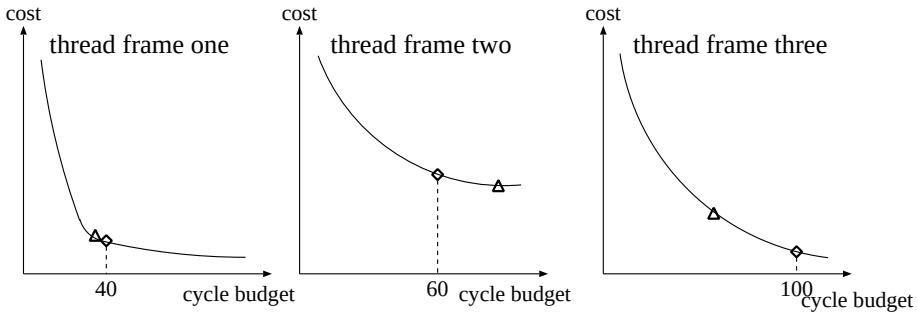
The procedure of embedded systems design is to make decisions and tradeoffs between different values, qualities and costs. This is called the design space. Among all the tradeoffs, some can be fixed *a priori*, while it is beneficial to put off some decision-making till the run-time stage, which becomes ever more crucial for nowadays dynamic applications. This distinguishes our design methodology from others. Conventional compiler oriented design methodologies also explore the design space. However, they do the exploration, evaluation and decision making all at design time. When a design finishes, everything is fixed and the system can not adapt itself to the run-time environments. Hence this approach often results in over-designed and/or energy wasting systems. The other extreme is to do everything at run time with a (Real-Time) Operating System. However, the scheduling problem is notoriously hard to solve. For non-trivial number of tasks, the run-time overhead is very high even with a heuristic. Moreover, due to the limitation of the computing time, the scheduling quality is questionable.

Our methodology is a hybrid of the above two approaches. We do the full exploration at design time but postpone some critical decision making till run time to best make tradeoffs dynamically according to the run-time environment.

The basic idea of our method can be clarified with a simple example (the terminologies used here will be explained in Chapter 3). In Figure 1.5(a), there are originally two thread frames. These two thread frames can each be represented by a typical cost-performance tradeoff curve. The X-axis represents the number of cycles to execute that thread frame and the Y-axis represents the cost of that implementation. Suppose at run time totally 200 cycles are available in the system and the two thread frames are working at the operating points marked as diamond on that figure. Whenever a new thread frame (thread frame 3), accompanied with its tradeoff curve, enters the system, the run-time scheduler has to take into account the tradeoff curves of all three thread frames to minimize the total system cost while satisfying the cycle budget constraint. Conventional design can only solve this problem in an over-designed way: no matter whether thread frame 3 presents or not, the three thread frames always work at the operating points marked as triangle. Clearly, this design is costly when thread frame 3 is not in the system.



(a) before the third thread frame comes



(b) after the third thread frame comes

Figure 1.5: Make tradeoffs for three thread frame system at run time to reduce the system cost. Suppose totally 200 cycles are available. Diamonds are our adaptive solution and triangles are the conventional static solution.

To provide that kind of dynamic adaptiveness, we propose a methodology called Task Concurrency Management (TCM). The purpose of task concurrency management is to determine a cost-optimal, constraint-driven scheduling, allocation, and assignment of various tasks to a set of processors. Task concurrency management comprises three steps, which will be detailed in Chapter 3 but are briefly discussed here. The first is concurrency extraction and improvement, which produces a set of thread frames. Each thread frame consists of many thread nodes, the basic scheduling units. Second, design-time scheduling is applied inside each thread frame at compile time, including a processor assignment decision in the case of multiple processing elements. Finally, runtime scheduling is applied to these thread frames on the given platform. We sepa-

rate task scheduling into two phases for three reasons. First, this scheme better optimizes the embedded software design compared to a pure design time approach. Second, it gives the entire system more runtime flexibility to deal with non-deterministic events. Third, it reduces runtime computation complexity compared to a pure run time approach.

1.2.1 Design-Time Task Scheduling phase

A thread frame's behavior can be described by task graphs such as Fig. 1.6, in which each node represents functions to be performed. Each edge represents

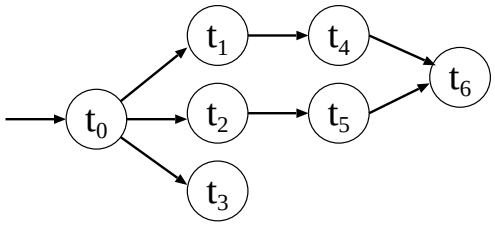


Figure 1.6: An example of a task graph.

the data dependency between two nodes. This task graph represents part of a voice coder [78] and will be mapped to a dual-processor platform. The two processors are almost the same, except that P1's working voltage is three times that of P2. Table 1.1 shows each node's execution requirement on those

	Execution Time									
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
P1	3	10	12	13	16	13	15	30	20	15
P2	9	30	36	39	48	39	45	90	60	45

	Energy Consumption									
	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
P1	27	90	108	117	144	117	135	270	180	135
P2	3	10	12	13	16	13	15	30	20	15

Table 1.1: Normalized thread nodes execution time and energy consumption.

two processors in terms of execution time and energy consumption. We have normalized the execution time and energy consumption numbers because only the relative value is important in this context.

In a well-designed CMOS circuit, the dominant energy consumption term is

the switching component [23], which can be written as

$$energy\ per\ transition = P_{total}/f_{clk} = C_{effective}V_{dd}^2$$

while the maximal frequency is given by

$$f_{max} = \frac{1}{T_d} = \frac{\mu C_{ox}(W/L)(V_{dd} - V_{Th})^2}{C_L V_{dd}}$$

Approximately we can say the energy consumption is proportional to the square of the supply voltage while the processor speed is linear proportional to it. Hence, by decreasing the supply voltage when possible (when the system is not at its worst case), the system energy consumption can be reduced significantly, though the processor speed is also slowed. This technique is called Dynamic Voltage Schaling (DVS). In our example, we assume two fixed voltage and thus avoid the overhead and efficiency loss of continuous DVS.

Figure 1.7 shows the performance-energy tradeoff curve we have derived for

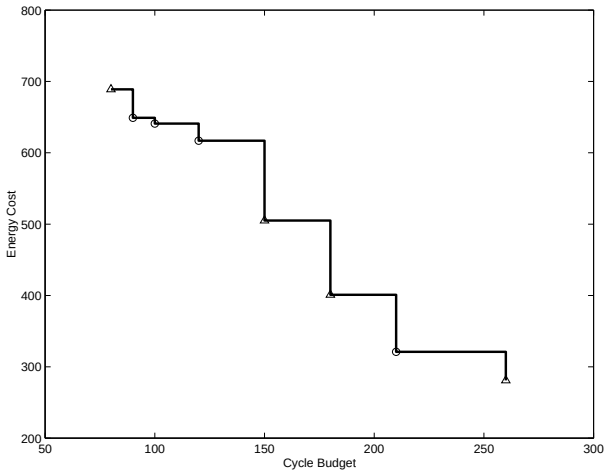


Figure 1.7: Performance-energy tradeoff curve derived in design-time scheduling. Triangles are the operating points to be passed to the run-time scheduler.

the voice coder example and the given processors, after we have tried all the assignment and ordering possibilities for the nodes. Among all the possible operating points, only those chosen by the design-time scheduler as typical cases (indicated by triangles) are passed to the run-time scheduler. The more points are passed, the better the runtime scheduler's results, but at the cost of greater run-time computation complexity and overhead.

1.2.2 Run-time Task Scheduling Phase

Design-time scheduling provides a series of possible allocation and scheduling options inside a thread frame, but only the run-time scheduler decides which option is used. Each option has a thread node assignment and ordering pattern pre-computed by the design-time scheduler. The run-time scheduler considers computation requests from all the ready-to-run thread frames and selects an option for each thread frame so that the entire system's combined energy consumption is optimal. Working with the thread frame as its operational unit, the runtime scheduler considers the timing constraints among thread frames, such as data dependency or execution order. In Table 1.2, for example, each of two thread frames has three options that are identified by the design-time

	Thread Frame One			Thread Frame Two		
	opt.1	opt.2	opt.3	opt.1	opt.2	opt.3
Cycle Budget	20	60	100	40	60	80
Energy Cost	110	80	50	90	60	50

Table 1.2: Example of two thread frames.

scheduler. These options correspond to different cycle budget and energy cost combinations. At runtime, if the total cycle budget for the two thread frames is 100, the energy-optimal schedule is option 1 for thread frame 1 and option 3 for thread frame 2 (see Figure 1.8). If the cycle budget is 140, however, the optimal schedule becomes option 2 for thread frame 1 and option 3 for thread frame 2. Complex trade-offs are involved in distributing the total execution

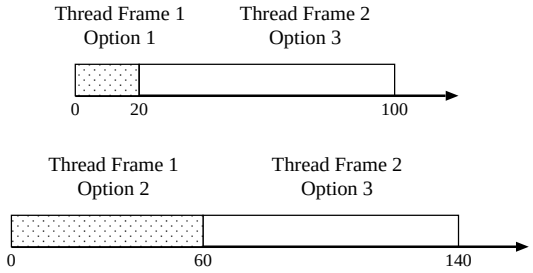


Figure 1.8: Run-time scheduler selects different operating points of the same thread frame according to the system deadline to minize the energy consumption. The top combination is chosen when the deadline is 100; while the bottom one is chosen when the deadline is 140.

period over the different thread frames. An algorithm is needed here to explore the tradeoffs at run time.

1.2.3 Combine Them Together: A Simple Experiment

We use an artificial but typical example to illustrate how we combine the two scheduling phases discussed earlier together and to illustrate how the novel scheduling approach can benefit the design. Here we only give the result briefly. The detail can be found in Section 4.1.

Fig. 1.9 shows an application which requires the cooperation of five thread frames, denoted from 0 to 4. 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 or communication applications (e.g. MP3 or video decoding).

After applying design-time scheduling, we get one Pareto curve for every thread frame. The Pareto curve we extract for task graph 2 is given as an example in Fig. 1.10. 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.

We have simulated the scheduling of the above problem for 1000 periods and the results are summarized in Table 1.3, in which our two-Vdd scheduler is denoted

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

Table 1.3: Energy consumption of the motivation example with different schedulers.

as PC_2. For comparison, we also list the energy number for an "optimal" scheduler which uses a continuous optimal DVS strategy and 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 used by the reference case[2]. When a third voltage is available (PC_3), 15% more energy can be saved compared to the inter-task DVS case. This result also comes close (within 7%) to the theoretical optimal value, which is unachievable because it requires perfect knowledge of the future and continuous V_{dd} scaling.

[2]The energy and time overhead of implementing a continuously changeable voltage is potentially high.

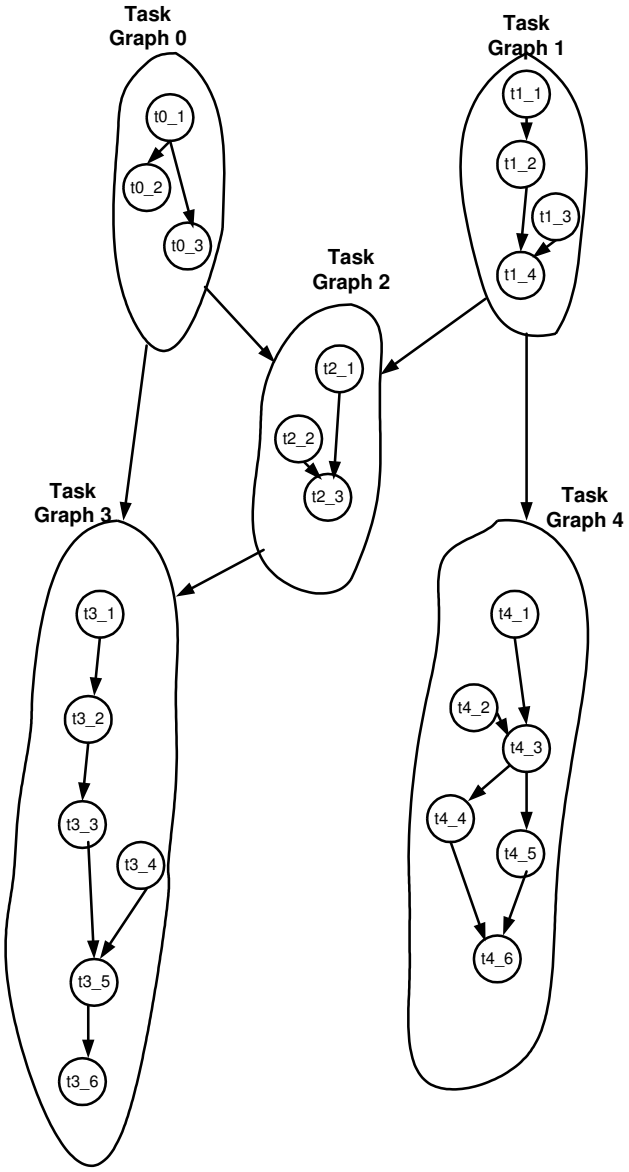


Figure 1.9: The task graph of the motivational example.

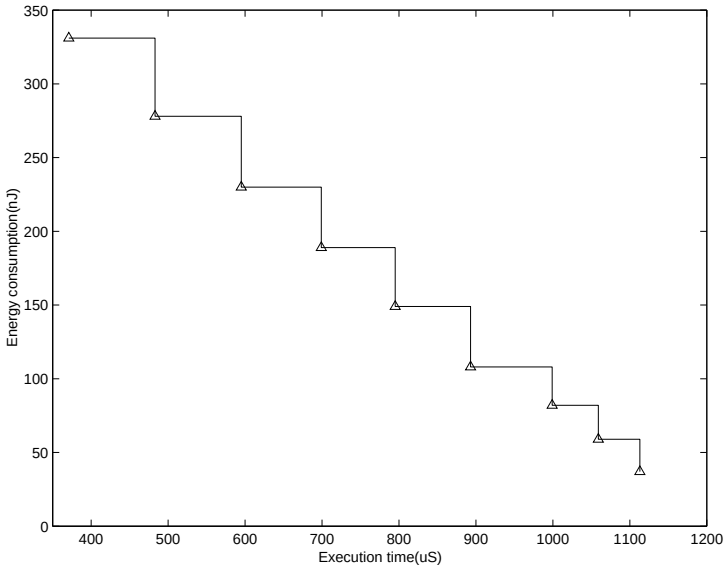


Figure 1.10: The Pareto curve of task graph 2.

1.3 Main Contributions

In developing the design technology of embedded software for concurrent and dynamic applications, as described in this thesis, we have achieved a number of original contributions:

- **A complete design flow and design methodology** has been defined in cooperation with several PhD students working in a team including also Chun Wong, Paul Marchal, Stefaan Himpe and Aggeliki Prayati [180, 183, 175, 169, 106]. In that methodology, an application is first modeled with MTG*, then transformed in order to increase and expose the parallelism embedded. After that, a hybrid design time and run time scheduling is done to adapt the application best to the dynamic system context. This flow is different in many ways compared to the state of the art (see Chapter 2 and Chapter 3).

- **A fast and scalable run-time task scheduling algorithm** has been developed to schedule applications sequentially while exploring different tradeoffs [178]. The problem is modeled as a Multiple Choice Knapsack Problem and a greedy heuristic is proposed. Given the pre-computed information stored as Pareto curve, the heuristic is able to make tradeoff

at run time both fast (speedup of more than 10) and effectively (suboptimality less than 5%). The incremental and scalable feature makes the heuristic well suitable for on-line task scheduling.

- **A Tabu search based heuristic** has been developed to schedule application at run time to further explore the parallelism provided by multiprocessor platforms. Different from the previous heuristic, which executes applications sequentially on a multiprocessor platform, this heuristic allows several applications to run simultaneously, each application utilizing part of the platform (overlapping in time). Energy savings up to 34% have been observed compared to the sequential scheduling heuristic, for the same real time system constraint. To our knowledge, till now, no other run time technique has been proposed for this case.

- **The design flow and design methodology have been verified** by both simulation and on hardware platforms [181, 182]. A 3D Quality of Service application is used for the simulation on top of a PC and a RTOS. The result gives good motivation to our methodology. An experiment with a 3D application on a real XScale board further proves the effectiveness of our method.

- **A middleware-like module has been developed** to allow mapping and scheduling tasks dynamically on a multiprocessor platform, as decided by the run-time scheduler [179]. Inserted between the application and the RTOS, this module integrates the application, the run-time scheduler and the RTOS at low overheads. Experiments on a real dual-DSP platform illustrate significant performance improvement.

1.4 Chapter Overview

The rest of this thesis is organized as follows.

Chapter 2 gives a detailed survey of related work, including task scheduling, voltage scaling and simulation framework.

Chapter 3 presents our task concurrency management design flow in detail. All the steps in this design flow are covered. A simple example is used to illustrate the function of each step.

Chapter 4 presents a fast and scalable heuristic for the run-time Pareto curve based scheduling. A motivating example is given first, followed by the problem formulation and the heuristic algorithm. Then both artificial examples and real-life examples are used to test the effectiveness of this algorithm.

Chapter 5 presents a Tabu search based run-time heuristic scheduling algorithm. Compared to the algorithm in Chapter 4, this heuristic is much more effective and efficient when a cluster of (heterogeneous) processors are available.

Chapter 6 validates our task concurrency management flow with two real-life demonstrators. The 3D QoS adjustment application is simulated on a PC on top of a RTOS. The PocketGL application is implemented on a XScale board running Linux.

Chapter 7 introduces the run-time system to integrate our two-phase scheduling methodology on top of the RTOS in the multiprocessor context. Tests are done on a dual-DSP board to illustrate the overhead of the integration. An experiment with H.263 decoding proves the advantage of applying task concurrency management.

Chapter 8 finally concludes this thesis and points out interesting areas for future research.

Chapter 2

Related Work

Task scheduling has been investigated overwhelmingly in different communities in the last decades. Conventionally, it is used to guarantee objectives such as timeliness or resources constraint, or to optimize objectives such as system response time. Recently, as power consumption is getting a serious problem in embedded systems, many approaches have been proposed to handle that problem. In this chapter, we first give an overview on the conventional and low-power scheduling techniques. After that, we discuss a little about the tools and frameworks which may be helpful and needed in our research work.

2.1 Scheduling Theory

For a system with multiple tasks, an ordering has to be done to decide the execution order of these elements. If more than one processor or processor element exists in that system, another decision, assignment, has to be made as well to decide on which processor a task should be executed. The ordering and assignment problems are more or less coupled in a multiprocessor system and traditionally the term scheduling is used for this coupled problem. Or more formally, scheduling involves the allocation of resources (processor, I/O, bus, etc.), the ordering of the tasks in time (on a relative or absolute time axis), and the binding of them in such a way that certain constraints are satisfied (e.g., timing constraints in real-time system, dependency, resource) and some performance/cost metrics are optimized (e.g., average latency in UNIX; energy in portable systems; quality in QoS). In our context, we do not always impose a complete ordering and assignment, so also partial scheduling constraints can be the result of our TCM design-time scheduling approach.

Scheduling policies or algorithms are the ways to achieve this in practice. Roughly, scheduling algorithms can be divided into design-time and run-time algorithms based on whether the task execution order is decided at design-time or run-time. They are usually known as static and dynamic scheduling respectively. Sometime they are also called off-line and on-line scheduling. In the following subsections, we separate the algorithms into three classes: static scheduling, fixed/dynamic priority scheduling and dynamic scheduling. The priority based scheduling is a kind of dynamic scheduling. However we put it in a separate section because of its importance and the huge number of existing techniques. Finally, we will discuss some topics about Operating Systems.

A very important concept is the Real-Time (RT) system.

> "A real-time operating system has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. A real-time system is considered to function correctly only if it returns the correct result within any time constraints."
>
> –A. Silberschatz and P. Galvin, 1998, Operating System Concepts.

Two aspects distinguish the RT system: rigid time constraint and on-time response. For a RT system, if it can not finish one task before a specific deadline, the system will be in an exception state (hard RT) or a bad but still endurable state (soft RT). Compared to a non-RT system, scheduling is crucial in a RT system because it is the only way to meet the deadlines and that is the most important mission of the system.

In the real-time community, normally the black-box model is used, where the scheduling unit is the task and no details inside the task are known to the scheduler. The tasks are modeled with a task graph, describing the dependencies, and worst case execution time (WCET) are assumed for each task. Comprehensive overviews of scheduling algorithms for real-time systems are given in [166, 145, 135, 4].

Subsection 2.1.1 looks at the algorithms used for static scheduling. Subsection 2.1.2 addresses the priority based scheduling. It belongs to the dynamic scheduling category but we put it in a separate subsection because it is too important to be mixed with other dynamic algorithms. After that, subsection 2.1.3 discusses some other dynamic algorithms and finally the scheduling policies used by Operating Systems is summarized.

2.1.1 Static Scheduling

Up to now, mainly static table-driven or cyclic executive approaches are used for static scheduling [5, 134]. Such *static table-driven* methods perform static schedulability analysis at design-time and the resulting schedule (or table, as it is usually called) is used at run time to decide when a task must begin execution. This approach is highly predictable but also highly inflexible and obviously it can only be applied to periodic tasks. Aperiodic systems can be transformed into periodic tasks, introducing modeling overhead which can be translated into penalties in timing and cost. The table can be constructed using the well-known Earliest Deadline First or Rate Monotonic technique (see subsection 2.1.2). However, when other constraints like precedence, exclusion, resource requirements, etc., are also taken into account, the scheduling problem is NP-hard. Good heuristics have to be found for it. Aperiodic tasks can be handled more efficiently with the methods described later.

Xu and Parnas [177] have examined this scheduling problem using a branch-and-bound technique. The algorithm described in [133] considers communication and replication constraints and is applicable to distributed systems. In [134], a scheme is given to adapt the static scheduling so that some aperiodic tasks can also be considered.

In [44], a white box view is used for the scheduling, where performance is measured by the worst case delay. Communication is included in the scheduling, which applies a two-step strategy: first to find the minimal worst case delay then to generate the corresponding schedule table. Many other works apply a similar approach [53, 80, 98].

2.1.2 Fixed or Dynamic Priority Scheduling

Priority based algorithms are quite common and used in most time-sharing systems. The task which has a higher priority always preempts the current task and gets executed. In non-RT systems, the priority of a task depends on what kind of application it is and is assigned rather arbitrarily. It can be changed by the user or the Operating System. In RT systems, priority assignment is related to the time constraints associated with a job or task and this assignment can be either *fixed* or *dynamic*. Here we only discuss the RT systems. None of these algorithms considers cost and the timeliness is the only objective.

Basic Theory

In Liu and Layland's classic paper [91], they investigated the problem of scheduling periodic tasks on a single processor and proposed two preemptive algorithms. Rate-Monotonic (RM) assigns static priorities to tasks based on their periods and a task with shorter period gets a higher priority. Earliest-Deadline-First (EDF) assigns the priority dynamically: the earlier a task's deadline, the higher its priority. For both of these algorithms, schedulability analysis has to be done in advance. At the cost of some run-time overhead, EDF has a better schedulability.

In addition to EDF, a task's laxity (given by the amount of time one can wait and still meet its deadline) can also be used as its dynamic priority. This leads to the Least-Laxity-First (LLF) algorithm [112].

In Liu and Layland's original paper, there are several restrictions. The RM policy has been extended in a variety of ways to deal with shared resources, aperiodic tasks, tasks with different importance levels, and mode changes. A good overview can be found in [146]. In [85], an exact rate monotonic test is provided, which is both sufficient and necessary. The restriction that the tasks share a critical instant is released in [6]. Arbitrary start times are allowed instead. [5] presents an efficient optimal priority assignment for tasks with offsets. Leung and Whitehead [88] suggest Deadline Monotonic to relax the deadline restriction: task priorities are assigned in inverse order to task deadlines other than task periods in RM.

Chetto et. al. introduce precedence constraints to EDF [26]. They transform the original task set Γ to task set Γ^*, modifying the release times and deadlines so that each task can not start before its predecessors and can not preempt their successors.

Handling Aperiodic Tasks

To provide service for aperiodic or sporadic tasks, several bandwidth preserving algorithms have been proposed [160, 86, 137, 161, 162]. The two common approaches for servicing soft deadline aperiodic requests, namely background processing and polling, do not give satisfactory result, but they provide a base for considering aperiodic tasks. The Priority Exchange (PE) and Deferrable Server (DS) algorithms [86] preserve some execution time allocated for aperiodic service and yield improved response times.

Unlike DS and PE, which periodically replenish their server execution time to full capacity, the Sporadic Server (SS) only replenishes its server execution time after some or all of the execution time is consumed by aperiodic task execution. By this, SS can provide almost the same schedulability as PE but at

an implementation complexity comparable to DS. Also the run-time overhead is lower because SS needs not to replenish the server capacity periodically as long as it is not consumed. For hard sporadic tasks, each is allocated a high priority server individually. SS can provide a guarantee as long as the aperiodic task's deadline is equal to or greater than its minimum inter-arrival time. If that is not true, other priority assignment and schedulability analysis coming from deadline monotonic should be used for the sporadic servers.

Different from PE, DS and SS, the Slack Stealing algorithm [84] does not create a periodic server for aperiodic task service. It rather creates a passive task (*Slack Stealer*), which attempts to make time for servicing aperiodic tasks by "stealing" all the processing time it can from the periodic tasks without causing their deadlines to be missed. At first, it computes a slack function for the periodic tasks. The slack function is updated by the real execution time of periodic tasks and addition of aperiodic tasks. Based on this function, it is easy to find the slack time piece for pending aperiodic tasks. This algorithm can be extended to guarantee firm aperiodic tasks [137]. Also it is natural to include resource reclaiming to make use of the idle time which is required by the worst case but is not used. The Dynamic Slack Stealer suggested in [35] tries to circumvent the disadvantage of the conventional Slack Stealer by computing the slack at run-time. This makes the algorithm applicable to a more general class of scheduling problems.

Similar to the fixed priority servers above, Spuri [161, 162] proposes several soft aperiodic task servers, namely Dynamic Priority Exchange, Total Bandwidth Server, Earliest Deadline Latest Server and Improved Priority Exchange Server. These servers can provide service to the soft aperiodic tasks in deadline-based scheduling algorithm (EDF). Buttazzo's algorithm [20] is an improvement to the original Total Bandwidth Server. All the above dynamic servers are for soft aperiodic tasks. When it comes to hard aperiodic tasks, the concepts of optimality and performance are quite different.

Priority Inversion due to Resource Sharing

The problem of resource sharing is tackled with a Priority Inheritance Protocol in [145, 8]. A *resource* is any software structure that can be used by a process to advance its execution. To control accesses to a shared resource, some synchronization primitives should be used. However, the direct application of them may cause uncontrolled priority inversion. To solve such a problem, Priority Inheritance Protocols (PIP) are used in fixed-priority scheduling: when a job J blocks one or more higher priority jobs, its critical section is executed at the highest priority level of all the jobs it blocks; after exiting its critical section, job J returns to its original priority level. Still, the basic priority inheritance

protocol has the following two problems: deadlock and chained blocking.

The Priority Ceiling Protocol (PCP) [145] can prevent the formation of deadlock and chained blocking. The Stack Resource Policy (SRP) [8] extends PCP in three essential points:

1. It allows the use of multiunit resources.
2. It supports dynamic priority scheduling.
3. It allows the sharing of runtime stack-based resources.

Whereas under the PCP a task is blocked at the time it makes its first *resource request*, under the SRP a task is blocked at the time it attempts to *preempt*. This early blocking slightly reduces concurrency but saves unnecessary context switches, simplifies the implementation of the protocol, and allows the sharing of runtime stack resources.

2.1.3 Dynamic Scheduling

Dynamic Planning-Based Scheduling

This kind of schedulers performs feasibility checks dynamically. A task is *guaranteed* by constructing a plan for task execution whereby all guaranteed tasks meet their timing and resource constraints. In a distributed system, when a task arrives at a site, the scheduler at that site attempts to guarantee that the task will complete execution before its deadline, on that site. If the attempt fails, the scheduling components on individual sites cooperate to determine which other site in the system has sufficient resource surplus to guarantee the task. If all fail, the task will be rejected by this system, leading usually to an "exception" to be issued.

In [136], an algorithm is described to guarantee nonpreemptable tasks arriving at a site given their arrival time, deadline or period, worst case computation time, and resource requirements.

Dynamic algorithms that do not *a priori* know the arrival times of tasks cannot guarantee optimal performance [38]. An algorithm is considered better than another if for some task sets the former can find a feasible scheduling while the latter can not.

With regard to cooperation between processing elements, several schemes have been reported in the literature [16, 61] and they differ in the way a site treats a task that cannot be guaranteed locally.

The variance in the tasks' execution times compared to their WCET may result in some tasks completing earlier than expected by the scheduler. Reclaiming

unused time to improve the schedulability of dynamically arriving tasks is the motivation behind the work in [148]. This is a form of slack stealing at run time.

Chetto [25] investigates the maximal idle time available for sporadic tasks for a processor with a set of hard periodic tasks. He turns to Earliest Deadline algorithm, defining EDS (earliest deadline as soon as possible) and EDL (earliest deadline as late as possible). Like As Soon As Possible and As Late As Possible, EDS and EDL give a bound to the available idle time in a super period. This bound gives the criteria for sporadic acceptance and is implemented by the algorithm in that paper.

Schwan's dynamic preemptive algorithm has its root in EDF [142]. By using two separate lists, slot list (SL) for the scheduled time slot and earliest deadline first list (EL) for tasks, Schwan provides an efficient dynamic algorithm for periodic and sporadic tasks with arbitrary arrival time, start time and deadline. When a new task comes, the slot list is checked to find out the earliest time to complete that task. If that time is greater than the task deadline, it is unschedulable and rejected. Otherwise, the SL and EL will be updated to reflect the new decision.

Dynamic Best Effort Scheduling

Best effort scheduling is the approach used by many real-time systems deployed today. In such a systems, a priority value is computed for each task based on the task's characteristics and the system schedules tasks according to their priorities. Confidence in the system is gained via extensive simulations, in conjunction with recoding the tasks and priority adjustment.

Often used real-time scheduling algorithms, such as, EDF and LLF work well as long as no overloads occur. They degrade extremely under overload. The best effort approach proposed in [93] tries to minimize this degradation. Many priority-driven preemptive scheduling and other methods are tested with this approach.

The biggest disadvantage of dynamic best effort algorithms lies in their lack of predictability and their suboptimality. For most real-world circumstances, optimal dynamic algorithms do not exist though. However, it is useful to quantify the worst case behavior of the dynamic algorithms. [10] analyzes such bounds for the problem of preemptively scheduling sporadic task requests in both uniprocessor and multiprocessor environments. It is proven that no on-line algorithm can exceed an upper bound compared to a unrealizable clairvoyant algorithm, in sense of schedulability.

Resource Allocation and Assignment

Resource allocation and assignment problem is discussed a lot in the multiprocessing community or a multiprocessor context [165] since any inappropriate operation can then cause a system deadlock. Sometimes, this problem is also considered as a communication and memory access problem, because the communication channel and memory are both resources too.

In section 2.1.2, we discuss the priority inversion problem due to the resource sharing. The solution to that is to add a priority inheritance protocol to prevent the low priority tasks from getting the resources.

Hegazy *et al* [55] consider an application model where timeliness requirements are expressed using benefit functions. They propose adaptation functions to describe anticipated application workload during future time intervals. Two heuristic algorithms are proposed to compute resource allocations in polynomial time. The objective is to maximize aggregate application benefit and minimize aggregate missed deadline ratio. A proportional shared resource allocation algorithm is proposed in [164] for realizing real-time performance in time-shared operating systems. Processes are assigned a weight which determines a share (percentage) of the resource they are to receive. Gertphol *et al* [47] introduce a novel performance metric to capture the effectiveness of a resource allocation. A MILP (mixed integer linear programming) based approach is developed to determine a resource allocation that has the highest metric value.

Other Scheduling Approaches

In [187], with only partial knowledge of the environment, a new dynamic scheduling method is proposed, which minimizes the maximal response time of the system. Therefore, it suits only for soft RT at most.

Communications between tasks are also considered in some papers. [121] discusses the bus allocation problem for distributed embedded systems. Four approaches are considered for scheduling of messages. They differ in the way the messages are allocated to the communication channel (either statically or dynamically) and whether they are split or not into packets for transmission. This is further extended in [123, 122]. For both tasks and communications, a time-table based static scheduling is used for the periodic case and a priority based scheduling is used for the sporadic case. In [87], a high-throughput communication network is considered, in which multiple time-division-multiplexed data streams are transferred over several parallel physical channels. A method is presented to guarantee the throughput for hard-real-time streams. Recently, many research works are done on Network-on-Chips [14, 185], but most of them assume a random traffic.

2.1.4 Summary

A key mission of the operating system is to manage the various resources available to it (main-memory space, I/O devices, processors) and to schedule their usage by the various active processes. Any resource allocation and scheduling policy must consider three factors: fairness, differential responsiveness and efficiency. Of course, when the system has hard RT constraints, time constraints always have the highest priority. For soft RT systems, timing is very crucial also to avoid deadline misses as much as possible.

A standard Operating System like UNIX uses a paradigm based on the combination of design-time and run-time scheduling: for some urgent IO, preemptive priority based scheduling is used; for normal processes, round-robin and time slicing are used [163]. The objective is to give high performance, fast response and a certain level of fairness.

Almost all modern RTOSs provide a strict priority-based preemptive scheduling mechanism in their kernel (e.g., Virtuoso, VxWorks and pSOS). The priority of each task is either fixed at design time or changed dynamically, but it is the responsibility of the user to assign the priorities, i.e., the user is supposed to implement his own scheduler on top of that kernel.

In our context, we need an OS with the following features:

1. It must be a RTOS to guarantee fast and deterministic response.
2. It provides a priority-based scheduling mechanism as described earlier to allow us to implement our scheduler. The number of priority levels should be high enough to allow each (sub)task to be assigned a different level. So a few dozen levels should be present at least.
3. It should support heterogeneous and multiprocessor platforms.

Also, we expect a good development framework to be provided for that OS. We have compared several available RTOSs and the result can be found in [168]. One of the main candidates is Virtuoso [172], on which our one demonstrator is based on [181] (see Section 6.1).

2.2 Low-Power and Cost Considerations

It has long been realized that we can reduce the system energy consumption by selectively shutting off or slowing down functional components which are idle or under utilized. This observation entails the Dynamic Power Management (DPM) and Dynamic Voltage Scaling (DVS) approaches. [13] and [68] give

good surveys on the design methodology and tools of low power system design and synthesis. Good tutorials about the physics behind it can be found in [23, 22]. The key equations governing power and performance are summarized in [113].

2.2.1 Dynamic Power Management

Whenever a system is in the idle state, DPM either shuts it down completely or swaps it into some sleep state to save energy. Advanced Power Management API provides a way to implement it. Since the state swapping between the normal and low-power states requires extra time and energy, it can increase the system response time undesirablely. The key issues of DPM are to decide whether to switch to a power saving state and to which one if yes. DPM makes decisions based on the activity history and its prediction to the future. How it does the latter distinguishes a DPM algorithm from another.

[12] and [11] give a good overview on DPM. The research groups of Prof. Benini at Bologna and Prof. De Micheli at Stanford have been very active in this field and many papers are published [27, 28, 97, 95, 94]. In [96], the authors apply the scheduling and DPM on device drivers and integrate it into RedHat Linux6.2 OS. Simunic *et al* [155] give an example to integrate DPM and DVS.

In [158], a power control (PC) method is added with the DPM approach to obtain further energy savings when the system is active. Different from former time-out, predictive or stochastic models, two new approaches are proposed to better model the system behavior for general user request distributions [156]. These approaches are event-driven and give optimal results verified by measurements.

2.2.2 Dynamic Voltage Scheduling

When the cost is system power or energy consumption, decreasing the supply voltage is lucrative to a low power design for CMOS digital circuits, because for current technologies and Vdd levels, the energy consumption of CMOS digital circuits is approximately proportional to the square of the supply voltage (for power it is cubic), though it will almost linearly slow down the cycle speed as well.

Traditionally, the CPU works at a fixed supply voltage, no matter whether the work load is high or low. Actually, when the work load is low, the fast speed of the CPU is unnecessary and can be traded for a lower energy/power consumption by reducing the supply voltage. Many papers have been published on this method and a good overview can be found in [129, 68].

Scheduling real-time systems with DVS has been studied extensively in various flavors, such as uniprocessor or multiprocessor, static or dynamic, etc.

Uniprocessor

Several papers were presented a few years before DVS was getting popular. Weiser [171] suggests a practical algorithm, PAST, to integrate DVS into a task scheduler by slicing the time into pieces and monitoring the past histogram. Simulation is done on some dumped real-life trace data. Govil [50] extends this work by developing several other algorithms, to better predict the work load of the current time slice. No deadline is taken into account in either work.

Yao *et al* [184] propose an $O(n \log^2 n)$ off-line optimal algorithm for single processor variable voltage scaling, minimum-energy scheduling. Only convex power functions can be handled. Two on-line heuristics were proposed as well: AVR (average rate) and Optimal Available. The former just computes the average rate for each task and accumulates them together for each time unit to find the speed. Then it schedules the tasks with EDF. The latter recomputes an optimal schedule for the problem instance consisting of the newly arrived job and the remaining portions of all other available jobs after each arrival. The effectiveness of AVR is studied in the rest of this paper. This work is the basis for later off-line DVS scheduling.

Off-line scheduling algorithms for non-preemptive hard real-time tasks are discussed in [57, 66]. In [60, 66, 114], more practical variable voltage processor models are used, assuming that processor voltage cannot change instantaneously or continuously, which makes the problem much harder to solve. A heuristic approach is described in [60], and a linear programming formulation is introduced in [66]. Yao, Demers and Shenker [184] have provided an optimal preemptive off-line real-time scheduling algorithm for a set of independent tasks with arbitrary arrival times and deadlines on a variable speed processor. This is extended in [111] to include switch overhead and discrete voltage levels. [76] extends the work to consider non-uniform load.

In [66], a discrete supply voltage is assumed and an ILP (integer linear programming) model is used for off-line scheduling. Some theorems presented in this paper look interesting:

- If a processor can use only a small number of discrete voltages, the voltage scheduling with at most two voltages minimizes energy consumption under any time constraints.

- Two voltages which minimize energy consumption under a time constraint are immediate neighbors to the ideal voltage.

Shin *et al* [152] add a simple extension to a normal fixed priority scheduler to allow either to slow down the processor, when there is only one eligible running task and its required execution time is less than its allowable time frame, or to shut down the processor, when it is detected that there is no task eligible for execution until the next arrival of a task. Then in [153], Shin *et al* extend it by adding a 2-phase voltage scaling. An off-line algorithm gets the basic voltage for a set of tasks scheduled with priority based scheduler, while the on-line algorithm refines the off-line decision: slow it down when there is only one task in the queue or shut it down when none is present in the queue. The on-line scaling can partly make use of the slack time. Kim *et al* [72] extend this work by adding a slack estimation heuristic.

Quan and Hu propose an off-line algorithm for fixed-priority real-time task set with arbitrary release time and deadline [129]. Two algorithms are given. The first one takes $O(N^2)$ time (N is the number of jobs) to find the minimum constant speed needed to complete each job, since constant voltage tends to result in a lower power consumption. The second algorithm, with $O(N^3)$ time complexity, is built above the first one and it gives two results: the minimum constant voltage (or speed) needed to complete a set of jobs and the voltage schedule. [130] continues the previous work. A theoretical limit in terms of energy saving is established. Two algorithms for deriving the optimal voltage schedule are provided. The second one is built above the first while it further reduces the computation cost.

Two on-line scheduling algorithms are given in [114, 115] when the processor voltage is not assumed continuously variable. The static voltage scheduling assumes that a task set is statically order-scheduled in advance (EDF, LDF...). Then a static voltage scheduler uses an ILP like approach to find the voltage for each task. At run time, two dynamic voltage schedulers are proposed: one assumes it knows the arrival times and deadlines in advance while the other does not. [115] is also a good tutorial on DVS.

Recently, several stochastical algorithms are proposed [159, 51]. In [159], Slacked EDF algorithm is proposed for an independent arbitrary task set. It is stochastically optimal and it tries to minimize the maximal lateness (therefore, not for hard RT) and energy consumption by DVS. An upper bound on the processor energy savings is also derived. Optimal RT scheduling of periodic tasks is given. [51] takes into account task variation, length and power demand with a statistical model. Then it schedules the "heavier" tasks later to benefit from the slacks possibly available from the previous tasks.

Chandrakasan[21] compares several energy saving tactics concerned DVS: *continuously variable supply voltage, workload averaging, quantized supply voltage and parallelism.* The latter three tactics or their combination can get as good a result as the first one. Pering [117] has set up a benchmark to evaluate sev-

eral OS based DVS algorithm. The two interval-prediction heuristic algorithms are found not good enough compared to the optimal solution. More thread or workload information other than rough estimation is needed for a better result. Grunwald *et al* [52] also evaluate several on-line clock scaling algorithms, which are based on the CPU utilization information and not yet on the application information, with an experimental pocket computer. Their conclusion is that currently proposed algorithms consistently fail to achieve their goal of saving power within the limit of not changing the interactive behavior of the user applications.

Intra-Task DVS Above we have discussed the slack time that comes from low processor utilization or execution time variation from the WCET. The former can be handled by static DVS, while the latter is partly solved by on-line DVS. Since these on-line DVS algorithms re-evaluate the working voltage at the boundary of tasks, we call them "inter-task" DVS. To further exploit the possibility of DVS, some researchers suggest to look inside the task and do DVS at a granularity smaller than task. We call this kind of algorithms "intra-task" DVS.

In MPEG decoding, the variance in execution time on frame basis can be very large. Simunic *et al* propose a stochastic model to predict the execution times for streaming multimedia applications on a frame-by-frame basis [155]. The prediction algorithm developed is then used as a part of a power control strategy that merges DVS and DPM.

A combination of hierarchical Finite State Machines and Synchronous Data Flow (SDF) actors is used in [82] to model the system. It schedules at the SDF actor level. For each data frame, a Runtime Execution Path Identification step is inserted to find out the actual execution path (similar to our scenario concept in section 3.3). When one actor finishes, the scheduler is recalled to make use of the slack time, which increases the runtime overhead a lot (some transformation such as presented in [36] may be used to improve it).

Lee and Sakurai [81] also consider the run-time DVS to fully utilize the slack time. They partition a task into several pieces, called *time-slots*, and then dynamically control the clock frequency and the supply voltage on "time-slot by time-slot" basis, depending on the feedback from the software execution.

In [7], the compiler generates some checkpoints at compile time. These checkpoints identify places in the code where the processor speed and voltage should be re-calculated. Checkpoints also carry user-defined time constraints. An available power profile, changing over time, is also included. It can handle the variation-slack-time. In [151, 170], the variation of the execution time to WCET due to the branch selection and varying loop iteration number is analyzed and utilized, with the help of some static timing analysis tool. At compile

time, the path dependent analysis results are stored. At run time, whenever a branch or loop causes the execution time variation, the speed is reset.

Multiprocessor

Though not as many as the single processor case, recently quite a few papers are published for multiprocessor DVS.

In [141], first a genetic GA (genetic algorithm) based algorithm is used to obtain the mapping and scheduling of a task graph. Then an off-line heuristic is applied to find the voltage, taking into account the power profile difference of each task. Similarly, [98] applies a GA based algorithm to find a valid mapping and scheduling. Then the tasks are swapped and shifted to explore the power profile or to provide a better DVS option.

In [99], for a multiprocessor and multiple link platform with a given task and communication mapping, a two-phase scheduling method is proposed. The static scheduling is based on the slack time list scheduling. A critical path analysis and task execution order refinement (rather inefficient) method is used to find the off-line voltage scheduling for a set of periodic real-time tasks. The run-time scheduling is similar to resource reclaiming and slack stealing, which can make use of the variation from the worst case execution time and provide the best-effort service to aperiodic tasks.

In [186], an EDF based multiprocessor scheduling and assignment heuristic is given, which is shown to be better than the normal EDF (though it can guarantee neither the feasibility nor the optimality). After this step, an Integer Linear Programming model is used to find the voltage scaling accurately or approximately by simply rounding the result from a Linear Programming solver. The result is claimed within 97% accuracy. Experiments based on TGFF are given to show the effectiveness of the proposed work.

Liu *et al* [92] use dynamic programming to solve the combined problem of partitioning, communication speed selection and processor voltage scaling. They use data of the XScale and LXT-1000 processors (both from Intel). Experiments are done with an automatic target recognition algorithm on a pipelined homogeneous multiprocessor model.

Almost all the above DVS approaches fix the mapping and ordering (even the voltage setting in some cases) of tasks ,found at design time. The on-line part, if it exists, is only responsible for refining the operating voltage. This is completely different from our approach, which allows dynamic task mapping, ordering and voltage selection.

Other Approaches in DVS

[29] proposes that contents providers should supply the information of the execution time variations in addition to the content itself so that the RT scheduler can use it instead of the WCET to estimate the workload accurately. It considers the frame-based, soft real-time applications. A generic computation cost model is used so that the same content specific information can be used for different processors. A similar approach can be found in [124], where the application informs the OS of its future workload to accurately schedule the voltage.

Im *et al* [65] propose to add a buffer to fully utilize the slack time. The goal is to determine the minimum buffer size needed to achieve the maximum energy saving. They exploit the slack time fully by buffering multiple input data or output results so that there is always at least one runnable task on the processor. It is intended for soft real-time applications. A fixed-priority algorithm is used to set the order first, then the tasks are checked one by one to utilize the slack time left before. The novel issue is the buffer size estimation. A similar example is given in [104]. A tradeoff between the buffer size and the latency is investigated. On- and off- line algorithms are developed to minimize the energy when the buffer size is limited and each application has a deadline constraint.

In [67], the processor is divided into several synchronous domains linked with asynchronous connections. This approach allows to apply DVS separately on each synchronous domain. Similar works can be found in [144, 143].

In [62], DVS is used with a stochastically guaranteed completion ratio. Key assumptions include that the deadline is always less than the sum of WCET and the probability of Best Case Execution Time is much bigger than WCET.

DVS is even used for communication connections [147]. By applying a history-based DVS algorithm to a voltage/frequency variable data link, the authors claimed a 3.2x average power saving at a low latency and throughput cost. The DVS algorithm part is synthesized in hardware and tested in a 2D 8x8 network, where random traffic is generated. This approach looks quite interesting and the result is quite promising for really long lines where the converter overhead is negligible.

Work at UCLA in Group of Prof. Potkonjak [57] presents an off-line scheduling algorithm for non-preemptive hard real-time tasks. Two schedule phases are iterated several times to find the order and the voltage: the first phase finds the schedule and the second phase finds the voltage. Each iteration of the two phases generates different solutions since the objective functions used in the first phase to guide the search strategy are randomized by a random offset. In [60], more general variable voltage processor models are used, assum-

ing that processor voltage cannot change instantaneously or continuously. It is based on EDF. Global design flow and system monitoring (including cache) are included. [59] presents an on-line algorithm which considers aperiodic (not known a priori) and periodic tasks together. An acceptance test is performed first for every aperiodic task. [58] puts a nonpreemptive scheduling heuristic into the frame of synthesizing a complete system, e.g., selection of processor core, determination of instruction and data cache size.

At UCLA several techniques have been developed to also handle QoS issues. In [126], the task scheduling to meet QoS constraints is performed for fixed voltages. Later papers [128, 127] have addressed how to determine the voltage and the schedule while minimizing energy within QoS constraints [128] or to determine the required system resources (including varying voltages) to meet QoS constraints [127]. These techniques however consider only independent tasks.

2.2.3 Battery Life Related

Generally speaking, all the low-power methods can prolong the battery life. Here we focus on researches where the battery life profile is taken into account explicitly.

In [100], the scheduler is a combination of RM and EDF and the tradeoff between the battery life and the service quality is considered. The scheduler prefers the more critical and less energy consuming tasks by combining these considerations into priority setting. It can not really save energy, though it can prolong the battery life.

A very interesting battery model is given by Rakhmatov *et al* in [132].

2.2.4 Physical Implementation of DVS

For low-power digital circuit design, please refer to [23, 22]. In [31, 49], some efficient DC-DC converters are described. It's the kernel device to implement DVS.

In [19], a real implementation of a DVS processor is presented. The system consists of a DC-DC switching regulator, an ARM V4 microprocessor with a 16-KB cache, a bank of 64-KB SRAM ICs, and an I/O interface IC. The four custom chips are fabricated in a standard 0.6-um 3-metal CMOS process. The system can dynamically vary the supply voltage from 1.2 to 3.8 V in less than 70us. This provide a throughput range of 6-85 MIPS (million instructions per second) with an energy consumption of 0.54-5.6 mW/MIP. The efficiency of the DC-DC converter ranges rom 90% at high voltage to 80% at low voltage.

[118] describes the software implementation, including both applications and the underlying operating system. A simple thread based EDF DVS algorithm is used. Experimental results and scheduling overhead are given. This provides a good example of the underlying process technology and circuit assumptions for DPM and DVS.

2.2.5 Other Approaches

In [1] a speed-setting policy is applied based on a system model that correlates clock speed with best case, average case, and worst case sustainable frame rates, accounting for data dependencies in multimedia streams. It claims that energy can be saved by only frequency scaling, even at a constant supply voltage, because for a fast CPU many cycles are wasted to wait for the slow external hardware (e.g., off-chip memory).

Kadayif *et al* use the SimplePower simulator to profile the energy consumption [70, 71]. They assume a homogeneous multiprocessor platform and try all the possible number of processors for each loop nest. After the profiling stage, an ILP based optimization is triggered to find the real number of processors used for each loop, then corresponding processors are activated and deactivated at run-time, though the decision is done at compile time. Some overheads, e.g. re-synchronization, are included.

Lee *et al* [83] have recently proposed a hot-swapping extension to the Rate Monotonic scheduling. They use several implementations, each with different power and execution time pairs (not Pareto optimal though), and one of them is selected at run-time. In that sense it partly resembles our operating point selection. At run-time, for each swap request (received or generated separately), the scheduler checks the feasibility of swapping and makes the swap on-the-fly when possible.

[75] assumes a multiprocessor architecture, each of which has different performance/energy tradeoffs but shares the same instruction set (different versions of Alpha processor in this paper). The exact processor selected for the execution of a task is decided at run time, either at time intervals or task boundaries, aiming at reducing energy or power-delay product. A very low overhead for the core switching is reported. [56] also considers this kind of activity migration (hopping between duplicated units), but to reduce the power density and the temperature.

Observing that the OS routines have a rather low instruction level parallelism, the authors of [89] suggest to tune down the processor parallelism (e.g. from 8-issue to 1-issue) for these routines to save energy.

Work at Princeton in group of Prof. Jha All the works at Princeton up till very recently have focussed on fixed voltages. COSYN [34] is a heuristic-based constructive co-synthesis tool. It takes periodic acyclic task graphs and a Processing Element library as input, then generates a low-cost heterogeneous distributed embedded-system architecture, meeting real-time constraints. For scheduling, it employs a combination of preemptive and nonpreemptive static, list scheduling philosophy based method. With some changes in task clustering and cluster allocation, it can take low power into consideration in a limited way. The other works coming from the same research group include MOCSYN [41], CORDS [39], MOGAC [40] and COHRA [32, 33]. In MOCSYN and MOGAC, a multiobjective, genetic algorithm based co-synthesis framework is used to give cost-performance trade-off. MOCSYN uses a preemptive static critical path heuristic scheduling algorithm, while MOGAC applies a nonpreemptive slack-based list scheduling algorithm. The costs can be chip area, power, etc., or a combination of them. Hard real-time constraints are considered in the co-synthesis approach. CORDS integrates the reconfigurable FPGA into the co-synthesis approach and it uses a preemptive static critical path scheduling algorithm with dynamic task reordering based on FPGA reconfiguration time. In the hierarchical co-synthesis approach, COHRA, a combination of preemptive and non-preemptive static scheduling is employed. Recently, the DVS technique is also integrated into their co-synthesis work [98, 99].

Our methodology also aims to reduce the cost or power of a system, but it is different from the above works mainly in several issues. Firstly, it is separated into the design time and the run time phases. The design time phase is to provide as broad design space as possible, while the run time phase is to allow enough flexibility in the system at a low overhead. Secondly, we apply the concept of Pareto-optimal set [116], where every solution is better than any other one in at least one direction (e.g. either it consumes less energy or it runs faster). This allows us to consider many different cost functions, not only the power, in a generic way. Thirdly, all the task mapping and ordering (and voltage selection if applicable) are done dynamically at the run time phase. Forthly, we introduce the concept of scenario to catch the dynamic feature of the applications.

2.3 Platform and Simulation Framework

In this section we only list some work interesting to us. It is not our intention to be complete here. They are only elements or tools reused from external sources that supply information or support for our methodology and do not constitute our main focus.

Energy and performance of platform components are in many cases co-estimated.

The execution of a system is first profiled with a simulator or other high-level profiler. The collected data is then used to both estimate the energy consumption and the performance. In accordance with this approach, we have merged the overview of the related work on performance and energy estimation of SoC or SiP (system in package) components. We will explicitly indicate which environments focus on performance only. In section 2.3.2, specific timing analysis methods applicable at the thread node level are discussed.

2.3.1 System-level Performance and Energy Models

In the context of system-synthesis, both industrial and academic co-design tools allow to simulate multiple processors (see [46] for an overview). The main focus of these tools is the generation of the communication interface between the components. They provide a scalable, modular and flexible environment to co-simulate the hardware/software of heterogeneous target architectures. This allows to verify the functional correctness of the generated interface and to some extent estimate the performance of the system.

Several researchers have added power estimation modules to these co-simulation frameworks. In [45], the authors describe energy metrics which help the designer to steer hardware/software partitioning. The energy consumption of the hardware modules is estimated based on register-transfer level description of the ASIC. Software energy consumption is estimated at the instruction-level (see [167] or [18]). The authors of [90] and [48] propose an energy/performance estimation framework for a single CPU with a cache memory hierarchy. In this framework the energy and performance of each component is individually evaluated. Only a limited interaction between the components can be taken into account. In [157], a methodology is presented for cycle-accurate simulation of energy dissipation. The methodology combines performance and energy estimation models of each system component into a single cycle-accurate instruction-level simulator. Since the interaction between the components affects directly both the performance and energy consumption, their integrated approach is accurate within 5% although crude (but fast) energy models of the components are used. A similar approach is followed in [77]. The energy consumption of the embedded software is estimated using an enhanced instruction set simulator. The power consumption of the remaining hardware is simulated with either a gate-level or register-transfer level simulators that reports power consumption on demand at the cycle-level accuracy. These tools are used to improve the performance of the embedded software or to tune the hardware parameters (e.g. cache size, communication parameter trade-offs). Compared to these authors, we focus on a design with heterogeneous multiprocessor target architectures.

In the context of scientific computing community, multiprocessor simulators are used for a long time. They allow to perform architecture studies in a reasonable time. Rsim [63] is developed to study the influence of superscalar processing nodes on the performance of shared-memory multiprocessor architectures. Another example is SimOS[139]. This simulator is used to characterize new architectural designs and to steer the development of operating systems and to evaluate the performance of applications. Although embedded platform-based design involves similar problems as the ones tackled in the scientific computing domain, the design trade-offs are different. In the embedded system domain energy consumption plays an important role besides the performance of a system. This axis adds extra complexity to the design of a system.

In section 4.2 of [154], the author describes how a cycle-accurate energy consumption simulator is implemented. They model each component typical in embedded systems with equivalent capacitance for each of its power states. Energy spent per cycle is a function of equivalent capacitance, current voltage and frequency. The equivalent capacitance allows them to easily scale energy consumed for each component as frequency and voltage of operation change. On each cycle of execution, the ARMulator sends out the information about the state of the processor ("cycle type") and its address and data busses. Based on these data, the energy consumption of that cycle is computed. Our current simulator takes a similar approach.

Almost all the frameworks are either cycle-accurate, which is accurate but slow, or analytical, which is fast but not very accurate. Recently, [69] proposes a middle path between the previous two approaches. They use probabilistic models for components, customized with application behavior. The models are at a higher level of abstraction which results in faster simulation, at the expense of slightly reduced accuracy. The methodology can handle multiprocessor systems containing processors and application-specific hardware.

2.3.2 Timing Analysis and Simulation

Timing analysis results are needed for the thread node characterization in the TCM approach. It is related to performance estimation in the sense that they both intend to estimate the execution time, but timing analysis tends to do that formally at a higher level, with some computation model, e.g., Petri Net. A good review on timing analysis can be found in [64]. Here we only give a short discussion on the most related works.

In embedded systems, the timing consumption is state and input data dependent. Formal analysis of such dependencies leads to intervals rather than a single value. [173] presents an approach to analyze the process behavior using intervals, exploiting program segments with single paths and taking the execu-

tion context into account. This is a potential way to perform our thread node level timing analysis, especially when combined with the ideas of [69] .

A two-step approach is proposed in [79]. The first step applies a per-task data analysis to get a table of useful cache block, which is believed in this paper to be the cache-related preemption delay additional to WCET. The second step uses a linear programming method to get the estimation of the whole task set. The method is explained and experimented for the direct-mapped instruction cache. Though extensions to set-associative and data cache are given, the effectiveness and accuracy are doubtful.

Richter *et al*[138] propose a timing analysis frame for the heterogeneous platform. It is based on static analysis and somehow conservative because it has to use WCET estimation. A similar approach can be found in [103].

Chapter 3

Model and Methodology

The key to understanding the reasons behind TCM is to realize that the only way to handle very dynamic applications in a resource efficient way is to postpone some design decisions till the run-time stage. Fixing everything at design time, according to the worst case estimation of the system, will result in either an over-dimensioned power hungry platform or a poorly performing application most of the time. But the minimal amount of effort should be done at the run time to minimize the overhead. Hence defining the best position of the interface is a key issue.

3.1 Overview of the TCM Flow

As initial step, designers specify an embedded application at a *gray-box* abstraction level, which is a Multi-Task Graph (MTG) model combined with high-level features of a Control-Data Flow (CDFG) Graph model [166]. The specification represents concurrency, timing constraints, and interaction at either an abstract or a more detailed level, depending on what is required to perform good exploration decisions later. Yet, to keep the complexity at an acceptable level, it does not contain all implementation details. Within this specification, the application is represented as a set of thread frames. Non-determinism behaviors can occur only at the boundary of thread frames. Each thread frame is a piece of code performing a specific function, which is partitioned into thread nodes, the basic scheduling units.

The purpose of the methodology is to determine a cost-optimal (e.g. energy consumption, deadline miss rate) constraint-driven (e.g. throughput or latency) scheduling of various thread nodes on a set of homogeneous or heterogeneous

processors. Different kind of processors normally execute the same thread node at different speeds and with different energy consumptions. These differences make it possible to explore the energy-performance trade-off at the system level.

The methodology is depicted in Figure 3.1, and it comprises several steps:

Gray-box model extraction as explained previously.

Concurrency improvement Transformations on the gray-box model are applied to improve the concurrency and to increase the opportunities for optimization. This step is out of scope of this thesis.

Scenario characterization To avoid worst-case implementations of the application, the dynamic behavior of each thread frame is broken into several design-time analyzable scenarios.

Design-time scheduling is applied to every scenario of each thread frame. Different from traditional design-time scheduling, it does not generate a single solution but a Pareto curve consisting of different scheduling energy-performance trade-off points.

Run-time scheduling An application-specific run-time scheduler is integrated in the application on top of an RTOS. At run time, this scheduler identifies for each active thread frame the running scenario. On each corresponding Pareto curve, it selects one trade-off point to minimize the global energy consumption of the application and to satisfy the global timing constraints by finding a reasonable execution time distribution among all the active scenarios. This selection is done by applying a fast heuristic described in Chapter 4 and Chapter 5. This allows to reduce the performance overhead of the run-time scheduler as much as possible, without relevant penalty on the result quality.

In fact, this flow can be further extended to include the task level data access and memory management [106], which is however not the focus of this thesis. In the sections below, we will discuss the gray-box model, the scenario selection, the design-time and the run-time scheduling, especially with simple solid examples. Readers are referred to [174] for further discussion on the modeling part.

3.2 The Gray-box model

Traditionally, in the real-time community, researchers tend to represent an application as task graphs and look tasks as *black boxes*. The internal operations

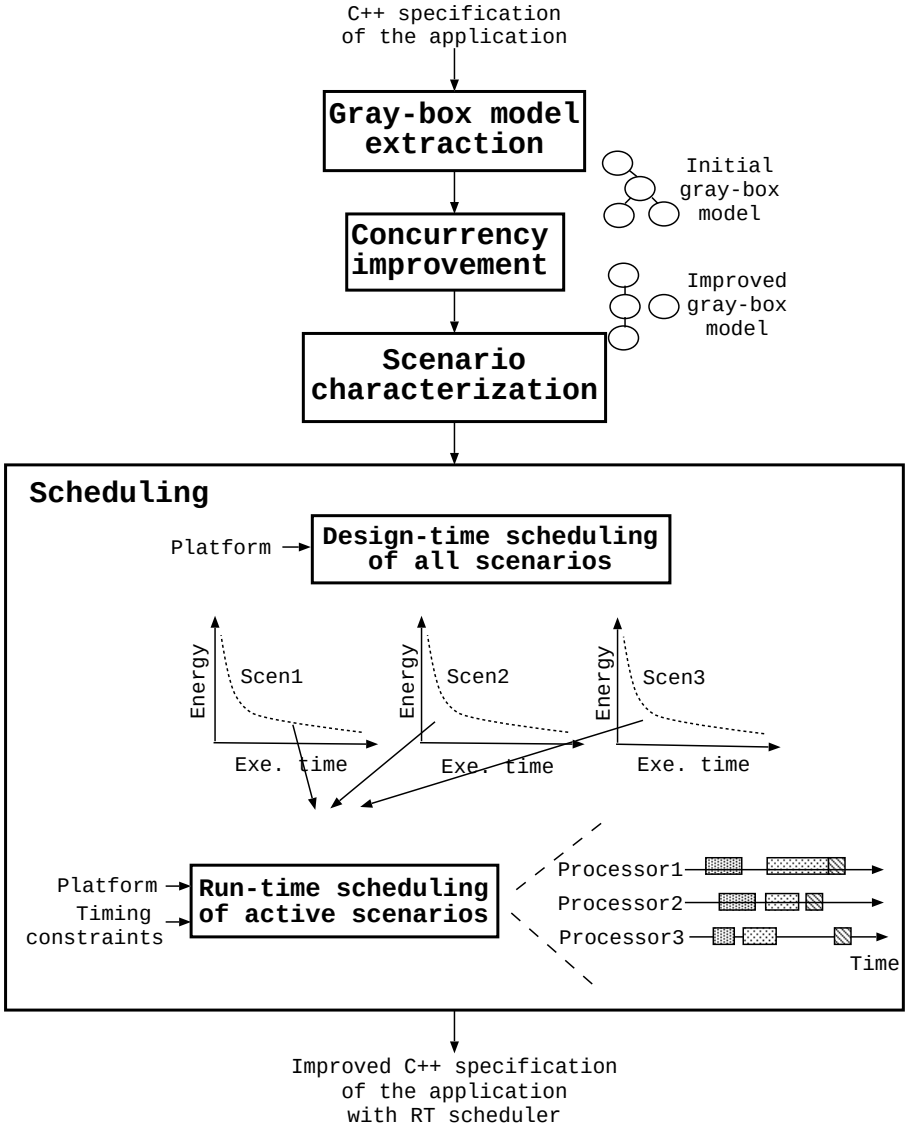


Figure 3.1: **Overview of our TCM design flow.**

and data structures of tasks are not exposed to the designers [166, 145, 135, 4]. This abstraction is at too high a level and it does not allow to expose some important information (e.g. the number of iterations of a loop) inside the tasks. In contrast, in the embedded system community, many papers focus on *white-box* task descriptions [15, 9], which are at the operation level (e.g. CDFG). This level is too low and has too much unnecessary details, which are typically unavailable at the early design stage. Instead, we use a *gray-box* model which is an abstraction level between the above white- and black-box models [125].

The gray-box model distinguishes the important details from the unimportant ones. However, what details are considered as important depends on the user and the application. The name "gray-box" denotes that the model is neither a black view nor a white view of the application. That is, some low-level and important details are visible while the irrelevant details are hidden in a black box. The gray-box model is by itself an important research topic and it is beyond the scope of this thesis to fully cover it. For an up-to-date description of the gray-box model, please refer to [149].

We use gray-box model to capture the concurrency, control flow, synchronization and data communication applications explicitly. Basically, applications are constructed from Thread Nodes (TN) and Thread Frames (TF).

Definition 3.1 (Thread Node) *A thread node T is a maximal set of connected operations with a deterministic execution latency $\Lambda(T) = [\delta(T), \Delta(T)]$. $\delta(T)$ and $\Delta(T)$ are the minimum and maximum execution time of the thread node T respectively. Associated with thread node T is the CDFG representation of this set of operations.*

Definition 3.2 (Thread Frame) *A thread frame is a maximally sized piece of functionality capable of running without intervention from the run-time scheduler and having a single thread of control. A thread frame has exactly one entry control port but it may have multiple exit control ports. A thread frame can have any number of data ports associated to it.*

Applications are represented with a two-layer model. The TN is the atomic scheduling unit of our design-time scheduler. TNs represent the intra task detail and function like the conventional white-box model. Every thread node has its associated CDFG representation, but this representation is used only when we profile the behavior of the TN. When scheduling is considered, either at the design-time or at the run-time stage, it is abstracted as a node with its minimum/maximum execution time. Therefore, it has an abstraction level higher than the node of conventional white-box models. Sometimes the primitive TNs are too fine grained. In that case, they can be further clustered into scheduling thread nodes, which are more coarse grained and are atomic to the design time

scheduler [174]. In later parts of this thesis, unless mentioned explicitly, we use TN to alternate the scheduling TN. The TN abstraction reduces the overhead of the design-time scheduler because it now does not look into the node and schedule at the operation level.

TFs are composed of TNs. By definition, the non-deterministic behaviors can only happen at the boundaries of TFs. Basically this means that inside a thread frame no dynamic task creation, event handling, resource contention or synchronization exists. All these things can only happen at the borders of TFs. To find the detail how these behaviors are modeled and depicted, please refer to [150].

For every TF, the design-time scheduler is called to find the Pareto curve of that TF and store the result in a predefined format. At run time, every TF is considered as an atomic scheduling unit by the run-time scheduler [1]. This approach dramatically reduces the number of objects the run-time scheduler has to handle. Hence, the run-time scheduling overhead is reduced. Besides reducing the run-time overhead, making the thread frame maximally sized also increases the design space explorable to the design-time scheduler, which normally leads to better system scheduling.

Consider the piece of simple C-like pseudo code below.

```
int in[], out1[], out2[], out[]; /* shared variables */

main() {
  create_task(tf_1); /* thread frame 1 */
  create_task(tf_2); /* thread frame 2 */

  readin_buf(&in); /* read in data to be processed */

  /* the following two lines can be executed in parallel */
  start_task(tf_1);
  start_task(tf_2);
}

/* thread frame 1, video decoding */
tf_1() {
  float c1, c2; /* local variables */

  tn_1(in, &c1, &c2); /* thread node 1 */

  /* due to data dependency, tn_2 and tn_3 can only start */
```

[1] There is ongoing research in IMEC to break that boundary partially to further optimize the scheduling result.

```
  /* after tn_1 finishes. however, tn_2 and tn_3 can be   */
  /* executed in parallel.                                */
  tn_2(in, c1, out1); /* thread node 2 */
  tn_3(in, c2, out2); /* thread node 3 */
}

/* thread frame 2, audio decoding */
tf_2() {
  int buf[]; /* local variable */

  /* tn_A and tn_B can be executed only sequentially */
  /* due to data dependency.                         */
  tn_A(in, buf);  /* thread node A */
  tn_B(buf, out); /* thread node B */
}
```

This is an example of an application comprising of two TFs, each of which has 3 and 2 TNs respectively. The gray-box model is depicted in Figure 3.2.

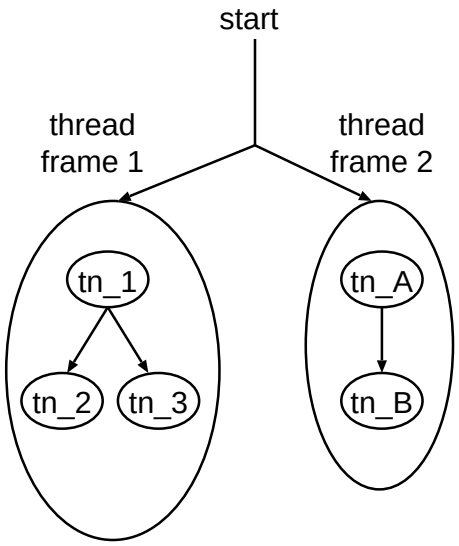


Figure 3.2: The gray-box model of a simple example.

Note the two levels of parallelism: the parallelism between TNs inside a single TF (tn_2 and tn_3 of thread frame 1) and the parallelism between TFs (thread frame 1 and thread frame 2). The former is handled by the design-time scheduler, whereas the latter is managed by the run-time scheduler. The run-time

scheduler is also responsible for managing the non-determinism caused by event handling, task creation, resource contention and synchronization. The behavior of an application many times is also determined by the input data. This can be handled by the scenario concept and is explained in the next section.

3.3 Scenario Selection

In many cases, dynamic behaviors will occur inside a TF because of input data dependencies. For example, in the MPEG-2 video decoder, depending on the image frame type, a different decoding engine will be used. Three types of frames exist: I (intra) frame, P (predicted) frame and B (bidirectional predicted) frame. I frames are simply coded as still images, not using any past or future information and can be decoded independently; P frames are constructed from the most recently reconstructed I or P frames; B frames are predicted from the closest two I or P frames, one in the past and one in the future. The computation powers needed for decoding these frames are significantly different, and quite different TNs have to be executed correspondingly. Also, some data-dependent loop boundary (while loop) can exist inside a TF. A TN can be executed, for example, from 10 to 100 times depending on the input data. Obviously, one single Pareto curve is not enough to represent the behaviors of these widely different situations, or at least it is not cost-efficient to do that (because of the worst case assumption). To capture the data-dependent dynamic behavior inside a TF, we introduce the scenario concept.

In our case, a scenario represents a combination of TF behaviors (for different data-dependent parameter sets) that have a closely resembling Pareto curve. It is extracted also based on the profiling data indicating which combinations occur very frequently. It does not make much sense to provide separate scenarios (and Pareto curves) for a set of situations that seldom occur, even if they have widely spread Pareto curves. The simple worst case consideration is enough. Due to the rather low chance of occurrence, further refinement will not have big impact on the system performance. It should be clear that by representing an entire set of behaviors with a single Pareto curve we lose some of our optimization potential. This single Pareto curve should indeed reflect the worst-case behavior of the set it represents to ensure real-time constraints. Nevertheless, the clustering of the entire set into one scenario gives nice (controlled) trade-offs between optimization potential and the overhead to store and scan the scenarios at run-time.

An example is given in Figure 3.3, where we have five TNs, 1 to 5, two conditional bypasses (1 to 3 and 3 to 5) and a loop over node 4. Only 4 execution sequences are possible at the TN level,

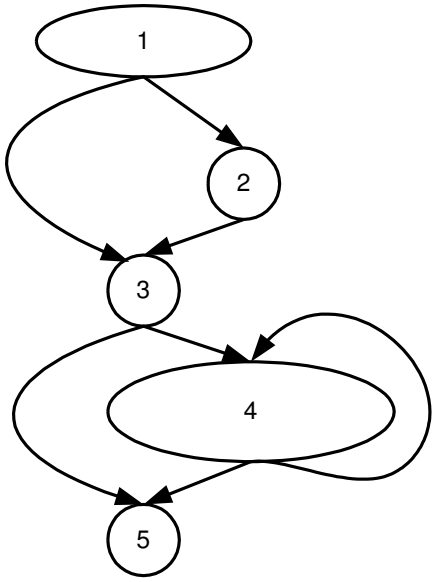


Figure 3.3: A simple example of the scenario.

1. 1, 3, 5;
2. 1, 2, 3, 5;
3. 1, 3, 4...4, 5;
4. 1, 2, 3, 4...4, 5.

If TN 2 needs much less computation power and consumes much less energy compared to TN 1, 3 and 5, the first two execution sequences can be merged and represented by a single Pareto curve. Actually, it is the Pareto curve of sequence 2. If the loop boundary over TN 4 has a big variation, for instance, from 1 to 30, and the execution time of TN 4 is comparable to the other nodes, it will be better to split the third execution sequence into 2 (or even more) scenarios according to its loop range (e.g, 1-10 and 11-30). Another possibility would be that since TN 2 is executed in sequence 4, the loop range over TN 4 would be affected and be limited to, e.g., 2-6. Therefore no split is necessary in that case. At the end we obtain only 4 scenarios: one for the first two execution sequences combined, two for the third sequence split and one for the last sequence. Note that the number of potentially different data-dependent parameter combinations would be $30 * 2 + 2 = 62$. Clearly such an explosion is

avoided by the way we generate scenarios, and each of them rather accurately depicts a representative pattern of the application.

For each scenario, we apply our design-time scheduler to get a separate Pareto curve, which represents the trade-off characteristics of that TF much more precisely. During the actual execution, the run-time scheduler will check the input data, select the active scenario based on the actual input data, and schedule it with all the other TFs. Apart from these intra-TF scenarios, we can also define inter-TF scenarios which predict the behavior of cluster of TFs. Similarly, some typical execution patterns of these clusters can be found and predicted for representative input streams.

3.4 Two-phase scheduling

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

As shown in section 3.2, we model applications with TNs and TFs. The design-time scheduling is applied on the thread nodes inside each thread frame at compile time, including a processor assignment decision of the TNs in the case of multiple processing elements. On different types of processors of a heterogeneous platform, the same TN will be executed at different speeds and with different costs, i.e., energy consumption in this thesis. These differences provide the possibility of exploring the cost-performance tradeoff at the system level.

The idea of our two phase scheduling can be illustrated with the simple example in Figure 3.2. Here we assume a dual-processor platform. For the five thread

	processor 0					processor 1				
	1	2	3	A	B	1	2	3	A	B
exec. time (us)	10	30	15	20	32	20	60	30	40	64
energy (uJ)	30	86	41	75	90	8	22	10	19	23

Table 3.1: The execution time and energy consumption of TNs in Figure 3.2

nodes in that example, we assume they have different execution times and

energy consumptions on different processors. The numbers are summarized in Table 3.1.

Now for every TF, the design-time scheduler will try different mapping and ordering of the TNs of that TF, satisfying all the dependency and time constraints. An example is given for TF 1 in Figure 3.4, where the execution time and energy consumption are shown also. When TN 2 and TN 3 are assigned

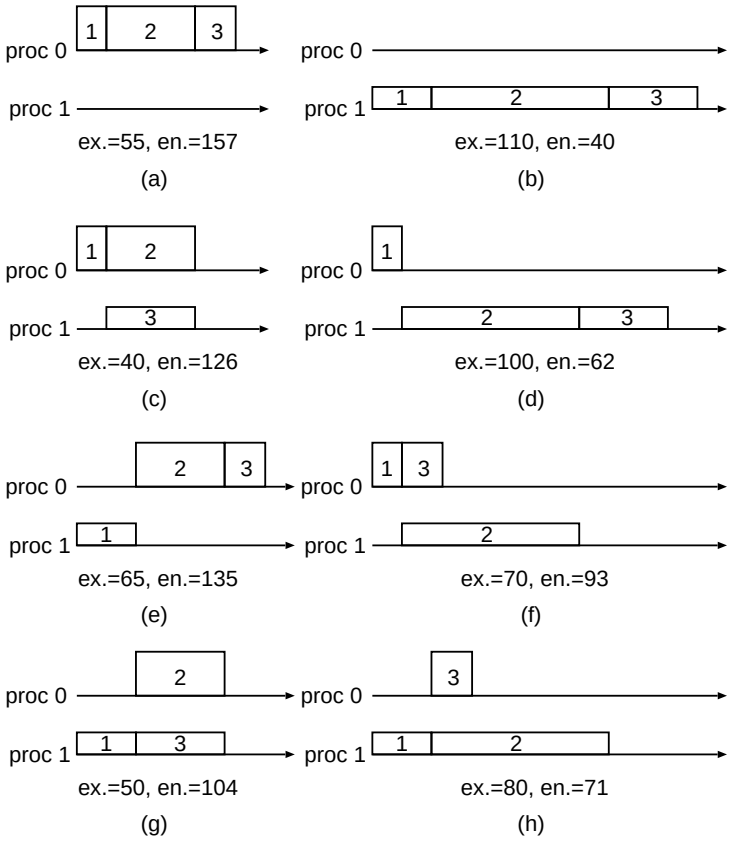


Figure 3.4: The design time scheduling of thread frame 1.

to the same processor (e.g. (d)), it makes no difference which one has to be executed first. For simplicity, we show only one possible order. However, extra constraints may exist and they will further fix the order. The design-time scheduling result can be represented as a Pareto curve and it is shown in Figure 3.5. From that figure, we can see that not all scheduling decisions are beneficial. For instance, (a) and (e) neither run faster nor consume less energy

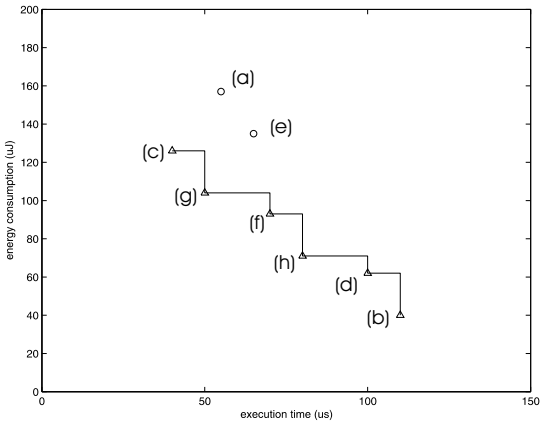


Figure 3.5: The Pareto curve of thread frame 1. Scheduling (a) and (e) are not on the curve.

compared with all the other schedulings. We say they are dominated or they are not on the boundary of a Pareto curve. Similar results can be obtained for thread frame 2 (see Figure 3.6 and Figure 3.7).

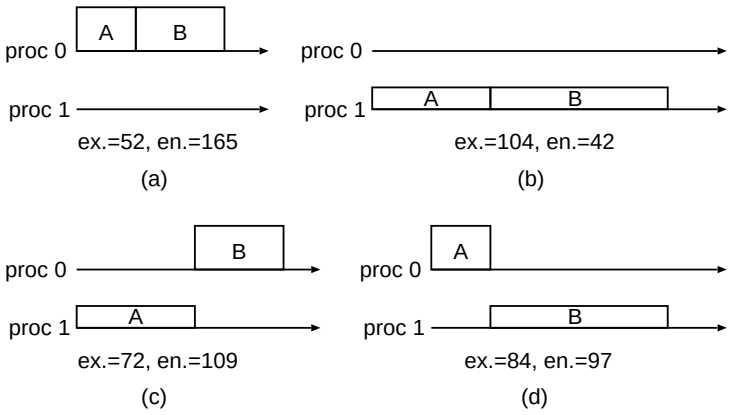


Figure 3.6: The design time scheduling of thread frame 2.

Here we illustrate only a simple example. With the increase of the number of TNs/processors, complex inter-TN dependencies and time constraints, it becomes impractical to do the design-time scheduling by hand. Therefore an automatic tool, known as the TCM design-time scheduler, is needed [174].

Only at run time the system level information will be complete. Given the

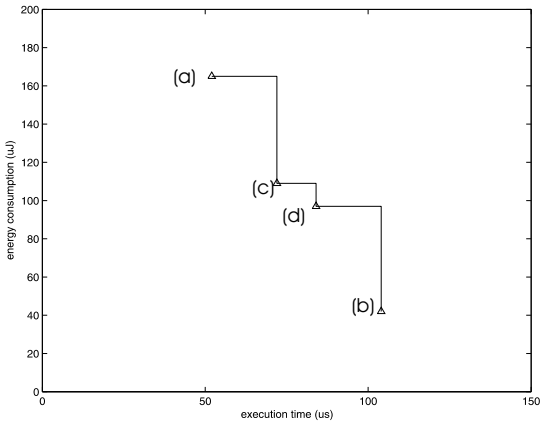


Figure 3.7: The Pareto curve of thread frame 2.

number of TFs, the Pareto curve of each TF and system constraints such as the global deadline, the run-time scheduler will select a mapping and/or ordering decision pre-computed by the design-time scheduler for every active TF and combine them together to get the system scheduling. For the above example, when the global deadline is 125us, the run-time scheduler will select design-time scheduling (g) for TF 1 and (c) for TF 2, combine them together and find the system scheduling with the minimum energy consumption. The main goal of this thesis is to solve the problem of how to find the global scheduling and how to support it with implementable run-time systems on real platforms.

Given a thread frame, our design-time scheduler will try to explore different assignment and ordering possibility, and generate a Pareto-optimal set [116], where every point is better than any other one in at least one way, i.e., either it consumes less energy or it executes faster. This Pareto-optimal set is usually represented by a Pareto curve. Since the design-time scheduling is done at compile time, computation efforts can be paid as much as necessary, provided that it can give a better scheduling result and can reduce the computation efforts of run-time scheduling in the later stage. However, if very data-dependent behavior is present inside the TF, the design-time exploration still has to assume worst-case conditions to guarantee hard real-time requirements. In order to improve the match with the real behavior even more, we have introduced the scenario concept as already discussed in section 3.3.

At run time, the run-time scheduler will then work at the granularity of thread frames. Whenever new TFs are initiated, the run-time scheduler will try to schedule them to satisfy their time constraints with an aim to minimize the system energy consumption. The details inside a thread frame, like the execu-

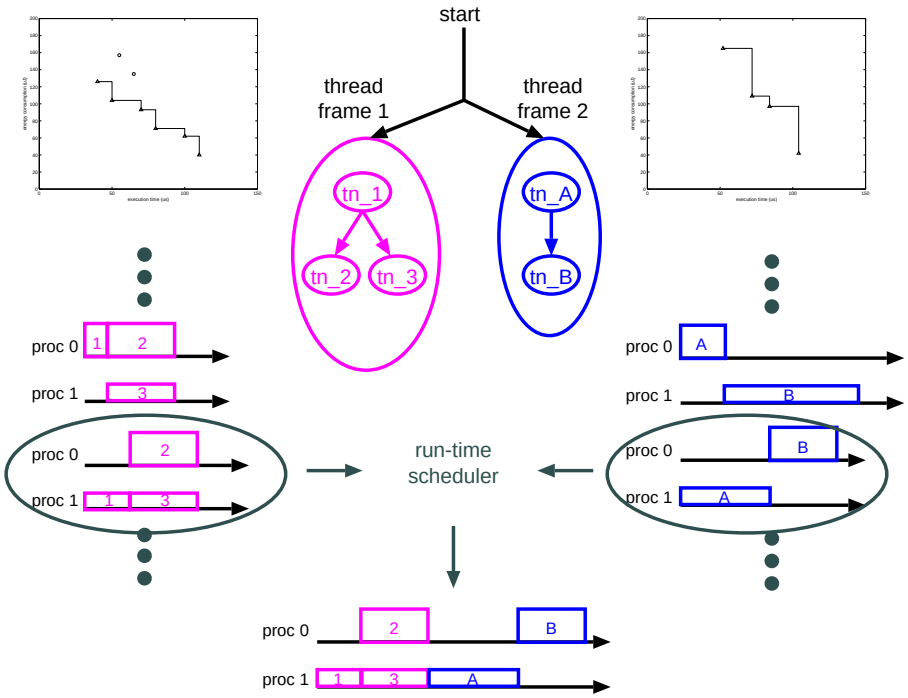


Figure 3.8: When the global deadline is 125us, the run-time scheduler selects design-time scheduling (g) for TF 1 and (c) for TF 2, combines them together and finds the system scheduling.

tion time or data dependency of each thread node, can remain invisible to the run-time scheduler and this reduces its complexity significantly. Only essential features of the points on the Pareto curve will be passed to the run-time scheduler by the design-time scheduling results, and will be used to find a reasonable cycle budget distribution for all the running thread frames.

In summary, we separate the task scheduling into two phases, namely design-time and run-time scheduling, for three reasons. Firstly, it gives more run time flexibility to the whole system. We can indeed accommodate more unforeseen demands for more execution time by any TF, by "stealing" time from other TFs, based on their available Pareto sets. Secondly, we can minimize energy for a given timing constraint that usually spans several TFs by selecting the right combination of points. Finally, it minimizes the run time computation complexity, which reduces the energy and time penalty so that faster reaction time can be achieved (up to 1ms). This is needed for modern multimedia and wireless communication applications. The design-time scheduler works at the

gray-box level but still sees quite a lot information from the global specification. The end result hides all the unnecessary details and the run-time scheduler can operate mostly on the granularity of TFs, not single TNs. Only when a large amount of slack is available between the TNs, a run-time local refinement on the TF schedule points can result in further improvements.

This methodology can in principle be applied in many different contexts as long as Pareto-curve like tradeoffs exist. For example, in the context of DVS, the cost can be the energy consumption. Thus our methodology results in an energy-efficient system. When the cost is energy and the horizontal axis is replaced by the quality of service (QoS), the problem becomes the energy minimization with a guaranteed QoS, as e.g. formulated in [128]. Also the deadline miss rate can be optimized in soft hard real-time applications (e.g. video decoding) for a given platform and a set of deadlines (see section 7.4 for experiment results).

Chapter 4

Fast and Scalable Run-Time Task Scheduling

As explained in Chapter 3, a run-time scheduler can efficiently explore the design space and make system level tradeoff according to the dynamic context. For that sake, a fast and effective heuristic is needed. In this chapter, we first motivate the problem with a simple example, then the problem is formulated and a greedy heuristic is described. After that, experimental results on both randomly generated and real-life applications are explained.

4.1 Motivational Example

The advantages of our novel approach can be shown by an example generated by Task Graph For Free (TGFF) [42]. Figure 4.1 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 Table 4.1, in which the first row shows

	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 4.1: Task graphs generated by TGFF.

the number of nodes in each task graph, the second row is the number of arcs,

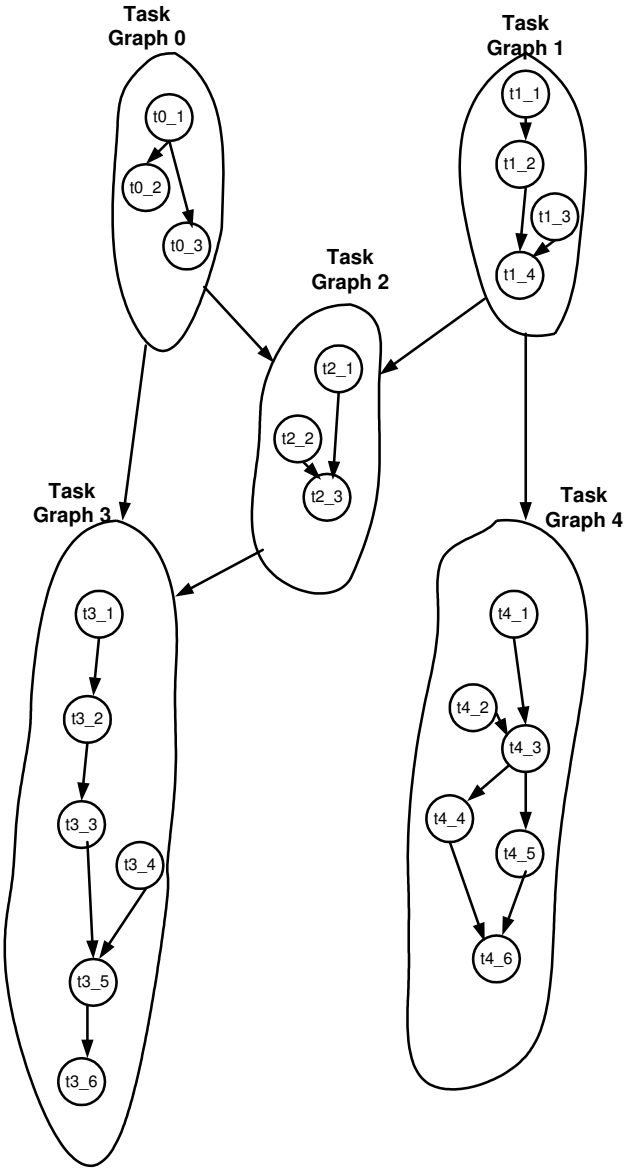


Figure 4.1: The task graph of the motivational example.

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 application domain (e.g. mp3, video decoding or the physical layer of a wireless modem). 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 example, 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 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 state of the art on-line inter-task DVS algorithm [152], 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 Table 4.1. 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 [58], 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 technique 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 [31]). 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 thread nodes and scheduled at that granularity. Thirdly, we schedule it in two phases, as already explained in Chapter 3. At design time,

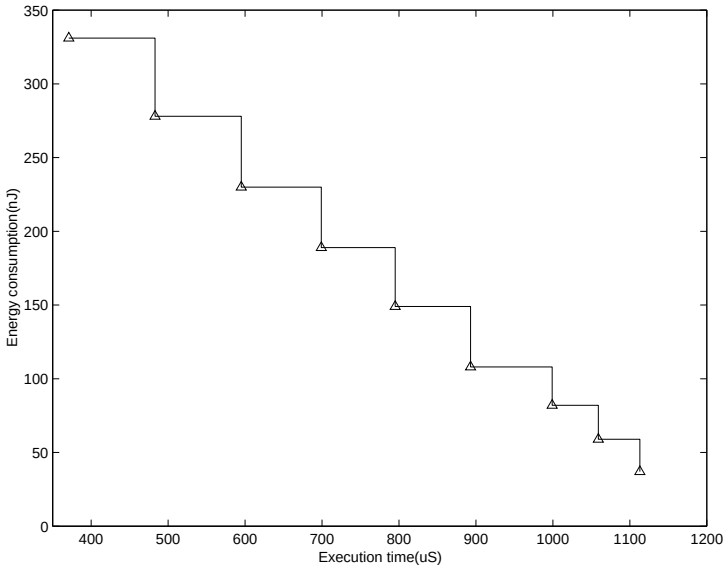


Figure 4.2: The Pareto curve of task graph 2.

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 Figure 4.2. At run time, knowing the active (scenario-based) 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 time frame when we schedule the application. This is realistic for real-life multi-media applications by inserting extra code to extract the necessary information (i.e. the scenario that is currently present), at the start of each frame (see e.g. [181]). If it is impossible to do so, a guess should be made at the start of the task and then the run-time scheduling decision should be re-evaluated as soon as that information 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 (up to 1ms), 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 Table 4.2, 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 4.2: 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, 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, which assume *a priori* full knowledge of the system and non-loss continuously variable voltage.

These results show clearly the potential advantage of our method. In section 4.2 we will present a run-time scheduling heuristic on how to achieve it in practice.

4.2 Run-time Scheduling Algorithm

Section 4.1 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 a formal mathematical model. Then a greedy heuristic is proposed for our specific problem.

4.2.1 Application Model

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

- At the beginning of every time frame, there are k TFs waiting to be executed, each represented by a Pareo curve.
- Each thread frame i has N_{ij} Pareto points, i.e., N_{ij} different ways of mapping and ordering on the given platform and they are represented with their execution time t_{ij} and energy consumption e_{ij}.
- At any moment, only one thread frame can be executed on the given platform. In other words, that thread frame occupies the platform exclusively[2].
- There is a global deadline D before which all the thread frames have to finish.

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

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

4.2.2 Problem Formulation

For the application given above, we can formulate our run-time scheduling as follows. Since k thread frames exist and each of them has N_i Pareto points, we can introduce an integer variable x_{ij} to denote whether the jth Pareto point

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

[2] This constraint will be removed in Chapter 5

of TF i is selected (x_{ij} equals 1) or not (x_{ij} equals 0). For each thread frame, one and just one Pareto point can be selected, which leads to:

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

When a Pareto point i is selected for TF j, it means execution time t_{ij}. The total system execution time can never exceed the global deadline D for real-time systems. Therefore we have:

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

The goal of our run-time scheduler is to reduce the total system energy consumption as much as possible. This can be represented as:

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

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

$$\begin{aligned} minimize: \quad & z = \textstyle\sum_{i=1}^{k} \sum_{j=1}^{N_i} e_{ij} x_{ij} && (4.1) \\ subject\ to \quad & \textstyle\sum_{i=1}^{k} \sum_{j=1}^{N_i} t_{ij} x_{ij} \leq D, && (4.2) \\ & \textstyle\sum_{j=1}^{N_i} x_{ij} = 1, i = 1, \ldots, k, && (4.3) \\ & x_{ij}\ is\ 0\ or\ 1, i = 1, \ldots, k,\ j = 1, \ldots, N_i. && (4.4) \end{aligned}$$

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

The minimization problem can be transformed into a different form [110]. Taking into account that each Pareto curve is an ordered set, we can substitute e_{ij} with s_{ij} as

$$s_{ij} = (e_{i0} - e_{ij}), s_{ij} \geq 0. \quad (4.5)$$

Thus Eqn. 4.1 becomes a maximization problem:

$$maximize : z^{'} = \sum_{i=1}^{k} \sum_{j=1}^{N_i} s_{ij} x_{ij} \quad (4.6)$$

With the same set of constraints, this is a classic Multiple Choice Knapsack Problem (MCKP) and it is known as NP hard [107].

When of limited size, MCKP can be solved optimally in pseudo-polynomial time through dynamic programming. For bigger instances, it is generally solved by a dynamic programming (DP) algorithm constructed from the exact solution of its linear relaxation, LMCKP, by replacing Eq. 4.4 with

$$0 \leq x_{ij} \leq 1, i = 1, \ldots, k, j = 1, \ldots, N_i. \quad (4.7)$$

Several exact algorithms have been proposed to solve the reduced LMCKP problem in $O(n)$ time [107]. To evaluate the result of our algorithm, we use the DP algorithm presented in [119]. However, the worst case computation complexity of DP is still exponential, which is not acceptable as an on-line algorithm for medium problem size. Another issue is that the computation time of DP is nondeterministic, which is undesirable for real-time systems.

Several approximate algorithms exist for MCKP but all have limitations or are not suitable for our problem. Current heuristics are designed for big problems, which can not be solved easily by any accurate algorithm due to the problem's NP-hard feature. They rival each other in which can get a solution closer to the optimal value or which can handle a bigger (or more difficult) problem. Execution time is only the second or the third concern to them, which makes them unsuitable to work as an 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. This can guarantee a deterministic computation time of the heuristic.

4.2.3 Greedy Heuristic

We have developed a fast and effective greedy heuristic with the above considerations in mind. Algorithm 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 $i+1$, see Figure 4.3) or to the left (from point i 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. For finding the initial solution we use an on-the-fly strategy. The difference between the time assigned to curve m and the actual execution time of its initial solution will be accumulated in the variable *slack* and added to the available

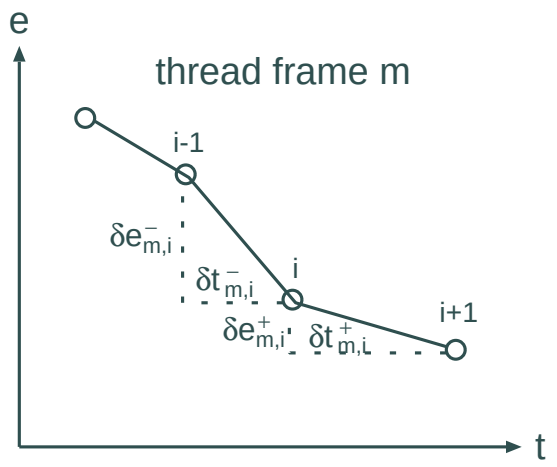


Figure 4.3: The Pareto curve of thread frame m.

time of the following curves.

After the initialization, we explore the chances of finer tuning the solution in two steps, *step*1 and *step*2. *step*1 checks the possibility of moving the operating point on one curve to the right and the operating point on another curve to the left in pair. At line 19, all curves are sorted according to the slopes of their current solutions, $slope^+$ descendingly and $slope^-$ ascendingly. Then the algorithm will try to find two curves m and n, which satisfy the time constraint and reduce the energy consumption most, when the solution of m is changed from i to $i+1$ and the solution of n from j to $j-1$ (Figure 4.4). When no such kind of tuning is possible, the algorithm will enter the next step.

*step*2 does the final tuning by finding any curve m which can still satisfy the time constraint if we move its current solution from i to $i+1$. It is possible to switch the order of these two steps. However, our experiments show the current order is faster and generally leads to better solutions. Another option is to move the operating point to the right as much as possible in *step*2. In that case, if *step*2 is done before *step*1, this will cause the heuristic to converge in fewer iterations but deteriorate the optimality of the final solution.

Assuming k curves and l points are present on each curve, the complexity of the initialization step is $O(k \log l)$ because for every curve we have only to do an ordered search (line 14). The complexity of the iterative stage is also very low. In *step*1 every iteration takes maximally $O(k^2)$ operations, while in *step*2 $O(k)$ operations. The heuristic ends when no improvement is possible, but we can interrupt the iteration at any moment to finish the run-time scheduling

Algorithm 1 The greedy heuristic algorithm.

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

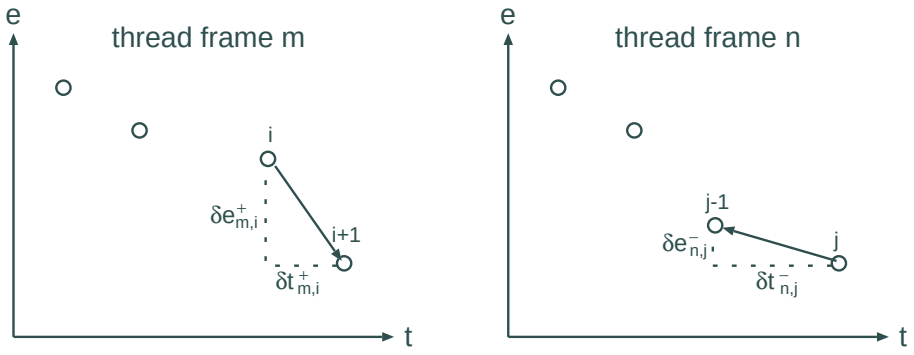


Figure 4.4: Incremental improvement step 1, when the operating points change from (i, j) to $(i + 1, j - 1)$. $\delta t^{+}_{m,i} < \delta t^{-}_{n,j} + slack$ and $\delta e^{+}_{m,i} > \delta e^{-}_{n,j}$ have to be satisfied to get a valid and meaningful solution.

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 4.3.

4.3 Experimental Results

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

4.3.1 Randomly Generated Test Cases

The first test set we have used is the task graphs generated by TGFF. For each task graph, a Genetic Algorithm [183] is used to extract the Pareto curve, on an architecture like the one we used in section 4.1. Finally the heuristic is applied to find the on-line task scheduling within a given deadline. A dynamic programming (DP) optimal algorithm [119] 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 the same Pareto curves and the results are summarized in Table 4.3 and Table 4.4.

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 4.3: 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 4.4: 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. Table 4.3 and Table 4.4 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 the execution time of the greedy heuristic against the DP solver; the third column is the maximum speedup; the fourth column gives the average error between the heuristic and DP solution and the fifth column is the maximum error. The next four columns are just the same but for the initial solution given by *step*0 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 executing the application functionality. So it has to be minimized or bounded, even though we can have a separate CPU to run the scheduler in some architectures. Our heuristic provides the capability of improving the initial solution iteratively until the time slot assigned to the scheduler depletes. This is especially important for big problem sizes, when the scheduler could not run to its end and still has to find a solution in a short time slot. Table 4.5 shows an example of the iterative improvement of our heuristic. This example

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

Table 4.5: The iterative improvement of the heuristic for a 20 curves, 9 points case.

is for the 9 points per curve, 20 curves case because it is the worst case in our experiment with respect to the execution time. The optimal result is 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.

4.3.2 Real-Life Applications

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

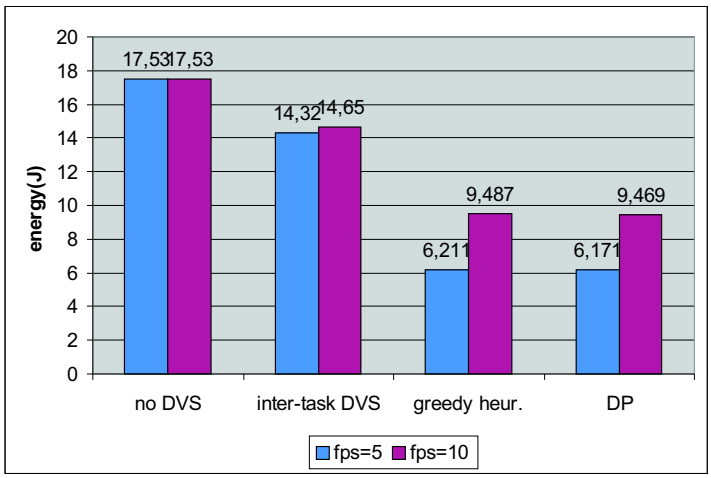


Figure 4.5: The energy consumption of QoS adjustment algorithm for 1000 frames.

application which will be explained in section 6.1.1 in detail. Every time frame, depending on the number of visible objects and which kind of objects they are, the QoS controller will adjust the number of vertices assigned to each object, in order to provide the best quality at a fixed computation power. Figure 4.5 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, during most of the frames, the heuristic can easily find the optimal solution due to the limited problem size.

Another real-life application we have experimented on is the Visual Texture Coding (VTC) decoder of the MPEG-4 standard. Similar to the QoS example, it is frame based. However, unlike the 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 (see [101] for further discussion). As shown in Figure 4.6, this example gives less space for voltage scaling because of its relative high and less varying work load. This

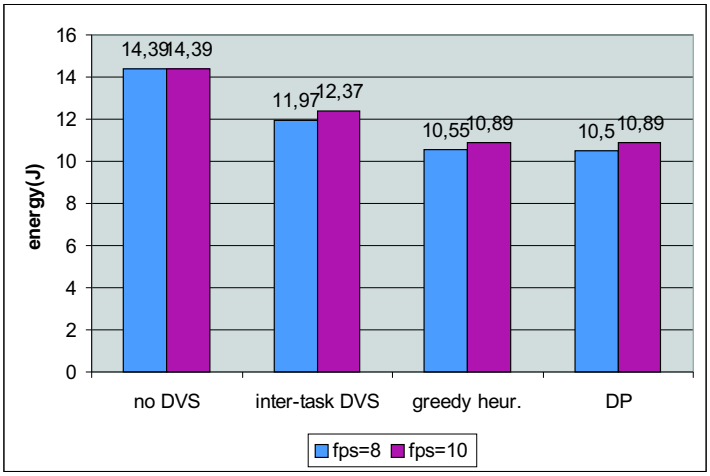


Figure 4.6: The energy consumption of the VTC decoder for 1365 frames.

is mainly due to the sequential feature of the initial task graph[3]. 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.

4.4 Conclusion

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

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

Chapter 5

Run-Time Algorithm for Overlapping Task Schedules

With the semiconductor processing technology entering into the deep submicron era, it is now possible to put multiple processors on a single chip and it has been recognized that the heterogeneous multiprocessor is the most performance and power efficient platform. How to schedule tasks at run time to best benefit from such a platform is a novel problem in the scheduling research community where homogeneous cases are the norm now. In this chapter, we first illustrate the advantage that heterogeneous platforms can bring. Then we model the run time task scheduling as a multi-mode project scheduling problem and a Tabu search based heuristic is proposed. Compared to the algorithm in the previous chapter, the new heuristic allows several applications to share the same multiprocessor platform simultaneously and thus normally results in a better scheduling result.

5.1 Motivational Example

5.1.1 The Heterogeneous Platform

A heterogeneous multiprocessor platform comprises several types of processor, on each of which the same TN can be executed at different speed with different energy consumption. Due to its characteristics, a TN can be more efficiently executed on one type of processor than the others. For instance, if it contains a lot of instruction level parallelism, it makes sense to run it on a VLIW processor because more than one operations can be issued per cycle. However, if it has

a lower level of parallelism (like a section of control dominated code), it will not run much faster on a VLIW compared to a simple processor core such as ARM, but it may consume much more energy. Conventionally this kind of partitioning and mapping is performed at design time. In this chapter we will show an approach to partly postpone that step so that the part of the decisions, which is very dependent on the dynamic behavior, can be explored at run time. Then much more relevant information is available about the dynamic system context, so the quality of the decisions can be highly improved. Compared to the inspector-executor method [43], our approach is still more effective because of the thorough design time exploration in each task that still precedes the run time phase.

The heterogeneous multiprocessor platform we use in this chapter as a representative example comprises two ARM 940T cores and two TI 6701 DSPs. According to TI's datasheet, the power consumptions of 6701 running at 167MHz are 1.33W and 0.57W for the high activity and the low activity cases respectively. Here high activity means all its 8 function units are in use, typically for highly optimized data dominated TNs. When it is running at low activity, only 2 of its 8 function units are occupied. The ARM core has a much lower power consumption, which is only 0.8mW/MHz, but also a much lower parallelism potential. This gives a power consumption of 134mW at 167MHz. All the above power numbers include the processor core and the attached local memory (used as cache or fast internal memory) but exclude the power consumption on the bus, the external memory or other peripherals. Taking into account that the TI DSP keeps two function units busy even at low activity, we can assume it is 1.5 times faster than ARM in that situation. The last assumption we make is about floating point operation. The TI 6701 is able to do floating point computation, which is not supported by ARM but can be emulated in software. Therefore, when TNs containing floating point computation are considered, we assume TI has an extra 2-fold speedup.

Putting all the above discussion together, we have the execution time and energy consumption in different situations in Table 5.1, where H (L) represents high (low) activity TNs and I (F) stands for integral (floating point) code sections. An interesting observation is, though in all the other cases the TI DSP is faster but more power hungry, it consumes even less energy than the ARM when highly optimized floating point computation is considered.

5.1.2 Design Space Exploration

Any TF can be separated into smaller TNs, implementing individual functions such as image compressing or user interface handling, and the TF can be represented as a task graph. Every node on that graph is a TN and can be classified

flavor	processor type	execution time(ms)	energy (mJ)
L/I	6701	8	4.56
	ARM	12	1.61
L/F	6701	4	2.28
	ARM	12	1.61
H/I	6701	2	2.66
	ARM	12	1.61
H/F	6701	1	1.33
	ARM	12	1.61

Table 5.1: The execution time and energy consumption in different situations.

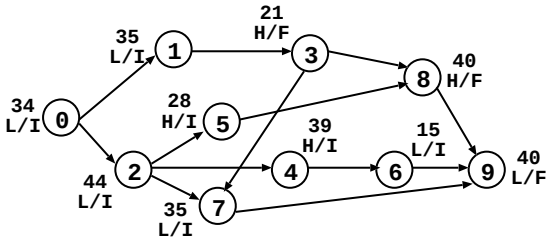


Figure 5.1: An example of the task graph.

as one of the four flavors we discussed previously. The directed arc indicates the precedence relations among the nodes. Figure 5.1 gives an example of task graphs. Besides the flavor of the node, the execution time on the ARM processor is also shown. Given such a task graph and a set of available processors, how to map and order the nodes is a classic scheduling problem. Different decisions can result in the tradeoff among the speed, the energy consumption and the number of resources used, which is known as design space exploration. For our example, Table 5.2 lists several possible solutions.

5.1.3 Run-Time Scheduling

Traditionally, design space exploration is fully done at design time. Whenever a solution is selected, it is fixed and can never be changed. For example, if a TF has been mapped to an implementation with 1 ARM and 1 TI, it is impossible to change to an implementation with 2 ARM and 2 TI at run time, though this change may bring some extra benefit. Our approach is different in that we also explore that design space at design time but we postpone the final decision-making stage till run time to better tune the system to the changing

mapping choice	# ARM	# TI	execution time(ms)	energy (mJ)
1	1	0	331	44.35
2	2	0	204	44.35
3	0	1	146	94.95
4	0	2	93	94.95
5	1	1	106	77.73
6			180	56.25
7			270	42.94
8	2	2	181	42.94

Table 5.2: The different mapping choices of the task graph on Figure 5.1.

requirements of the dynamic system context. As a very simple example, suppose we have a time budget of 181ms to run the TF on Figure 5.1. To minimize the energy consumption, it is best to select the mapping choice 8 (2 ARM, 2 TI) from Table 5.2. At some unpredictable time, another TF instance pops up and we are required to complete both of them in the same time budget. Then we have to change to the mapping choice 6 and run them simultaneously (each TF uses 1 ARM and 1 TI) to be most energy efficient within the real-time constraint.

To maximize the flexibility, we use a hierarchical model to represent the system. A system comprises several TFs and they are represented by a directed arc graph (see Figure 5.2). Every TF has its internal structure and can be further divided into TNs. A design space exploration step can be applied independently to every type of TF to find different mapping choices, as we have discussed. Due to the dynamic feature of the system, only at run time we are able to know how many TFs present, what their types are and what their precedence relations are. A deadline is also given by which all TFs have to finish. The goal of the run-time scheduler is to select a mapping choice for every TF and order them in time so that it can satisfy all the constraints and minimize the energy. This is a combined scheduling and optimization problem.

We can certainly apply the heuristic presented in Chapter 4, which will be referred as the sequential heuristic in this thesis, for the run time scheduling problem. However, that heuristic assumes a sequential execution of all the TFs. Whenever a TF is executing, it occupies the complete multiprocessor platform exclusively and no other TFs can be executed at the same time. This simplification is fine when the platform comprises only a few (typically 2 or 3) processors. It will not cause a serious problem because the parallelism inside a TF will keep the processors busy most of the time. However, when more processors are added into the system, the chance that a processor is idle

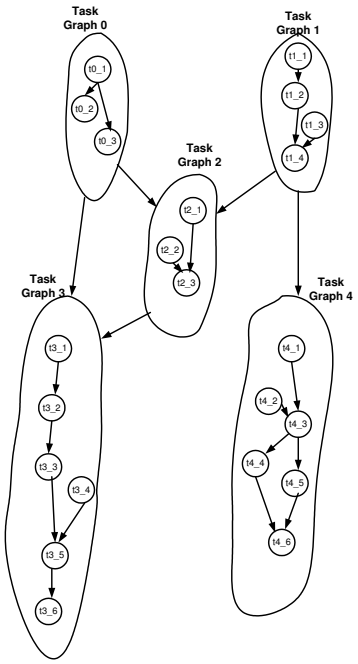


Figure 5.2: A system with 5 task graphs.

is increasing. To solve that problem, we propose a new run-time scheduling heuristic, which allows parallel execution of several TFs. As depicted in Figure 5.3, the execution time of TF0 and TF1 are overlapped partly, and that is why it is called the "overlapping" heuristic. The internal sub-tasks (thread nodes) are not interleaved though, partly to limit the exploration space at run time. There is ongoing research to reconsider that restriction [101].

For the system shown in Figure 5.2 and deadline 1200ms, our run-time scheduler will find a scheduling with energy consumption 541mJ (denoted as overlapping in Figure 5.3), while our reference case (sequential heuristic) consumes 701mJ energy. This clearly motivates our research focus.

5.2 Run-time Scheduling Heuristic

A problem related to our overlapping run-time scheduling is the multi-mode resource constrained project scheduling problem (MRCPSP), which is notoriously hard to solve [74, 73]. However, MRCPSP tries to minimize the makespan

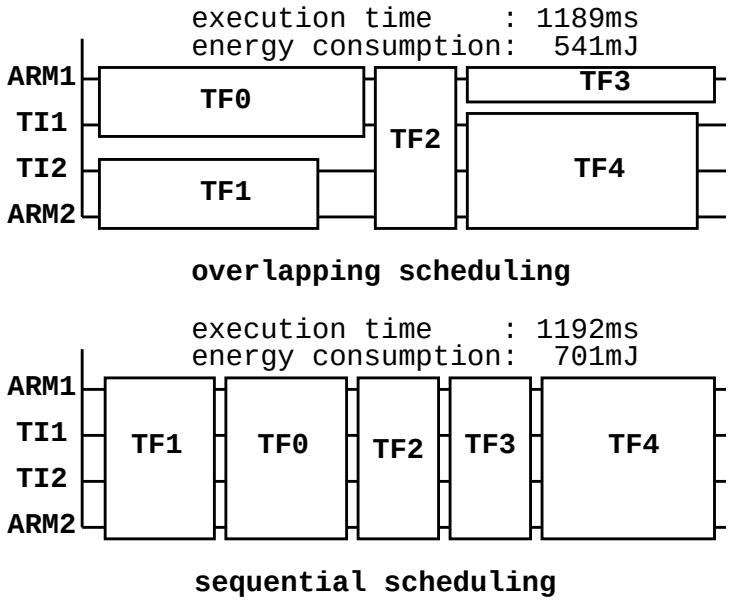


Figure 5.3: Overlapping and sequential scheduling give different results for the same task set and the platform.

(i.e., the completion time of the system)[1], while we have a given deadline for the system completion time and another objective to optimize. This makes our problem even harder to solve since we have to satisfy both the resource and the time constraints. For MRCPSP, several heuristics are proposed, but most of them are based on genetic algorithm, simulated annealing or branch and bound and are therefore not applicable as a run-time heuristic [54, 73].

Our problem is unique in several ways. Firstly, we consider a heterogeneous platform. Secondly, every TF can be executed in different mapping choices, each of which occupies different number/type of processors for a different period of time, consuming certain amount of energy. Thirdly, precedence relations exist among the TFs. Forthly, the run-time scheduler tries to minimize the total system energy consumption while satisfying all the constraints. To our best knowledge, this is the first work trying to solve that combined scheduling and optimization problem.

We consider a system which consists of J TFs labeled $j = 1, ..., J$. Precedence relations exist between some of the TFs. These precedence relations are given by sets of immediate predecessors P_j indicating that an TF j may not be started before all of its predecessors are completed. Each TF can be performed in one of several different flavors of accomplishment (or mapping choices), each of which requires a combination of different type and number of resources (ARM and TI processors in our case). TF j may be executed in M_j different mapping choices. Every mapping choice has the resource requirement, an execution time t_{jm} and energy consumption e_{jm}. We have to select a mapping choice for every TF, schedule the TFs with the given precedence relations and resources, and meet an explicit deadline. The objective is to minimize the total energy consumption.

In this chapter, we propose a Tabu Search based heuristic. Tabu Search is essentially a local search algorithm [73]. That is, it evaluates all solutions of the neighborhood and choose the one most potentially improving the solution. This concept, however, bears the possibility of cycling, that is, one may always move back to the same local optimum one has just left. In order to avoid this problem, a tabu list is set up as a form of memory for the search process. Usually, the tabu list is used to forbid those neighborhood moves that might cancel the effect of recently performed moves and might thus lead back to a recently visited solution. The scheme of our heuristic is illustrated in Algorithm 2. At first a ParallelSGS is called (explained later) to generate the initial solution. Then the local search is started. If it succeeds in finding a better solution, the current scheduling is updated. Otherwise the unsuccessful try is put into the tabu list to prevent it from being tried again in the near future. At the end of

[1]Other objectives are considered by a few researchers, but in no case known to the authors an explicit deadline is taken into account.

every iteration, the tabu list is updated to unfreeze the outdated tabu items. That is, these moves are again possible choices. The constants N and K limit the number of evaluations.

Algorithm 2 The Tabu Search heuristic.

```
1: INITIALIZATION
2: ParallelSGS;
3: TABU SEARCH
4: num_try=0;
5: A: num_fail=0;
6: while num_try < N and num_fail < K do
7:    select one potential move m which changes the TF j from mapping choice
      j_s to j_t and is not in the tabu list;
8:    if ParallelSGS(m) succeed then
9:       update the solution;
10:      goto A:;
11:   else
12:      put move m into the tabu list;
13:      num_fail++;
14:   end if
15:   num_try++;
16:   update the tabu list;
17: end while
```

In the Tabu heuristic, we apply the standard parallel scheduling scheme (ParallelSGS) to generate a scheduling, which is presented in Algorithm 3. For each iteration g a schedule time t_g is chosen. TFs which have been scheduled up to g are either element of the complete set C_g or of the active set A_g. The eligible set D_g comprises all TFs which can be precedence-and resource-feasibly started at t_g. Each TF has a finish time F_j. The algorithm generates a scheduling in at most J steps, where J is the number of TFs.

On line 8 and 9 of Algorithm 3, two decisions have to be made: how to select a TF from the eligible set and how to select a mapping for the chosen TF. For the ParallelSGS in the initialization stage of Algorithm 2 (line 2), we apply two heuristics. We select an TF with the maximum number of direct succeeding nodes and choose the mapping with the shortest execution time. According to the numerical experiment in [73], this tends to give a good initial solution. Whereas for the ParallelSGS in the Tabu Search stage of Algorithm 2 (line 8), the mapping choice has already been fixed. For the TF, we select the one which consumes most resources, hoping the completion of it will release more resources for other TFs. This is similar to the best fit heuristic. After the scheduling, every TF j is given a mapping choice number j_m and a finish time

Algorithm 3 The ParallelSGS algorithm.

```
1: INITIALIZATION
2: g=0,t_g=0,A_0=0,C_0=0;
3: while | A_g ∪ C_g |≤ n do
4:    g = g + 1;
5:    t_g = min_{j∈A_g}{F_j};
6:    calculate C_g, A_g, D_g and remaining resources;
7:    while D_g ≠ 0 do
8:       select one j ∈ D_g;
9:       select mapping choice j_m for j;
10:      F_j = t_g + p_{jm};
11:      calculate A_g, D_g and remaining resources;
12:   end while
13: end while
```

F_j. For any two TF i and j, if they satisfy $F_i < F_j - t_{jm}$ and $F_j < F_i - t_{im}$, no precedence relation is present between them and enough resources (processors in our case) exist to let them run in parallel for an overlapped period.

5.3 Experimental Results

For our experiments, we need tens of task graphs to perform a reasonable evaluation. Due to the lack of appropriate task graphs from real life applications, we have decided to use TGFF (Task Graph For Fee, a tool from Princeton [42]) to generate them randomly. This is also used by many other researchers in the scheduling community [186]. We have generated three sets of task graphs, comprising 5, 5 and 10 task graphs respectively. These task graphs have node numbers varying from 10 to 21. Every node is assigned an execution time (on ARM) and a flavor number (L/I, L/F, H/I or H/F, as explained in Section 5.1.1) randomly. For every task graph, a Genetic Algorithm is used to generate a set of schedulings for every possible mapping choice with different number of ARM and TI processors. The results from this design space exploration step are similar to Table 5.2 and are pre-stored for every task graph. This is done at design time. Only at run time, the system is able to be aware of how many task graphs present, what they are and what the precedence relations among them look like. Then, given a deadline, our Tabu Search based heuristic selects a mapping choice for every task graph in the current system, schedules them according to their precedence relations and makes sure they satisfy both the resource and the time constraints. At the same time, it tries to reduce the system energy as much as possible.

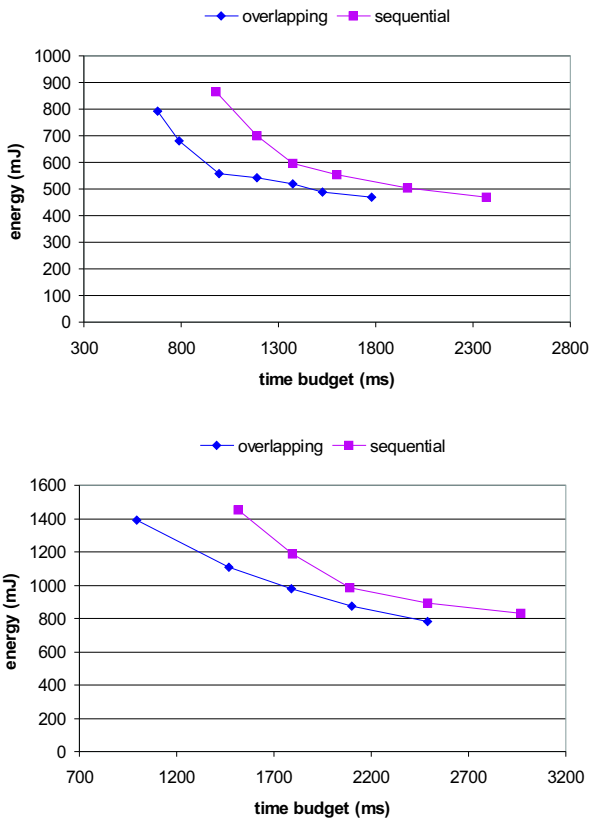


Figure 5.4: Result for set of 5 (top) and 10 TFs (bottom).

Because of the complexity of our problem, it is difficult to find the optimal solution to compare with. Moreover, as we discuss above, no earlier algorithms address the same problem with hybrid design and run time decision for optimization under real-time constraints. Consequently, we only compare our result with the heuristic presented in Chapter 4, which also makes tradeoffs at run time but it only considers a sequential task scheduling on the architecture with 2 ARM and 2 TI. The difference between them can be found on Figure 5.3. That heuristic has been proven efficient and effective in reducing the system energy and other cost. Since our Tabu Search based heuristic allows different TFs to overlap in the time axis and share the platform simultaneously, we refer it as the "overlapping" heuristic later in this thesis. The other one is called the "sequential" heuristic (see Section 5.1.3 for a more detailed discussion).

We have first generated two sets of TFs, one with 5 TFs and the other with 10 TFs. After applying the Genetic Algorithm for design space exploration, we execute our overlapping heuristic for different deadlines and compare it to the result from the sequential heuristic (see Figure 5.4). For these two sets of task graphs, the overlapping heuristic performs quite well. The first observation from the result is that the overlapping heuristic allows the system to meet a tighter deadline even when the same set of TFs is executed. In particular 33% reduction is observed for the 10 TF case. Moreover, if the same deadlines are assumed, the overlapping heuristic always defeats the sequential heuristic in terms of energy consumption, giving energy savings up to 24% and 34% respectively (when the leftmost point of the sequential scheduling is considered). With the deadline becoming less severe, the energy saving compared to the sequential scheduling drops because the potential energy saving space (the difference to the most time relaxed case) decreases. When the deadline is so loose that every code section is able to run at its most energy efficient way, no energy saving space is left for either heuristic.

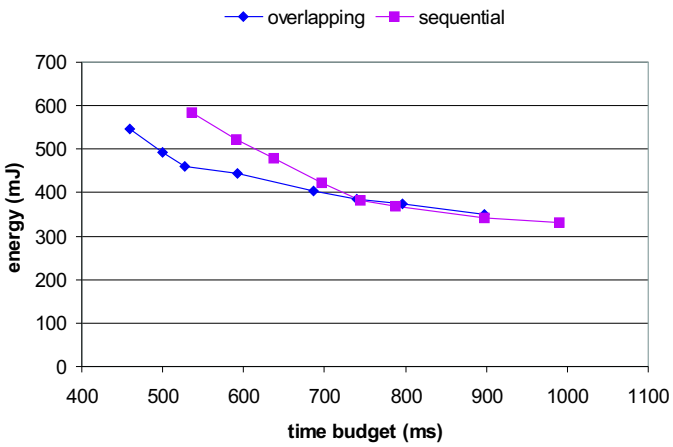


Figure 5.5: Result for another set of 5 TFs.

We have done the same experiment on another set of 5 TFs and the result is shown on Figure 5.5. Here we can also observe an energy saving up to 21% when the deadline is tight. However, as the deadline is loosened to some point, the overlapping solution is even a little worse than the sequential case. This is because the overlapping problem is intrinsically much harder to solve than the sequential problem. For some special cases, the result given by our current Tabu Search heuristic is a little farther from the optimal solution while the sequential heuristic still manages to give a good result. Also, the current

overlapping heuristic is tens of times slower than the sequential heuristic, due to the hardness of the problem. Hence, it is suggested that the overlapping heuristic should be mainly used for tight deadline constraints, where it can gain higher energy savings, while the sequential heuristic is used for loose deadline constraints, where it exhibits low overhead and good quality.

5.4 Conclusion

In this chapter we have illustrated that a heterogeneous multiprocessor platform can provide a wide design exploration space. Instead of fixing one point in that space at design time, we propose to postpone that decision making partly till run time to better tune the system to its dynamic context. In combination with further exploring the parallelism of multiprocessor platforms, we have modeled the problem as a multi-mode project scheduling problem and have developed a heuristic using Tabu search. Results from randomly generated task sets show energy savings up to 34% compared to the sequential heuristic presented in the previous chapter.

Chapter 6

Validating the Methodology with Demonstrators

We have validated our complete TCM design flow with two real-life demonstrators. The 3D QoS adjustment application is integrated with our run-time scheduler on top of the Virtuoso RTOS and then is simulated on a Windows PC. The PocketGL is implemented on a real XScale board with Linux and all the energy numbers are measured and recorded at real time. Moreover, the software is able to change the processor supplying voltage and frequency by APIs provided by the enhanced operating system. Both applications consider a uniprocessor architecture with discretely changeable supplying voltage. A multiprocessor experiment can be found in Chapter 7 .

6.1 3D rendering QoS Control Demonstrator

6.1.1 The QoS Application

To test the effectiveness of our approach, a real-life application, the QoS (Quality of Service) control part of a 3D rendering algorithm developed in the MPEG21 context, is used.

Figure 6.1 shows how 3D decoding/rendering is typically performed: a 2D texture and a 3D mesh are first decoded and then mapped together to give the illusion of a scene with 3D objects. This kind of 3D rendering requires that

each frame of the rendering process is recalculated completely. The required

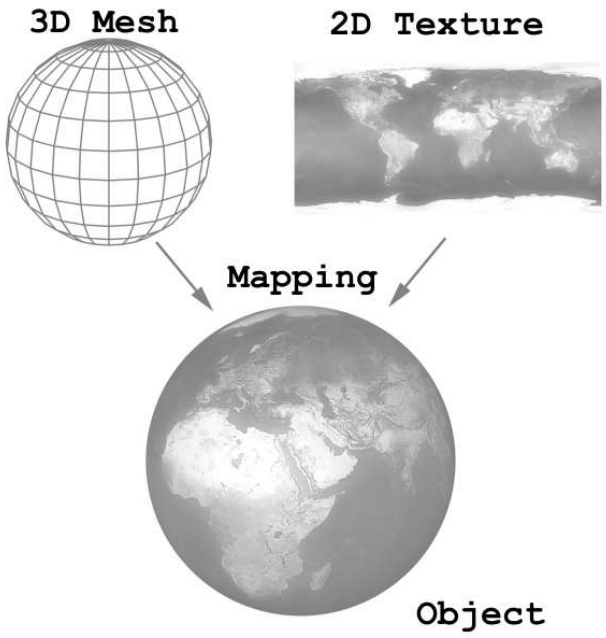


Figure 6.1: 3D rendering consists of 2D texture and 3D mesh decoding.

computation power depends significantly on its number of triangles. When the available resources are not enough to render the object, instead of letting the system break down (totally stop the decoding and rendering during a period of time), the corresponding mesh of the object can be gracefully degraded to decrease resource requirement, while maintaining the maximal possible quality.

The number of triangles that are used to describe a mesh can be scaled up or down. This can be achieved by performing *edge collapses* and *vertex splits* respectively, as shown in Figure 6.2. To perform an edge collapse, and thus remove the edge (V_s, V_t), we first remove the triangles which V_s and V_t have in common, and replace V_t with V_s in the triangles adjacent to V_t. We then recenter V_s, to keep the appearance of the new set of triangles as close as possible to the former one. The new set of triangles represents the same object with less detail but also with less triangles. The same principle but in a reversed direction is used to perform a vertex split. The edge collapse and vertex split approaches can be used repeatedly till a desired number of triangles are achieved.

For a 3D object, the more triangles that are used to represent its mesh, the

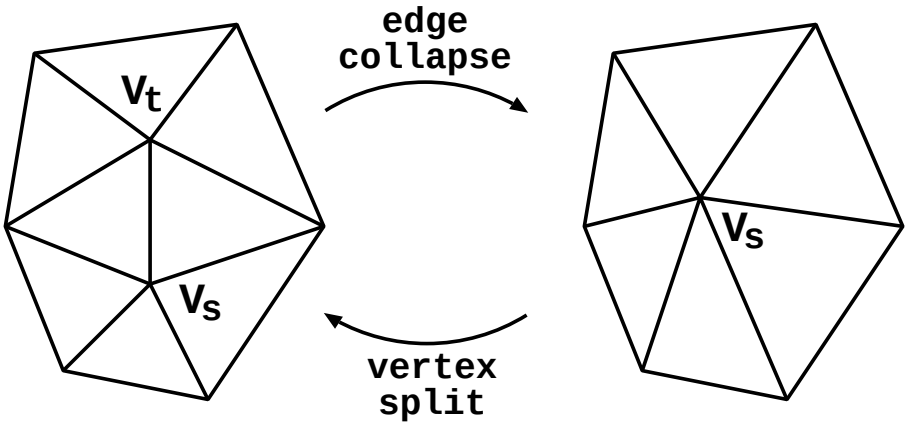


Figure 6.2: Edge collapse and vertex split.

more precise the description of the object. This increases the perceived quality. However, it slows down the geometry and rasterizing stages because more computation has to be done there. Consequently it decreases the number of frames that can be generated each second (FPS, frame per second), while most videos or 3D game applications desire a fixed FPS. Another issue that we have to consider here is that the same application can be run on different platforms, e.g., a desktop PC or a PDA, which provides completely different computation ability and power consumption features. Hence different qualities of the same service have to be supplied to achieve a similar FPS. For a given computation platform and a desired FPS, the number of triangles it can handle in one frame is almost fixed. Based on the number of objects in the current frame and what these objects are, the QoS controller will assign the triangles to each object so that the user can get the best-of-effort visual quality at a fixed frame rate.

6.1.2 Virtuoso RTOS

Virtuoso (now known as VSPWorks from Windriver) is a commercial RTOS, which features a high-performance kernel design with small memory footprint, and an advanced virtual single-processor (VSP) architecture for the development of embedded multiprocessor and distributed applications. The VSP combines the power of parallel processing with the simplicity of traditional multitasking programming. It implements a multilevel architecture (see Figure 6.3) to combine fast interrupt handling with scalable multi-tasking. At the heart of the system is a highly optimized nanokernel with very low context overhead. Below the nanokernel are the Interrupt Service Routines dedicated to

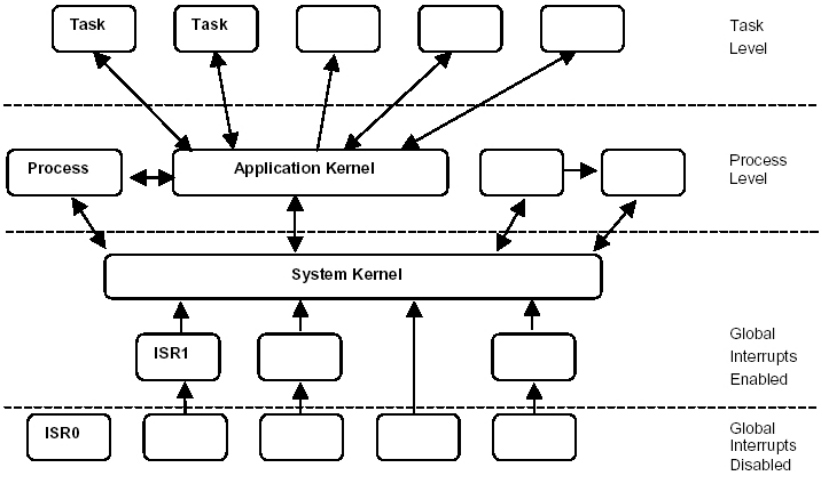


Figure 6.3: Virtuoso multilevel architecture.

high-speed interrupt handling, while the microkernel sits above the nanokernel and handles preemptive multitasking C/C++ tasks. Normally the nanokernel services take about 10 to 20 times less time than equivalent services at the microkernel level. This is not only due to the difference in number of registers that have to be swapped but mainly because of the much richer semantic context of the microkernel services. Each nanokernel process starts up and finishes as an assembly routine, and is scheduled as prioritized round-robin (preemption is not supported).

In this experiment, we have only used the microkernel level model and we therefore give some more information about it here. The microkernel level provides a virtual single processor model, which hides the detail of the low level HW and allows the programmer to use the normal multitasking programming model. It is based on the concept of microkernel objects, which are data structures with specific operations. Tasks, semaphores, mailboxes are all objects and are identified uniquely at the system-wide level. At a later node binding stage, the objects will be mapped to different processing nodes (processors) and be accessed transparently from the viewpoint of the application programmer. All the objects are allocated and bound at compile time and can not be moved around at run time. However, it is quite easy to try different bindings because of the available VSP model. Preemptive priority scheduling is provided at this level, but it is the responsibility of the user to guarantee the real-time feature, if it is required, and this step is always tricky. At compile time, the neces-

sary components of the RTOS will be compiled and linked together with the application, generating loadable image files.

6.1.3 Applying the TCM Methodology

For the 3D rendering QoS application introduced in section 6.1.1, we have applied our complete design flow, from modeling to scenario selection to scheduling, and then compared the result to a few reference cases, by using our simulation environment.

The complete flow is depicted in Figure 6.4. The 3D QoS code is first profiled by the ARM simulator, ARMulator (enhanced with an energy model similar to [154]), to identify thread nodes and to extract execution time and energy consumption data. These results are passed to the design-time scheduler to generate Pareto curves. On the other hand, the code is transformed into a multi-threaded version manually (work is ongoing to deliver this at least semi-automatically later on). The multi-threaded code is then executed on a PC. The run-time data obtained by simulating the code on the PC gives the run-time scheduler important information such as the number of running threads and the type of them. Set on top of Virtuoso, the run-time scheduler makes decisions with the run-time data and pre-computed Pareto curves and feeds back the decisions to the PC simulator. At the same time, the run-time part collects data such as the energy consumption and deadline misses.

An inconsistency can be noted in the above approach. On one hand, the code is profiled and scheduled at design time based on its execution on ARMulatore, which uses the ARM instruction set; on the other hand, the same code is executed on a PC simulator to generate the run-time data. Two reasons are present for that. Firstly, Virtuoso has not been ported to the ARM platform. It will take considerable effort to develop a new board support package and it is not interesting to us. Secondly, we need OpenGL library support to render the scenes on a monitor. That library is not available for ARM. Nevertheless, that inconsistency will not damage the effectiveness of the way we validate our design methodology. In fact, the PC simulator is used here to generate the run-time data (the TFs and their scenarios), which is used by our run-time scheduler to find the appropriate scheduling. When the above practical constraints are removed, as in the PocketGL demonstrator in section 6.2 and the integration experiments in Chapter 7, we can apply the same methodology and obtain similar results.

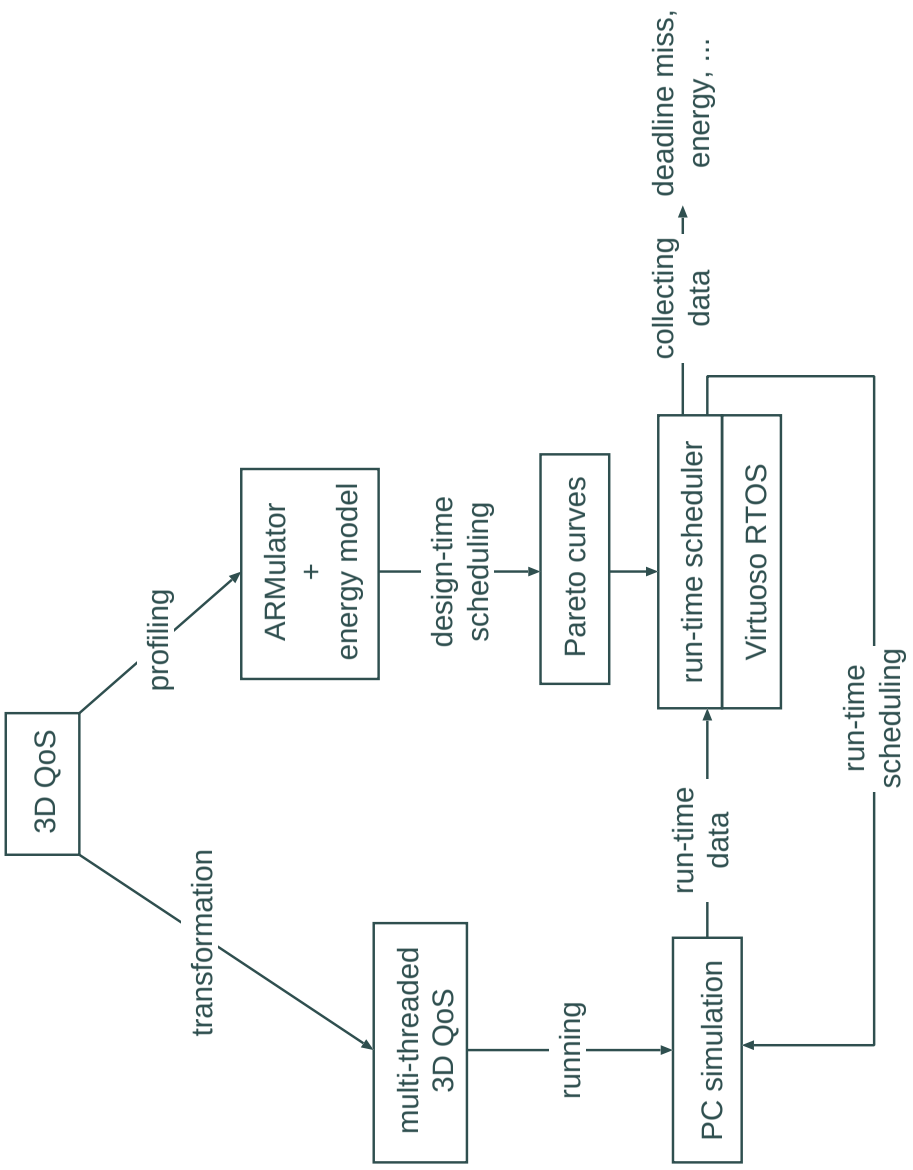


Figure 6.4: Apply the TCM design flow to the 3D QoS application.

Gray-box model

For every scene, we have to consider one *SceneUpdate* thread frame and several *AdjustObject* thread frames. The exact number of the latter depends on the output of the *SceneUpdate* thread frame of that scene and it varies from 2 to 12. This raises some problem to our run-time scheduler because it even does not know how many thread frames are in the current scene when it has to make scheduling. We solve that problem by considering the *AdjustObject* thread frames from scene i and the *SceneUpdate* thread frame from scene $i+1$ for one *scheduling frame*, as shown in Figure 6.5. In that case, the run-time scheduler

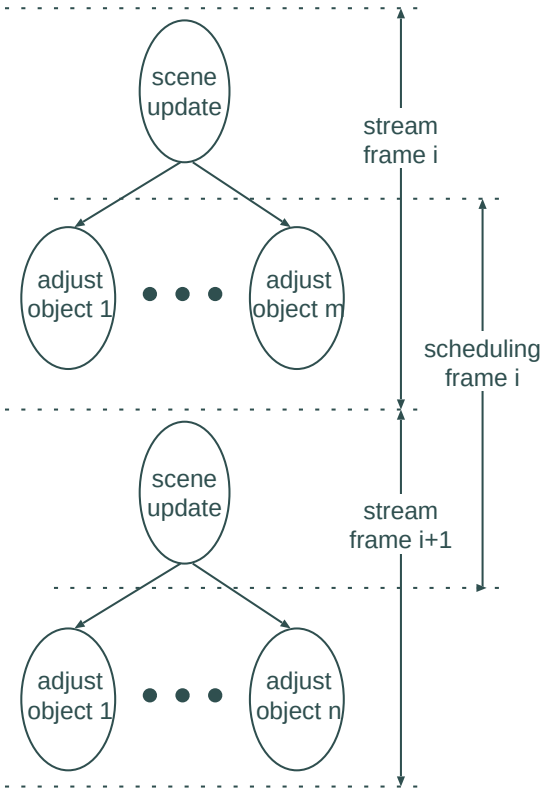


Figure 6.5: The run-time scheduler schedules the *AdjustObject* thread frames from scene i and the *SceneUpdate* thread frame from scene $i+1$ together.

has to make sure that the *SceneUpdate* thread frame can only start when all the *AdjustObject* thread frames of the previous scene have finished. This can be done by giving the *AdjustObject* thread frames priority levels higher than

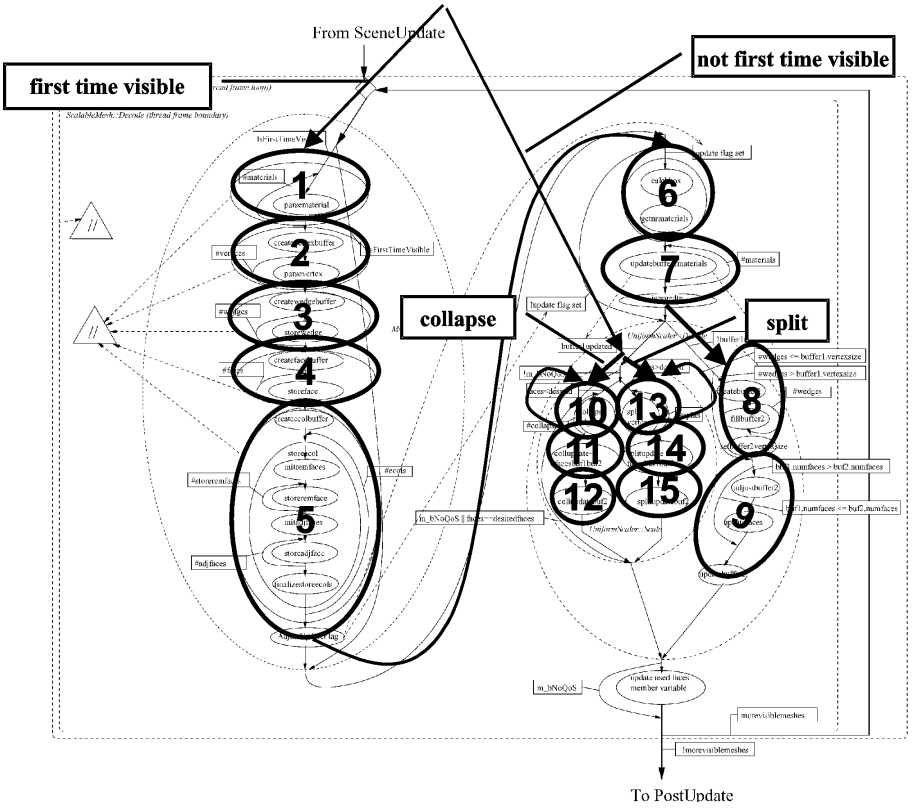


Figure 6.6: The gray-box model of the *AdjustObject* thread frame.

the *SceneUpdate* thread frame.

The *SceneUpdate* TF is quite static and only consumes a small part of the total execution time. Therefore, we look it as a thread frame consisting of only one thread node. The gray-box model of the *AdjustObject* thread frame is explained as below.

In the QoS kernel, for each visible object on the scene, a separate thread frame will be triggered, in which the number of triangles is adjusted to the number specified by the QoS algorithm. The gray-box model of that thread frame is shown in Figure 6.6, where all the internal TNs are numbered as well. Table 6.1 gives the profiled execution time and energy consumption of each TN on a 2.4V StrongARM processor.

Thread Node	Ex. Time (us)	En. Cons. (uJ)
1	35.6	52.2
2	1120.2	1475.3
3	2819.5	4173.4
4	4963.9	8698
5	8037.8	14112.6
6	1619.7	2840.1
7	3.2	4.3
8	761.1	1080.9
9	794.9	1180
10	63.7	113.1
11	432.2	638.1
12	0.2	0.1
13	68.1	120.3
14	705.5	1045.9
15	0.2	0.2

Table 6.1: Execution time and energy consumption of the TNs of the *AdjustObject* thread frame.

Scenario selection

From the gray-box model, we can see that based on whether each object is the first time visible, which is true only once during the whole stream for each object, one of the branches will be taken. If it is the first time visible, the mesh and texture have to be parsed, generated and bounded (TNs 1 to 9). If it is not, the current number of faces will be compared to the desired number of faces to decide whether to collapse edges (TNs 10, 11 and 12) or to split more vertices (TNs 13, 14 and 15). The edge collapse and the vertex split are done in a progressive and iterative way to avoid abrupt changes of the object shape with a while loop over TN 10 or 13. The iteration number of this loop depends on the difference between the current and desired number of faces of that object, and it varies from 2 to 1000 based on the profiling data.

Clearly, only one Pareto curve is not enough to represent these highly dynamic contexts. We have to distinguish first-time-visible or not-first-time-visible and the while loop iteration numbers. For the latter, if we fail to differentiate the iteration number over the loop body, we would have to consider the implementation for the worst case, which is 1000 iterations and much bigger than the average case. To avoid that, we have introduced the concept of "scenario selection" (see section 3.3) where different Pareto curves are assigned (in an

analysis step at design time) to run-time cases that motivate a set of different implementations. Based on this analysis, we have decided to use 9 different scenario's and hence also 9 Pareto curves in the QoS application to represent the run-time behavior of one object: the first one is when the object is first time visible; the others are when it is not and has to be collapsed or split. For "collapse" and "split", each are assigned four curves with different implementations, corresponding to different iteration sub-ranges. For example, the first curve of "collapse" will be selected if the actual iteration number falls between 2 and 12. Therefore, we only have to consider the worst case of that sub-range, which is 12 in this example, not the worst case of the whole range, which is 1000. Extra code has been inserted to enable this. We have selected these ranges based on the profiling data from the application and they are illustrated in Table 6.2.

	first time visible	collapse or split	range of iteration
scenario 0	yes		
scenario 1	no	collapse	2-12
scenario 2	no	collapse	13-30
scenario 3	no	collapse	31-180
scenario 4	no	collapse	181-1000
scenario 5	no	split	2-4
scenario 6	no	split	5-12
scenario 7	no	split	13-60
scenario 8	no	split	61-1000

Table 6.2: Scenario selection.

6.1.4 Implementation

Our run-time scheduler is implemented on top of the already discussed Virtuoso scheduler, which is priority-based and preemptive. Each task has an entry point and a priority level (as in almost all RTOSes, a lower value means a higher priority) assigned to it. At any time, the task with the highest priority is executed, unless

1. it is waiting for a semaphore or other system resources;
2. it has been stopped by another task;
3. it has reached its end point.

The priority can be changed at run time. The entry point is the point where a task will start execution whenever it is started, automatically (if it belongs to the EXE task type) or explicitly (Virtuoso kernel calls function *start*()), and it can also be changed at run-time. Since Virtuoso does not support run-time task creation and deletion, we have to allocate enough tasks when we initialize the application, then connect it to the code we want to run (setting the entry point) at run time and start it.

The detail of the implementation is shown in Figure 6.7, where all the solid horizontal arrows represent a context switch and all the vertical arrows or blocks represent a period of time. In the figure, we also give the name of tasks, where you can find them in the files and the priority assigned to each of them.

We have used four task entry points in this experiment. *TCM_QoS* is the whole application code excluding the scene update and QoS adjustment sections. Actually it works like a shell or wrapper in this implementation, and it sends out semaphores to synchronize the other tasks. *rt_schedule* is our run-time scheduler code. It checks the number and type of tasks it has to run, reads in the corresponding Pareto curves, selects the operation point for each task, then starts them. Both *TCM_QoS* and *rt_schedule* are Virtuoso EXE tasks, which means they will be started automatically when the application starts. *do_scene* is the code of *SceneUpdate* TF; *do_qos* is the code of *AdjustObject* TF. For each stream frame, there are one *do_scene* task, and a number of *do_qos* tasks. The exact number depends on how many objects are visible in that stream frame.

At the start point of the application, both *TCM_QoS* and *rt_scheduler* are runnable. The latter will execute first because it has a higher priority. After the scheduler initialization, it then waits for semaphore SCHEDULE and passes the control to *TCM_QoS*. *TCM_QoS* initializes the application first and calls *do_scene* directly for stream frame 0. This is different from the other frames and we will explain it later. After that, *TCM_QoS* triggers semaphore SCHEDULE and *rt_scheduler* is activated and get the control. *rt_scheduler* actually works on the *AdjustObject* TFs of stream frame i and the *SceneUpdate* TF of stream frame i+1, because we can not have the necessary information, such as how many *AdjustObject* TFs exist in frame i and the computation requirement of each TF, to direct the scheduling of the QoS TFs till we finish the *SceneUpdate* TF of stream frame i. Having selected the operation point for each TF based on the best known run-time knowledge of the stream frame, *rt_scheduler* starts *do_qos* and *do_scene* and waits for semaphore SCHEDULE again. At that moment, *do_qos* get the control because they have higher priorities, doing the real QoS adjustment for each visible object. When all the *do_qos* finish, *do_scene* starts, but it has to wait for semaphore SCN_UPDATE first to make sure that the current stream frame has finished and the next

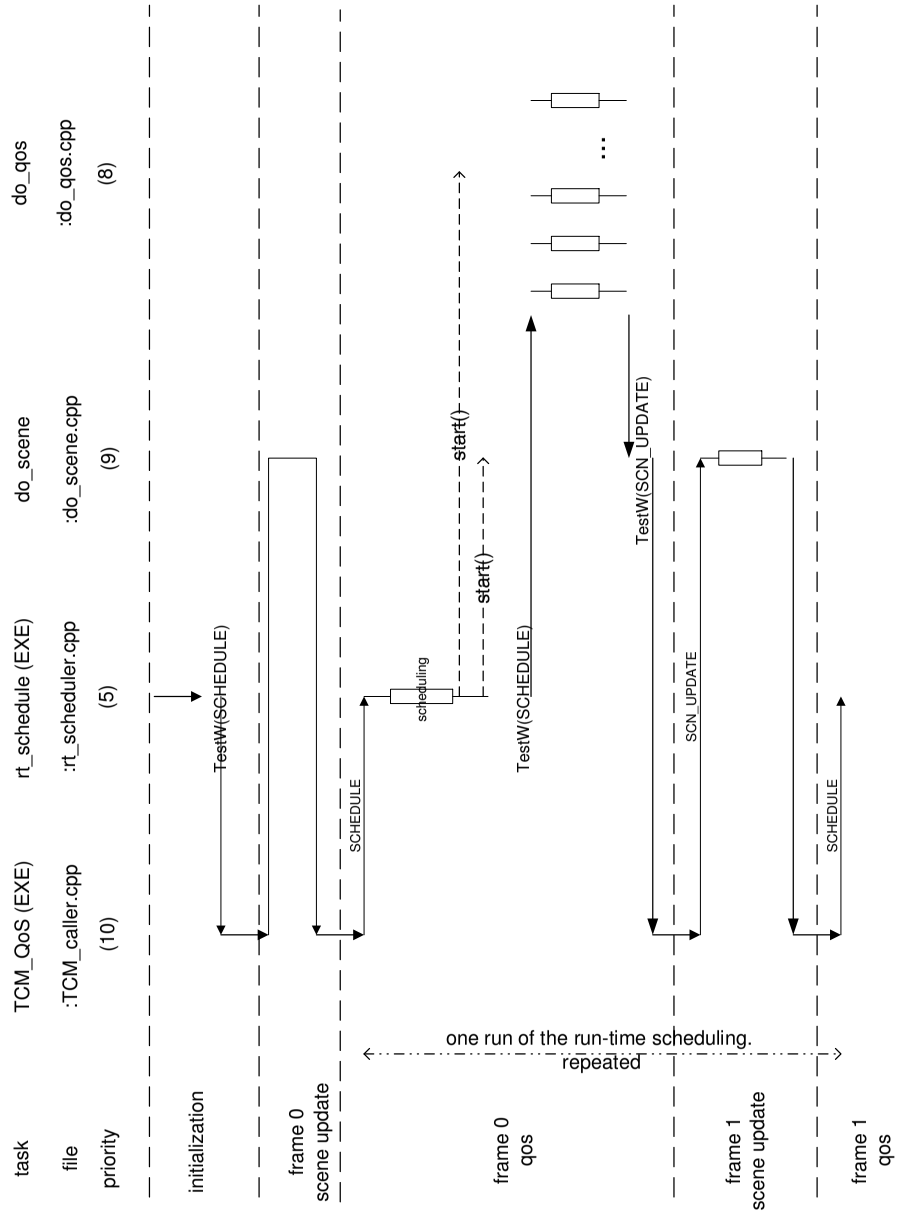


Figure 6.7: The implementation of the 3D QoS application scheduling.

stream frame is ready to start. Since *rt_schedule* and *do_scene* are waiting for semaphores and *do_qos* have finished their runs, *TCM_QoS* is the only ready task. It cleans up the current frame, prepares data for the next frame, then sends semaphore SCN_UPDATE and is preempted by *do_scene* immediately. This is the point where a new stream frame starts. *TCM_QoS* gets the control again after the *SceneUpdate* TF finishing its work, and activates the run-time scheduler by semaphore SCHEDULE. From that point on, the same sequence is repeated till the end of the application.

6.1.5 Reference Cases for Comparison

In the QoS kernel, which is typical for future object-based multi-media applications, we know at the beginning of each frame the characteristics of its content. In that case we know how many objects we have to render and also (in our approach) the best matched scenario of each object. Each scenario is represented by a Pareto curve computed at design time. From these, the run-time scheduler uses the heuristic algorithm of Section 4.2 [178] to select an operating point from each curve and activates it with the help of the RTOS. We have run the application for 1000 scenes and collected the data as a representative experiment.

For comparison reasons, we have also generated a reference case, REF2, to show how well a state-of-the-art slack-based DVS scheduler (like [153]) can do. We assume it has full access to the available application parameters at run-time too, but it does not exploit the scenario selection concept. In other words, it knows the number of objects it is going to render in that frame, but it does not exploit the scenarios. Therefore, it has to take an implementation that matches the worst case (TN 13 loops for 1000 times and the execution time is 68ms) for each object. However when one object finishes, the slack time (the difference between the real execution time and the worst case) will be reclaimed and used by the scheduler for the subsequent tasks (i.e. slack stealing [153] is exploited). Whenever the estimated remaining execution time is smaller than the desired deadline, a continuous DVS method is used to save energy.

Another reference case, REF1, is also generated, where all code is executed on the highest voltage processor. This is also the outcome if no information of the application is passed to the run-time manager, i.e. neither the number nor the kinds of the objects are known. Since we have a very dynamic application (the number of objects varies from 2 to 20), to handle the worst case and still meet the stringent deadline, the code has to be run completely on the high voltage processor. The majority of earlier techniques (see e.g. [68] for an overview) that use pre-characterized task data in terms of WCET times and corresponding energy, would lead to the REF1 result for applications with a

very dynamic behavior. Only a few would come close to REF2, as currently none of them combines all of the ingredients that were used to compose REF2 (see Chapter 2).

	REF1 (1 CPU, 1 V_{dd})	REF2 (1 CPU, 2 V_{dd})	TCM (1 CPU, 2 V_{dd})	TCM (4 CPU, 4 V_{dd})
fps=5	17.53J	14.32J	6.21J	4.93J
fps=10	17.53J	14.65J	9.49J	6.22J

Table 6.3: Energy consumption of the QoS application.

	REF1 (1 CPU, 1 V_{dd})	REF2 (1 CPU, 2 V_{dd})	TCM (1 CPU, 2 V_{dd})	TCM (4 CPU, 4 V_{dd})
fps=5	5	5	5	0
fps=10	26	26	26	1

Table 6.4: Deadline miss number of the QoS application.

6.1.6 Discussion of all results

The energy consumptions for all cases are shown in Table 6.3 and the number of deadline misses are shown in Table 6.4. All results are collected after the application has been run for 1000 sequential scenes and for different frame per second (FPS) requirements. The voltages we have used here are V_{dd}=1.2 and 2.4V for the two voltage case and V_{dd}=1.2, 1.6, 2.0 and 2.4V for the four voltage case. Normally, with a higher FPS, to satisfy a more stringent time constraint, parts of the code that are executed at lower voltages have to be moved to a higher voltage, resulting in the increase of the energy consumption. Also, the chance that a deadline is missed increases correspondingly. Compared to REF1, for the single processor situation, the TCM approach consumes much less energy (saving of 65% when fps is 5 and 46% when fps is 10), while the deadline miss ratio remains the same. The latter is easy to understand because when the time constraint is really stringent, the TCM method will automatically schedule all thread nodes to the highest possible voltage processor, which is just what REF1 does. When the time constraint is less tight, a much cheaper solution will be found by our TCM method. Comparing REF2 and REF1 you will find that for this type of dynamic multi-media applications, state-of-the-art DVS cannot gain much because it does not exploit different combinations of TF realizations. From the result we can also see that by increasing the number of processors, we can reduce the energy consumption even more while now meeting nearly all the deadlines.

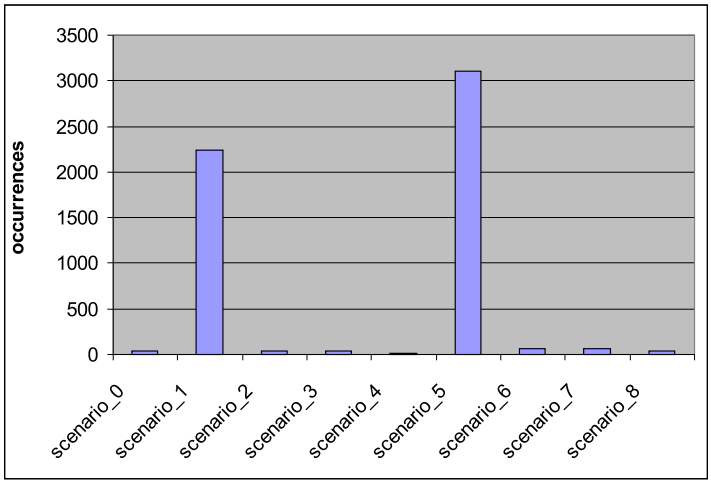


Figure 6.8: The distribution of the scenarios activation.

The distribution of the scenario selection is given in Figure 6.8, while Figure 6.9 gives the distribution of the selected Pareto points in scenario 6, both when fps is 5. From the figures we can see that the TCM scheduler activates different scenarios dynamically and selects the optimal Pareto point from the activated scenario depending on the run-time situations, i.e. the resource available and the number of competitors. Most of the time, scenario 1 and 5, which are the least time consuming ones, will be selected. Therefore, we avoid the worst case estimation and have more opportunities to scale down the voltage, compared to REF2. In scenario 6, the least energy consuming solution, Pareto point 5, is selected as long as it is possible; otherwise a more expensive one is chosen to meet the time constraint. The combination of the scenario and the Pareto point selection gives us the advantage of heavily exploring the design space at design time and finding the most energy efficient solution in accordance with the system's dynamic context at run time.

6.2 PocketGL Demonstrator on XScale Board

6.2.1 Overview

Up to now, the effectiveness of our approach was only conceptually shown based on simulations. In this experiment, our energy-aware dynamic task scheduling is applied to a 3D rendering application, showing a 3D animation on the screen.

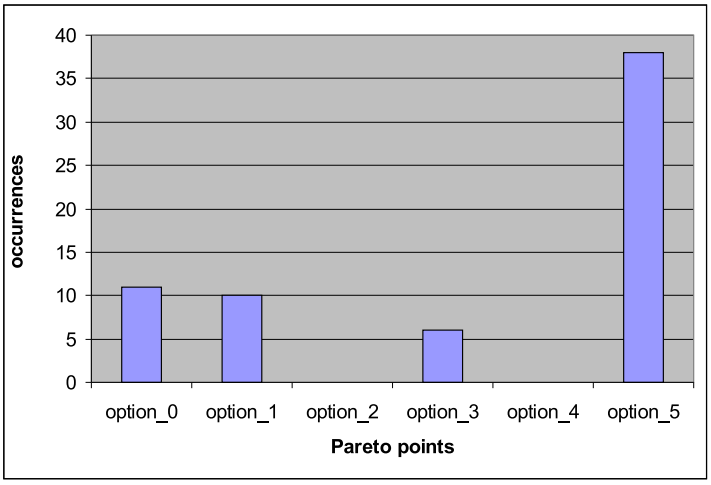


Figure 6.9: The distribution of the Pareto points activation in scenario 6.

This application is running on the Xingu development board of Acunia [2], with an embedded Intel XScale 80200 processor that allows two supply voltages (1.5V and ∼1V). Our run-time scheduler is integrated on top of a Linux RTOS. The board is also connected to an external power measurement device, to measure the total power consumed by the board without the LCD screen. With our approach, the supply voltage is dynamically adapted taking the variations in the scene, the number of visible objects, and the object complexity into account, to minimize the energy consumption while meeting the user-specified frame rate. This means, when the number of objects in the animation is low, the supply voltage is lowered; but when the number of objects becomes higher, or when these objects become more complex, more performance is required, and the supply voltage is raised. Compared with a standard implementation of the application where the clock frequency and the supply voltage of the platform processors remain constant, drastic energy reductions are possible. For this demonstrator, where the measured maximum energy gain is 53% (when the voltage is permanently lowered to 1V), the measured *average* energy gain for our scheduling is 40% compared with state-of-the-art DVS techniques.

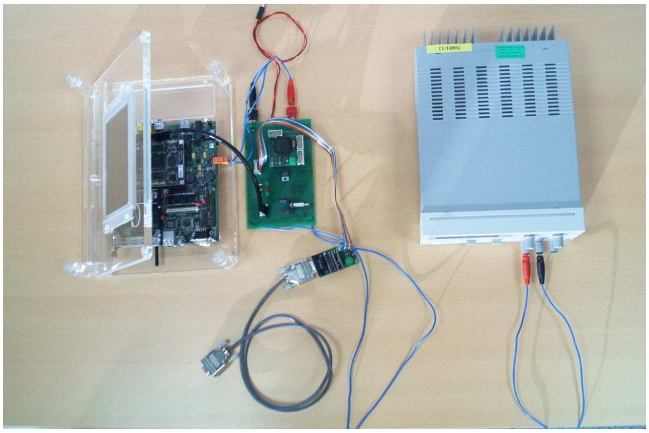


Figure 6.10: **Acunia board connected to the power measurement device.**

6.2.2 Demonstrator Setup

Target Platform

The target platform is the Xingu development board of Acunia [2], with an embedded Intel XScale 80200 processor [176]. This processor is a derivative of the ARM processor, and it has a cache memory of 32kB. It allows dynamic voltage and frequency scaling. Its default supply voltage is 1.5V, whereas its default clock frequency is 593.1MHz. The board has an LCD screen of $800*600$ pixels. It has also an external SDRAM memory, which is 64-bit wide, has a size of 64MB and runs at 100MHz, with a supply voltage of 3V.

DVS allows either a supply voltage of 1.5V with a clock frequency of 726MHz, or a supply voltage of $\sim$1V with a clock frequency of 528MHz. A Vdd switch takes on average 675usec before the board is completely stabilized.

To measure the power consumption, an external power measurement device is connected to the Acunia board. It measures the total power consumed by the board without the LCD screen. Indeed, this LCD screen normally consumes much more power than the XScale processor. The total power includes the idle power (i.e. the power consumed by the board when the processor is not active, including all the peripherals), the dynamic power consumed by the processor, the cache, and the SDRAM. The Acunia board connected to this device is shown in Figure 6.10. The idle power when the LCD screen is off is 3.688592W.

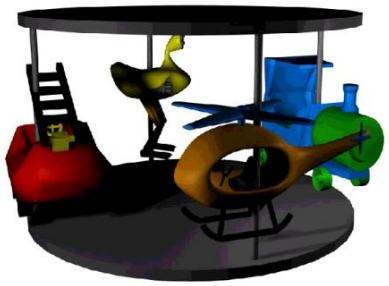


Figure 6.11: **Input scene for the application.**

The dynamic power consumed by a processor is modeled by:

$$P = C * Freq * V_{dd}^2, \tag{6.1}$$

where C is the effective capacitance of the processor, $Freq$ its clock frequency, and V_{dd} its supply voltage.

PocketGL 3D Rendering Demonstrator

The real-life application used in this experiment is a real-time 3D Pocket GL rendering application, showing a 3D animation on the screen. Pocket GL [120] is a 3D graphic library for pocket PC, allowing to draw 3D objects and manage 3D transformations. The application runs on top of a Linux RTOS. To allow dynamic voltage and frequency scaling, a compact-flash card with the correct Linux kernel must be introduced into the board, and a kernel module `xscale.o` that contains the needed `ioctl` functions, must be loaded. In the application, the rendering function is a task executed at each frame for each visible object. Hence the number of active tasks per frame depends on the number of visible objects. The complexity of each active task depends significantly on the object complexity (i.e. the number of its vertices), and the complexity of the object part to be rendered on the screen (i.e. the number of its visible triangles).

The input scene, illustrated in Figure 6.11, is a small roundabout, like the ones we find on fairs. Four objects hang on a pole: two highly detailed ones (the duck and the helicopter) and two low-complexity vehicles (the fire truck and the train). Two objects are also used to render the background of the

Scene object (Mesh)	Total number of triangles	Total number of vertices
`Duck`	52587	9941
`Helicopter`	41728	10404
`Fire truck`	6093	2117
`Train`	4952	1852
`Background1`	936	470
`Background2`	648	326
Complete scene	106944	25110

Table 6.5: **Scene complexity.**

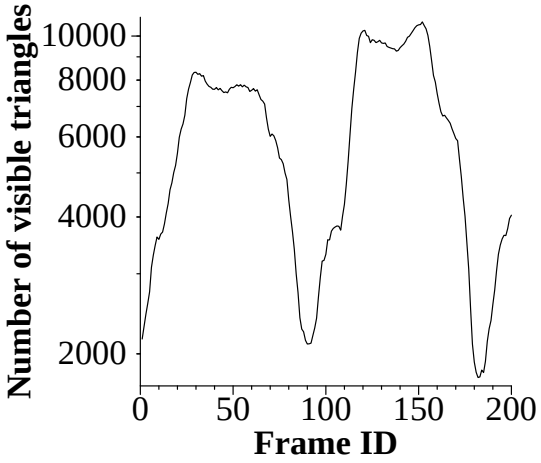


Figure 6.12: **Scene load variation.**

roundabout. While the camera positioned in the middle of the roundabout is rotating, the complexity of the objects to be rendered on the screen is changing. The complexity of these objects (called *meshes* in Pocket GL), characterized by their total number of vertices and triangles, is given in Table 6.5. From the 106944 triangles of the complete scene, the number of visible triangles per frame varies between 1721 and 10804 (see Figure 6.12), implying a load difference up to a factor 6.3 in the scene rendering. The maximum frame rate that can be achieved by running this application on the Acunia board is 11.75 Frame Per Second (FPS). In that case, the total power consumed by the board is 5.126W on average. The total power consumed by the SDRAM for this application is 0.208W on average.

6.2.3 Applying the TCM Approach

Gray Box Model Extraction

In the considered 3D application, for each visible object (i.e. mesh) on the scene, a separate thread frame, called `RenderMesh`, is activated. Its gray-box model is shown in Figure 6.13. `TransformMeshVertex` is a thread node transforming all vertex coordinates of the mesh depending on the camera position. `RenderSimpleMesh` (resp. `RenderComplexMesh`) is a thread node checking 1100 (resp. 5000) triangles at a time and drawing the visible ones on the screen.

Scenario characterization

Duck	Helic.	Truck	Train	Backgr1	Backgr2
4	4	5	5	1	1

Table 6.6: **Number of scenarios per mesh.**

From the gray-box model, we observe that: (1) the executed branch depends on the mesh complexity; (2) the number of thread nodes executed in this branch depends on the number of mesh triangles; (3) the complexity of these thread nodes depends on the number of non-visible triangles, the number of visible triangles but being out of the screen, and the number of visible triangles that will be drawn on the screen. These parameters are used to characterize all possible scenarios for this dynamic behavior of `RenderMesh`. For the input scene of Figure 6.11, 20 scenarios in total can be activated. Their distribution per mesh is shown in Table 6.6. To check the relevance of these scenarios, the application is profiled to verify for all thread nodes that the difference between worst, best, and average execution times are small.

For the design-time scheduling step, each scenario must be characterized with the average execution time and energy consumption of each thread node . In our case, this happens on the XScale processor of the Acunia board at the default supply voltage (1.5V) and clock frequency (593.1MHz). This is illustrated in Table 6.7 for one scenario of the `Helicopter` mesh. This is a complex mesh of 41728 triangles. Hence the thread node `RenderComplexMesh` is called 9 times. The reported average execution time is derived by profiling the application, whereas the average energy consumption is derived from the power model of the XScale processor (See Eq. 6.1).

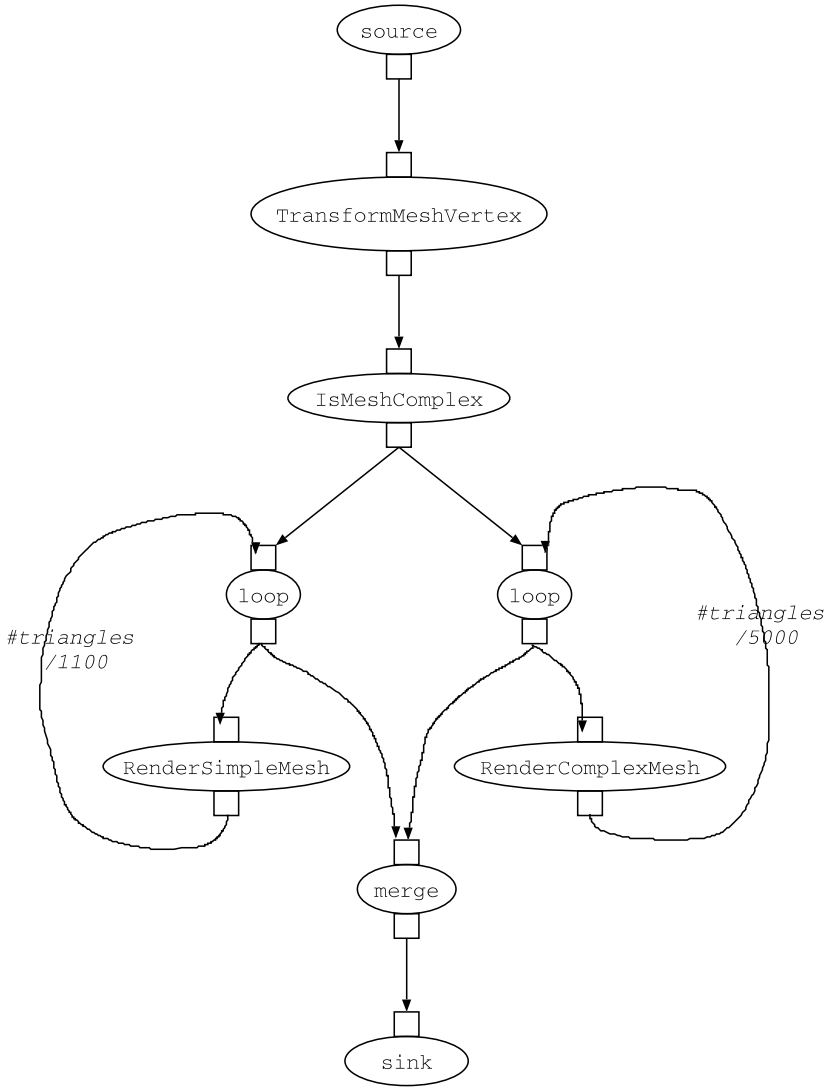


Figure 6.13: **Gray box model of thread frame `RenderMesh`.**

Thread node	Avg exe. time (usec)	Avg energy (uJ)
`TransformMeshVertex`	7344.31	5978.48
`RenderComplexMesh_1`	8437.25	6868.17
`RenderComplexMesh_2`	9631.79	7840.56
`RenderComplexMesh_3`	5444.63	4432.09
`RenderComplexMesh_4`	4411.97	3591.47
`RenderComplexMesh_5`	4440.55	3614.74
`RenderComplexMesh_6`	5920.69	4819.62
`RenderComplexMesh_7`	2718.05	2212.57
`RenderComplexMesh_8`	2370.17	1929.39
`RenderComplexMesh_9`	984.85	801.698

Table 6.7: **One scenario characterization for `Helicopter`.**

Design-time scheduling

The design-time scheduler is applied to each scenario of each mesh, taking into account: (1) the gray-box model restricted to the scenario; (2) the average execution time and energy consumption of each thread node as derived at the previous step; (3) the set of allowed supply voltages and corresponding clock frequencies, i.e. (1.5V, 726MHZ) and (1.09V, 528MHZ); (4) the average time to complete a Vdd switch on the board, i.e. 675usec.

The design-time scheduler explores different (Vdd, clock frequency) assignments to each thread node of the scenario and generates a Pareto curve, where every point is better than any other one in at least one way, i.e. either it consumes less energy or it executes faster. Since the design-time scheduler is performed at compile time, computation efforts can be paid as much as necessary, provided that it can give a better scheduling result and reduce the computation efforts of the run-time scheduler in the later step. E.g., for the scenario characterized in Table 6.7, the generated Pareto curve is shown on Figure 6.14. For instance, the Pareto point (0.056sec, 0.028J) assigns (1.5V, 726MHz) to thread nodes `RenderComplexMesh_i`, $6 \leq i \leq 9$ and (1V, 528MHz) to the other ones.

Run-time scheduling

At run time, at the beginning of each frame, the meshes that will be visible on the screen are first identified together with the `RenderMesh` scenario that will be activated. Then the run-time scheduler schedules all these active scenarios to satisfy the given frame rate and minimize the application energy consumption.

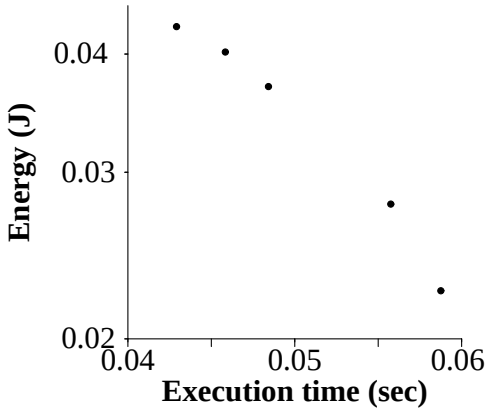


Figure 6.14: **One Pareto curve for** `RenderMesh(Helicopter)`.

Only some essential features of the points on the Pareto curves are passed to the run-time scheduler and are used to find a reasonable execution time distribution among all active scenarios. For each point on a Pareto curve, these features are: the execution time and energy consumption, and the (Vdd, clock frequency) assignment to each thread node. These features are stored in data structures whose size is $(8 + k * 24 + k * l * 20 + k * l * n * 8)$ bytes, assuming n thread nodes per scenario. Whenever a thread node function is called, the Linux function `ioctl(int d, int r, ...)` is executed, where `d` is the open file descriptor `DEVICE_FILE_NAME`, `r` is the request, and the third argument is either an input parameter or an output one. The request `IOCTL_SET_VOLTAGE` allows to set the Vdd, whereas the request `IOCTL_SET_FREQUENCY` allows to set the clock frequency.

We have run the application for different frame rate requirements, and `usleep` is called to put the processor to sleep whenever the scene is completely rendered and some slack time is available. We have run the application (1) using our scheduling; (2) using a state-of-the-art slack-stealing inter-task DVS [153]. This DVS scales the voltage according to the frame rate requirement, the worst case execution time of the scene rendering, and the slack time at the previous frame; (3) without any DVS, where the processor works at the fixed supply voltage 1.5V.

6.2.4 Experimental results

The average energy per frame for different frame rates and for a given leakage temperature is reported in Figure 6.15. It has been derived from: (1) the av-

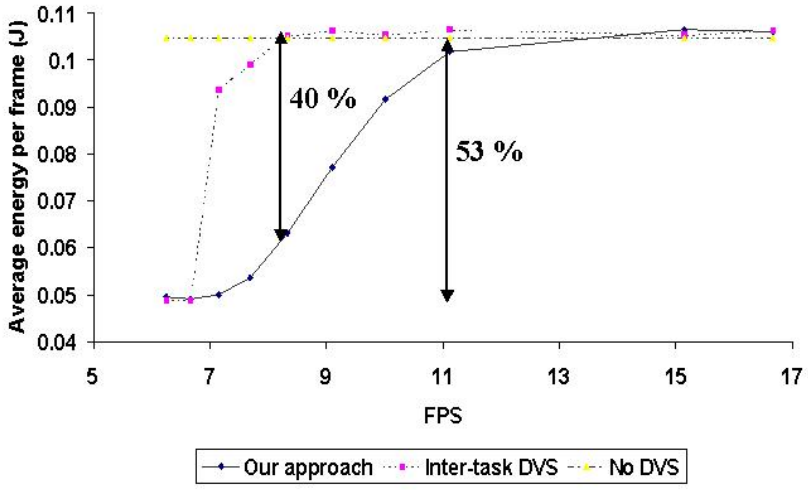


Figure 6.15: **Average energy per frame.**

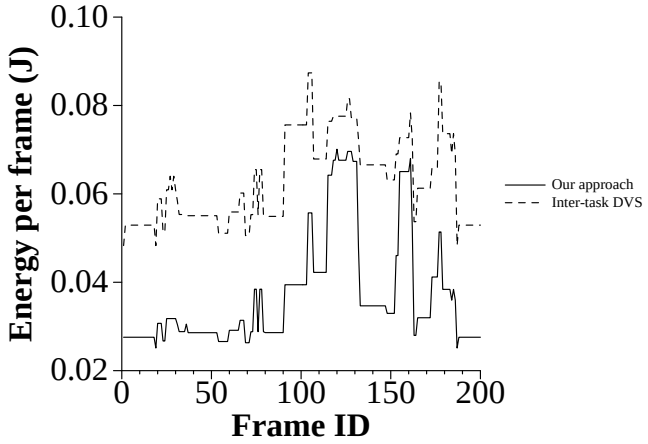


Figure 6.16: **Energy per frame during one roundabout rotation.**

	No DVS	Inter-task DVS	Our approach
Max. frame rate at 1.5V	11.75FPS	11.71FPS	11.62FPS
Perf. overhead	0%	0.34%	1.1%
Executable size	2.43MB	2.44MB	2.46MB
Overhead	0%	0.57%	1.14%

Table 6.8: **Scheduling overhead.**

erage processor power, i.e. the average total power consumed by the Acunia board without the LCD screen, and measured by the external device, minus the average power consumed by the SDRAM for this application (i.e. 0.208W, see Section 6.2.2), minus the idle power (i.e. 3.688592W, see Section 6.2.2); (2) the average execution time to render the scene. With higher frame rates, parts of the application code that are executed at lower voltages have to be moved to a higher voltage, resulting in the increase of the energy consumption. When the timing constraint is less tight, a much cheaper solution is found by our approach. E.g. when FPS = 8.33, the *average energy gain* is 40%, compared with the inter-task DVS. One observation is that, for this target platform, the measured *maximum energy gain* is 53% occurring when the voltage is permanently lowered to 1V [1]. Several reasons explain the difference between this average energy gain and the maximum one. Mainly, the low Vdd may not always be assigned without violating the frame rate requirement. The energy gain depends on the scene complexity. Energy consumption during one roundabout rotation for FPS = 8.33 is reported in Figure 6.16, where we see that the energy gain varies between 10.6% and 48.1%. Finally, some energy penalty is due to Vdd switch overhead (675usec) and to the run-time scheduler overhead. However this latter is quite small, as illustrated in Table 6.8. The size of data structures storing all Pareto curve information in our approach is 6908 bytes.

The state-of-the-art inter-task DVS cannot gain much for this FPS because it does not exploit different (Vdd, clock frequency) assignments to thread nodes in `RenderMesh`. Compared to our approach, it gives rise to a larger slack time after each scene rendering. This is illustrated in Figure 6.17. For more relaxed (resp. aggressive) frame rates, the energy consumption in both approaches becomes the same. This is easy to understand because both schedulers will automatically schedules all thread nodes to the lowest (resp. highest) possible Vdd.

[1] Average energy per frame with constant Vdd at 1.5V (resp. ∼1V) is 0.1046666J (resp. 0.0485006J). Hence (0.1046666-0.0485006)/0.1046666 = 0.5366 ∼ $(1.5^2 - 1)/1.5^2$.

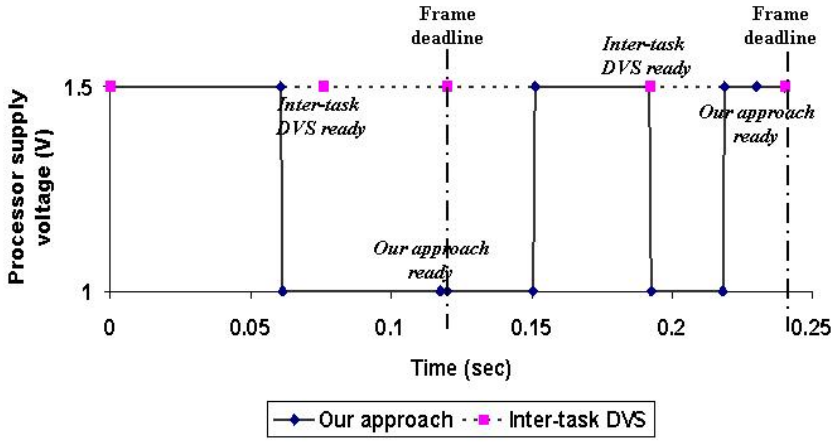


Figure 6.17: **Slack times for two successive frames.**

The occurrence percentage of the 20 possible scenarios for our input scene when FPS is 8.3 is shown in Figure 6.18. For each visible mesh, the run-time scheduler activates different scenarios dynamically. E.g., for `Background1` and `Background2` which are always visible, the only possible scenarios for these meshes are of course always activated. But for `Duck`, which is visible in 50.3% of the frames, Scenario 2 is only activated in 12.2% of the frames. Its Pareto curve consists of three points [2] , whose occurrence percentage is shown in Figure 6.19. Whenever this scenario is activated, the run-time scheduler selects the optimal Pareto point from the active scenario depending on the run-time situation, i.e. the number of visible meshes and their rendering complexity. In Scenario 2, the least energy consuming solution, Pareto point 2, is selected as long as it is possible; otherwise a more expensive one is chosen to meet the frame rate. The combination of scenario and Pareto point selection gives us the advantage of heavily exploring the design space at design time and finding the most energy efficient solution in accordance with the system's dynamic context at run time.

[2]Pareto point 0 $\sim (0.051sec, 0.051J)$, Pareto point 1 $\sim (0.64sec, 0.036J)$, Pareto point 2 $\sim (0.07sec, 0.027J)$.

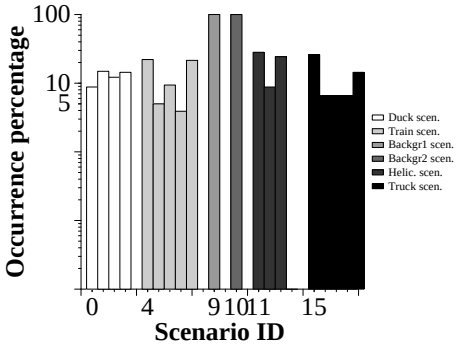


Figure 6.18: **Scenario distribution.**

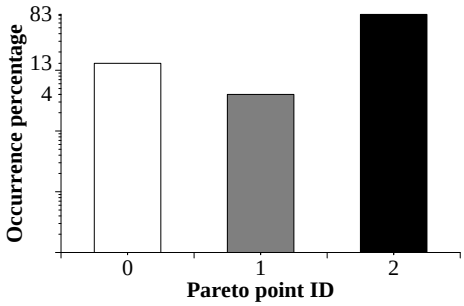


Figure 6.19: **Distribution of Pareto points in Scenario 2.**

6.3 Conclusion

In this chapter, we have validated our complete TCM design flow with two real-life demonstrators, using a Windows PC simulation or a real XScale board. Results from both demonstrators prove the effectiveness of our design methodology and illustrate how to integrate our design flow with applications and RTOSes. We consider only uniprocessor architecture for these two demonstrators. However, in Chapter 7, we will further show how to map and order tasks dynamically for a multiprocessor platform. New demonstrators are under development to fully explore the voltage scaling and (heterogeneous) multiprocessor features for new multimedia and wireless communication applications.

Chapter 7

Mapping and Ordering Tasks Dynamically on Multiprocessors

It has been realized that the large scale use of software programmable embedded processors will emerge as a key means to improve flexibility and productivity [102]. A range of processors will be used, to achieve different tradeoffs in time-to-market versus power, area or speed. For these heterogeneous multiprocessor system-on-chip (MP-SoC) platforms, a common key problem is the difficulty in refining and mapping the application to the platform.

The MP-SoC platform is different from traditional multiprocessor systems, which can be classified as multicomputer and multiprocessor as in [165]. Multicomputers are loosely coupled computer systems. Most typically, every computer has its own private memory space and its own (even different) operating systems. These computers exchange data by network and use protocols such as Remote Process Call to get service from other processors. Client and Server and Local Area Network systems are just two examples. Simply to say, it is a network-oriented system and the overhead to exchange data is very high. Multiprocessor systems are different in that they mainly exchange data by shared memory, which could be accessed by all or a few processors. In the extreme case, there is not even private memory and all memories are shared. Therefore these processors are tightly coupled together to complete difficult jobs such as scientific computation. Examples of multiprocessor systems include the IBM RP3.

Embedded MP-SoC systems are extremely tightly coupled and extremely cost

and power sensitive. Not like the general purpose multicomputer or multiprocessor systems, they are designed for a specific domain of applications. These characteristics decide it can not use the programming model designed for previous decoupled multiple processor systems. For instance, the Cache Consistency Protocol is very effective for multiprocessor systems and it removes the burden of keeping private copies of the same data consistent from the programmers. However, it is not practical to use the same protocol on embedded systems because it is extremely energy hungry. Instead, we can avoid that by smartly and carefully managing the data ourselves (see [105]).

The parallel programming models for embedded systems are still a challenge to the industry and are under research. Some people suggest a Network-on-Chip architecture and to use component based protocols such as CORBA and Java Remote Method Invocation [102]. The effectiveness of that approach is still to be proved. Here we do not assume any model but manage the inter-processor and inter-process communications by ourselves.

In previous chapters, we have discussed how to model a concurrent and dynamic application, how to apply the design-time and run-time scheduling. However, a middleware like layer is still needed to integrate the application with the RTOS and the hardware below. This integration should be generic enough so that it can be ported to almost all RTOSes; it should also be specifically embedded on the MP-SoC to avoid the high overhead of conventional multiprocessor programming model.

7.1 Dynamic Mapping and Ordering

After a Pareto point is selected by the run-time scheduler, the next problem is to find a systematic and generic way to map and order the TNs as decided at design time. Selecting the voltage in DVS is quite simple and for that purpose we do not need any special mechanism (only supporting APIs from the OS are needed). But to map and order TNs and TFs in an efficient and generic way is not that straightforward.

Consider the piece of C code below.

```
/* thread frame 1, video decoding */
int in[], out1[], out2[];
tf_1() {
  float c1, c2;
  tn_1(in, &c1, &c2);
  tn_2(in, c1, out1);
  tn_3(in, c2, out2);
```

```
}

/* thread frame 2, audio decoding */
int out[];
tf_2() {
  int buf[];
  tn_A(in, buf);
  tn_B(buf, out);
}
```

This is an example of two TFs, which have 3 and 2 TNs respectively. Their CDFG representation can be found in Figure 7.1. Notice that no dependency

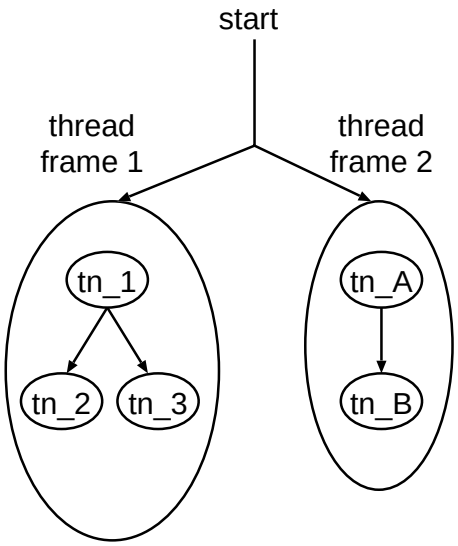


Figure 7.1: The gray-box model of a simple example.

exists between tn_2 and tn_3 of thread frame 1 and thus they can be executed in parallel. To compile the code to a dual-processor platform, we want to achieve several goals: a) the code is written in plain ANSI C (not any concurrent C dialect, e.g. SystemC); b) it should be compiled and linked by a normal compiler; c) only at run-time we will decide the execution order of the TNs and on which processor to execute them.

This is not easy remembering all the codes are compiled and linked statically, which in fact fixes the addresses of all the functions and global variables. Moreover, embedded multiprocessor programming requires well synchronized code

and protected data. It becomes even worse when we assume a dynamic and open system, which allows new TFs to come and join the running applications at any moment (however, the new coming TFs will wait till the next scheduling point to be scheduled and executed). One obvious solution is to wrap every TF and TN in a process/thread structure provided by the RTOS and to apply the tricky and error-prone multiprocess programming method. However, this will cause frequent switches between the process space and the RTOS space and frequent context switches among processes. It also requires inter and/or remote process communication mechanisms. All these are expensive, especially for multiprocessor platforms.

To handle the problems in a systematic and generic way, we have wrapped every TF into an object, which contains an initializer, a scheduler and a TF specific data structure. The scheduler keeps a set of function pointers. Every Pareto point just means a different set of values of these pointers. Whenever a new TF enters the system, its initializer is first called to register itself to the system. Then for a given Pareto point, the scheduler resets its pointers to the desired TNs in the appropriate order. Therefore, the scheduler can execute the TNs by referring to the function pointers in the given order, and map them accordingly. Figure 7.2 shows the execution scenario of one possible scheduling decision of our example.

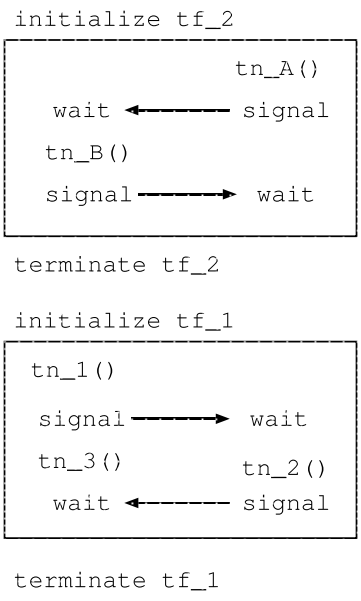


Figure 7.2: One possible run time mapping of the simple example.

Our approach has a low overhead because the complete code is held in one single process space and no unnecessary context switch involved. Meanwhile, it provides an easy solution to achieve a flexible and open system.

7.2 Experimental System Setup

In this section the conceptual approach of Section 7.1 will be demonstrated with a real platform and a RTOS.

7.2.1 The Experimental Platform

As shown below (Figure 7.3), the board consists of two TI C6202 DSPs, each

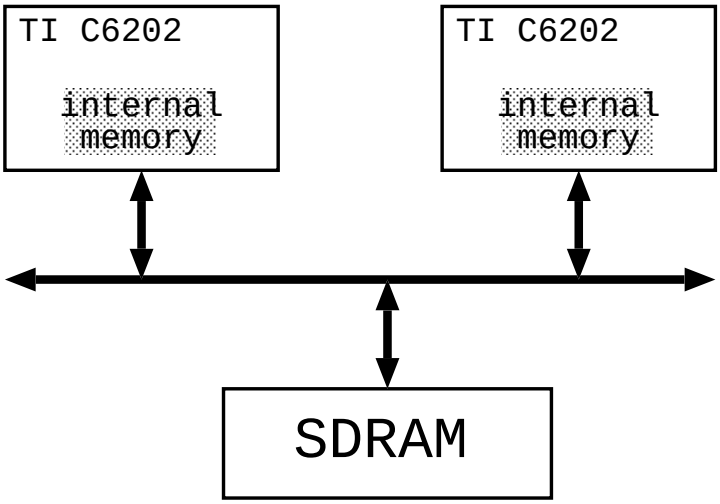


Figure 7.3: The experiment board.

with a 128KB on-chip memory. The on-chip memory is divided into two halves, one for stack and the other is program addressable and free for use. An off-chip 32MB SDRAM is also available. It holds the program codes, private data and a piece of shared memory. The DSPs and the SDRAM are connected by a bus. No data cache is present, either on chip or off chip, which means: a) slower data access speed. However, we can either implement a memory allocator, handling the internal on-chip memory as a full software-controlled cache, or simply address and maintain it ourselves in the program. b) no costly overhead is paid to obtain cache consistency and coherency which are anyhow

mainly useful for big multiprocessor systems running general-purpose software. The embedded software designer typically has better knowledge and full control over his system, enabling him to avoid and manage data conflicts in a more efficient way.

On every DSP, a copy of the Virtuoso Real-Time Operating System (RTOS) [172] is running. The RTOS provides the run-time environment, memory and task management, IO control and interrupt handling. However, Virtuoso is more a stand-alone OS than a tightly-coupled OS, in the meaning that every processor runs independently, communicating and changing data with its peers only when explicitly asked to do that.

7.2.2 The Run-time System

Our run-time system module runs like a middleware layer (Figure 7.4). It clearly separates the application from the lower level RTOS, giving the same APIs even on different operating systems. It is compatible with most current RTOS implementations as long as they have well defined APIs for task activation and synchronization. Therefore, it is easy to be ported and can be used as an integration component in a heterogeneous multiprocessor platform.

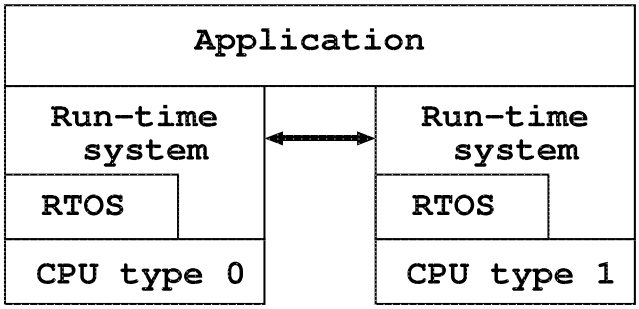


Figure 7.4: Our run-time system separates the application from the RTOS and the hardware below.

The run-time system performs the run-time scheduling hierarchically (Figure 7.5): the system side is responsible for managing Pareto curves, finding a Pareto point and dispatching the TFs, while the TF side does the real mapping and ordering based on the Pareto point selected by the run-time system. Whenever a new TF enters the system, it first registers itself to the run-time manager. At the next scheduling point, triggered by an interrupt from a timer or other application related events, the system side scheduler, taking into account the Pareto curves of all active TFs and running the algorithms presented

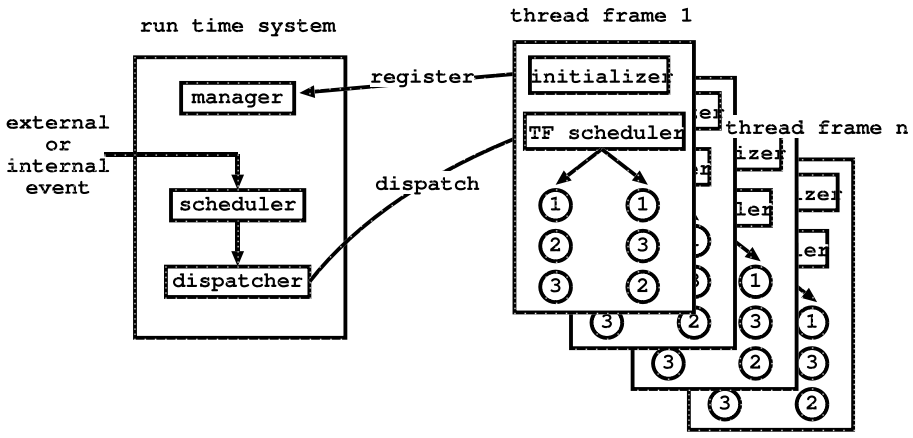


Figure 7.5: The run-time system.

in Chapter 4 and Chapter 5, finds when and which TF will be executed. Basically, this involves ordering the TFs, e.g. based on their priority, and selecting a Pareto point from the Pareto curve accompanying every TF, given the system restriction (e.g. the number of a specific resource) and an objective to optimize (e.g. the energy consumption or quality of service). Having decided the order and selected the option, the system dispatcher just calls the TF side schedulers one by one, passing the selected Pareto point to the TF. Accordingly, the TF side scheduler selects the appropriate implementation of its functions.

The hierarchical scheduling makes the system more dynamic, reusable and flexible. The run-time system does not have to know the TFs, which can be any TF, as long as they have a uniform API. The run-time system does not have to worry how to map and order the internal components of each specific TF. They are done by the TF code itself, which is generated at compile time with all the necessary details to achieve this.

Different to normal task scheduling, our implementation avoids the expensive task management service provided by the RTOS. Actually, except to the background managing thread, only one thread is running on each processor.

7.3 Implementation

We use two data structures to encapsulate the necessary data, as shown below.

```
/* system level management data structure */
typedef struct {
```

```
/* data */
int n_tfs; // number of thread frames
TF_QItem *head; // the head of the thread frame queue

/* function pointer */
TCM_Sys_Sched sys_scheduler; // pointer to the system side scheduler
} TCM_Sys;

/* thread frame interface */
typedef struct {
/* data */
int tf_id; // thread frame id
int prio;  // thread frame priority
int n_tn;  // number of thread node
void * data; // pointer to thread frame specific shared data
P_Curve *pc; // Pareto curve of the thread frame

/* function pointer */
TF_Init tf_initializer; // pointer to the thread frame initializer
TF_Sched tf_scheduler;  // pointer to the thread frame scheduler
} TF_If;
```

TCM_Sys stores the system level information, including the number of TFs, a pointer to the TF queue and a system level scheduler. Every thread frame is an instance of the TF_If, which keeps a function pointer to the initializer and a function pointer to the thread frame scheduler besides important data such as the priority level and the Pareto curve of that thread frame. Every TF can have its specific data. This data is shared between the TNs of that thread frame and mapped to the external shared memory so that it can be accessed from either processor.

There are totally three threads running in the system (see Figure 7.6). Thread *background management* is responsible for the system level management. It calls the initializers of the TFs, executes the Pareto curve based run-time scheduling, stores the scheduling decision and then calls the thread frame side scheduler. Thread *node 0* and *node 1* are the threads to implement the functions of the thread frame schedulers, the detail of which is given later. Thread *background management* and *node 0* are mapped to processor 0 while thread *node 1* is mapped to processor 1. For the run-time scheduling shown in Figure 3.8, we obtain the execution sequence as shown in Figure 7.6.

The thread frame side scheduler is coded as follows.

```
tf1_scheduler() { // the scheduler of thread frame 1
```

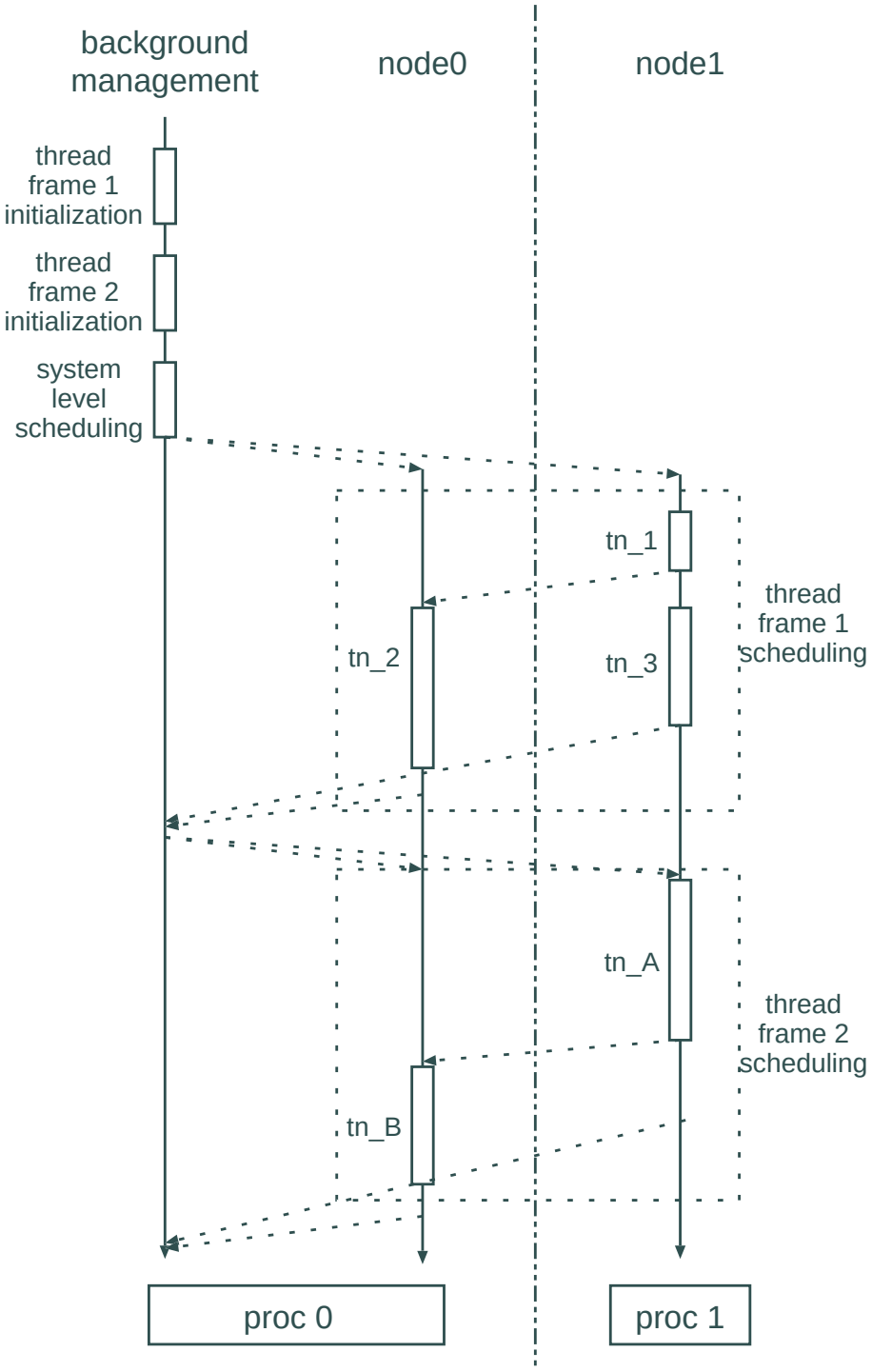


Figure 7.6: The execution sequence of the example shown in Figure 3.8. Thread *background management* and *node 0* are mapped to processor 0 while thread *node 1* is mapped to processor 1.

```
  if (node0) { // if it is executed in thread node 0
    switch(Pareto point) {
      case 0:
             semaphore_wait(GOON0);
             tn_2(); // execute thread node 2
      case 1:
             tn_1(); // execute thread node 1
             semaphore_signal(GOON1);
             tn_3(); // execute thread node 3
      case 2:
             ...
             ...
      case k:
             ...
    }
  } else {    // if it is executed in thread node 1
      case 0:
             tn_1(); // execute thread node 1
             semaphore_signal(GOON0);
             tn_3(); // execute thread node 3
      case 1:
             semaphore_wait(GOON1);
             tn_2(); // execute thread node 2
      case 2:
             ...
             ...
      case k:
             ...
  }
}
```

Thread *node 0* and *node 1* both call the function tf1_scheduler. However, they will execute different sections of that function. Depending on the Pareto point set by the system level scheduler, tf1_scheduler changes the mapping and ordering of the TNs accordingly (compare case 0 and case 1 in the code). This implementation is generic enough to work on any processor and RTOS, as long as priority-based scheduling is supported and the inter-processor and inter-process communication is allowed. It could also work on heterogeneous platforms, though more low-level integration problems may need to be solved, which we still have to find out.

7.4 Experiments and Results

We have applied our methodology and implemented it for two applications on the platform introduced in section 7.2.1. In all of our experiments, the codes are not deliberately optimized for our TI C6000 VLIW architecture. We are not discussing how to optimize the code at the instruction or data size level, which is out of the scope of this thesis and can be done with other state-of-the-art approaches. Here we consider only the task level parallelism. All the results shown later are free of instruction and data level parallelism optimization. In fact, the contents inside the bubbles on Figure 7.7 are untouched compared to the original MediaBench code. So the TNs are indeed "atomic" as intended by our gray box approach. However, this will not reduce the effectiveness of the methodology we describe here.

7.4.1 Experiment to Explore the Overhead

The first simple example we have investigated on our dual-TI platform is based on the DCT encoder from MediaBench [109]. This simple code has been used in an experiment designed deliberately to explore the overhead behind our run-time system.

The DCT encoder uses the discrete cosine transform (DCT) to compress a pixel image by a factor of 4:1 while preserving its information content. The encoder divides the image into blocks, each containing 8x8 pixels, as shown in Figure 7.7. For every block, the encoder reads the image into a buffer, finds its

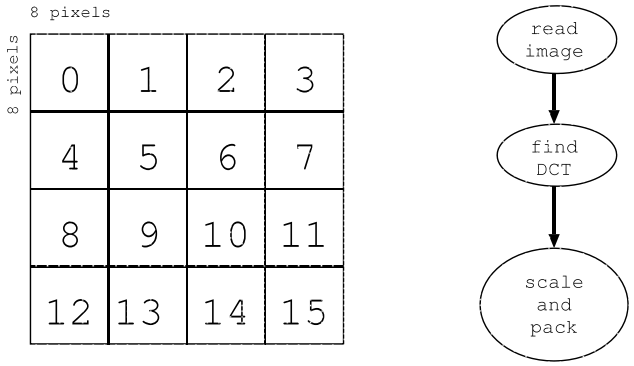


Figure 7.7: The DCT encoder.

DCT, then finally scales and packs it to the output buffer.

The original code (dct in Table 7.1) takes 1.7 million cycles to compress an

image of 32x32 pixels (4x4 blocks). This number is high for two reasons. Firstly, the TI C6202 has no floating point units, while the encoder involves a lot of floating point computation. Secondly, the DSPs are configured to work without program and data cache.

Next, we have adapted the encoder to our framework. A TF wrapper is generated for every image block. As shown in Figure 7.8, every TF of dct_tf has three TNs, R, F and S (representing the three bubbles in Figure 7.7). It has two possible schedulings, namely *scheduling*0 and *scheduling*1. If *scheduling*0 is selected by the system-level scheduler, the TF will first execute TN R on DSP0, then TN F on DSP1 and finally TN S on DSP0. The decision looks artificial and arbitrary, but it can verify whether our run-time system is running functionally correct and it can give us the overhead number behind our run-time system. Since one image block is put into one TF, an image of size 4x4 blocks will generate 16 TF wrappers, i.e. 16 Pareto curves, each with two Pareto points. The execution time of dct_tf, including all the overhead of our framework, is 2.18 million cycles (Table 7.1) for the 4x4 image.

In dct_tf, at any moment, only one DSP is running the functional TN, due to the sequential feature of the block compressing. To take advantage of the multiprocessor architecture, dct_dual wraps the TNs of every two block into one TF (Figure 7.8, R0 means the R TN of block 0, totally 6 TNs in one TF), which allows the TF scheduler to explore the task level parallelism and improve the performance accordingly. Again the schedulings given here are decided arbitrarily. For the same 4x4 image, it now takes only 1.43 million cycles, which is a 17% improvement over the original code.

As shown in Figure 7.8, different compositions of the TFs have been tried. We have also tried a bigger image with 8x8 blocks and the results are summarized in Table 7.1.

	4x4 blocks		8x8 blocks	
	no. of TFs	exec. time (M cycles)	no. of TFs	exec. time (M cycles)
dct	1	1.71	1	6.56
dct_tf	16	2.18	64	7.50
dct_tf2	8	2.08	32	7.16
dct_tf4	4	2.04	16	6.98
dct_tf8	2	2.02	8	6.89
dct_dual	8	1.43	32	4.52
dct_dual2	4	1.29	16	3.98
dct_dual4	2	1.26	8	3.83

Table 7.1: Execution time of different TF compositions of the DCT encoder.

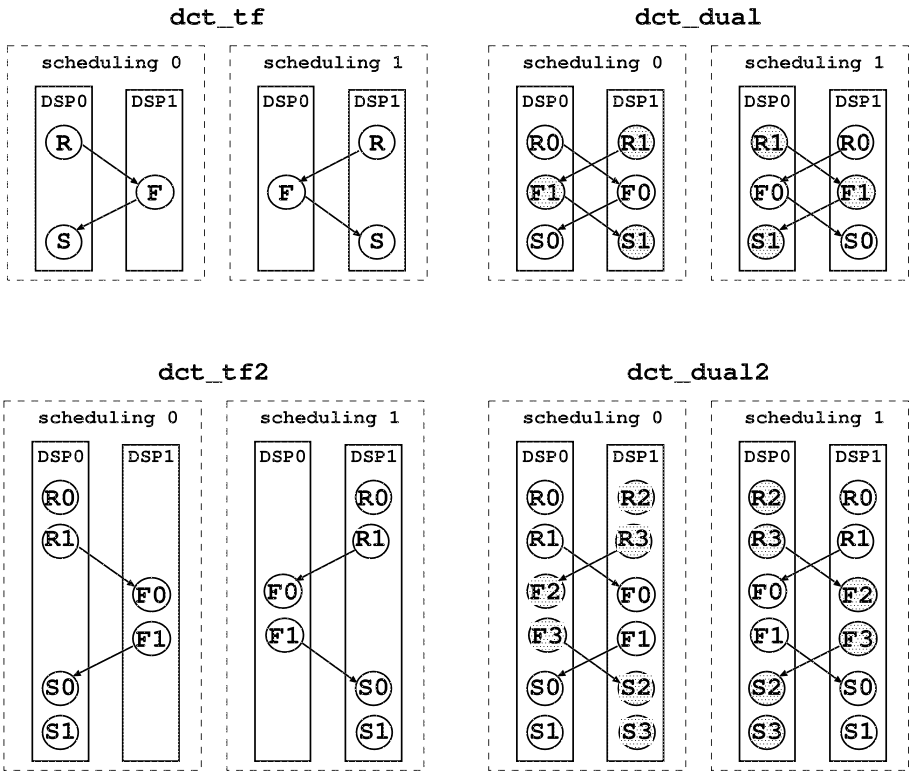


Figure 7.8: Different TF compositions of the DCT encoder.

From this table, we can see that the overhead of our run-time scheduling layer constantly drops with the decrease of the number of TFs, in both the dct_tf and dct_dual series for the same input image. In fact, this overhead can be decomposed into two components: the overhead per system and the overhead per TF (see Table 7.2). The former comes from the initialization and the booking of the system, and it is more or less the same for every system. For every TF running on the system, we also have to pay some overhead to start it, send data to it and synchronize it. Hence, the total overhead can be expressed as $n * a + b$, where n is the number of TFs (see Table 7.2). One obvious observation from it is that the per TF overhead of dct_dual is more than two times higher than dct_tf. This can be explained by Figure 7.8: dct_dual TFs have twice the amount of data communication and synchronization as dct_tf.

The per TF overhead is only 11K cycles in the dct_tf series and 29K cycles in the dct_dual series. This is acceptably low if the TF code is big enough, which

	dct_tf series		dct_dual series	
	4x4	8x8	4x4	8x8
a(per TF overhead)	11.19	10.88	29.11	29.50
b(per system overhead)	282	247	333	281

Table 7.2: The decomposition of the overhead, in K cycles.

is not the case here for this simple illustration but is true for any real-life application that we would consider like the H.263 test case in the next section.

Another interesting issue is the performance improvement from the dct_dual cases, which takes advantage of the task level parallelism exposed by our design methodology. For an image size of 4x4 blocks, the dct_dual and dct_dual4 improve the performance by 17% and 27% respectively. For an image size of 8x8, the improvements are 31% and 42%, which is already close to the 50% theoretical upper bound.

7.4.2 The Realistic H.263 Test Case

We have also investigated the ITU-T H.263 application, which is an international standard for video conference and other low-bit-rate video streams.

We have used the Telenor C exemplary implementation code, tmn-1.7. Except to necessary changes to enable our approach, we made no more optimization, neither the algorithm level nor the instruction level. In that code, the stream can basically have 3 different kinds of video frame: I, P, and PB. I frame is also called intra-frame because it is encoded only using the information of that video frame and does not depend on any other frame. From time to time, we have to insert an I frame because either we have a completely new video stream (e.g, the editing) or we have to get rid of the accumulated noise from predicting. P frame is forwardly predicted from another I or P video frame, by using Motion Compensation. PB frame actually contains the information of two frames. First a P frame (frame i+2) can be decoded from its previous I/P frame (frame i), then another frame, frame i+1, can be predicted forwardly from frame i and backwardly from frame i+2 and inserted between them. One sequence is that for the next period the processor can be idle because in the previous period it has already generated the video frames for both the previous and current period. For every I, P or PB frame, the code can be separated into two nodes, the decoding node and the conversion node. The decoding node does all the work related to reading in the data, entropy decoding, rescaling, idct and motion compensation, finally generating data in YUV format. To really show the video, we still have to convert it from YUV format to RGB (which is understandable by the display) and store it in the display memory, which is

done by the conversion node. Depending on the type of frame (I/P/PB) and the size of image, these two nodes will take a different number of processor cycles.

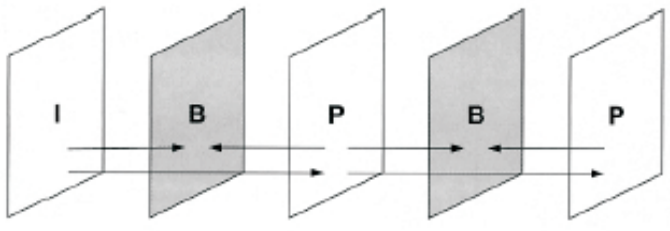


Figure 7.9: The I, P, and PB frame of H.263.

To simulate the dynamic behaviors of future applications (e.g. Philips' WWICE interface), we manipulate 5 video streams simultaneously where the frame size of each stream typically differs. Strm0 is the combination of different CIF clips from the standard benchmark Akiyo, Coastguard, Container, Foreman and Hall video streams. Every clip lasts for 100 frames. This is used to simulate the main video stream one is watching. At the same time, we have four streams (Strm1, Strm2, Strm3 and Strm4), which are generated from shorter video clips (5-50 frames each) in QCIF format. These streams are used to simulate the user triggered events. For instance, when a user is watching TV, he may talk to another person by the video phone and browsing on line, all on the same platform. For Strm1-4, between clips we randomly inserted 2-12 idle frames, to simulate the idle time of the user.

At the beginning of every period, the applications read in the frame headers of all the five streams, to see what kind of frames it is going to handle for the current period, then maps and orders these nodes on our experimental board. We have used streams of 1000 periods and the results are summarized in Table7.3.

We have compared the result of our dynamic mapping and ordering approach to 3 reference cases. In the first case (single in Table 7.3) we put all the nodes onto a single processor. It is used to give a reference on how a single processor performs. In the fixed_0_4 case, we put nodes of Strm0 on one processor (it is in CIF format and requires much more execution time) and Strm1-4 on another processor. In the fixed_1_3 case, we put both Strm0 and Strm1 on one processor, and Strm2-4 on another processor. When the frame rate is 20, case "single" will cause 524 deadline misses, which is more than half of the periods, while case fixed_0_4 and fixed_1_3 cause 245 and 131 deadline misses, which are better but still more than 10% of the periods are hardly usable. Under the same

condition, our dynamic case will cause only 39 deadline misses, which is only 3.9% of the total periods. When the frame rate is relaxed to 15, the deadline misses caused by the fixed_0_4 and fixed_1_3 are 53 and 1, while our dynamic case meets all the deadlines. We can notice that fixed_1_3 performs much better than fixed_0_4, but the exact number of deadline misses of fixed_1_3 depends on the input streams and can be worse than what we show here. For both frame rates, our solution is always best because it can adapt to the real need of that period. In all cases, the results of our dynamic solution have already included the implementation and scheduling overhead.

	fps=20		fps=15	
	deadline miss	energy(J)	deadline miss	energy(J)
single	524	14.71(no DVS)	265	14.71(no DVS)
fixed_0_4	245	10.75	53	9.17
fixed_1_3	131	10.14	1	8.07
dynamic	39	9.12	0	7.34

Table 7.3: The deadline miss and energy consumption for 1000 period.

Another advantage of our dynamic mapping and ordering approach is that it can increase the energy saving impact of DVS. Since the board we use does not support DVS, we have only simulated the effects of DVS on a PC by checking the available slack time of every period. As long as a DVS-compatible processor (e.g. XScale) is on our experimental board, we can really implement it with the same approach.

According to TI's datasheet, the power consumption of the 300MHz C6202 CPU core is 300mW. We assume the working frequency can be continuously slowed down to as low as 200MHz, and the supplying voltage will proportionally scale down from 1.5V to 1.0V. This assumption is commonly taken also by other academic researchers [152]. Since the frame header of each stream is decoded first at the beginning of every period, it enables us to make an accurate estimation about the execution time of every TN to be executed in that period. With the above assumptions, we have applied DVS on fixed_0_4, fixed_1_3 and our dynamic case. The results (Table 7.3) show that our dynamic approach has 10% (fps=20) and 9% (fps=15) energy savings even compared to fixed_1_3, which is the best possible result if the tasks are mapped statically by the state-of-the-art DVS technique [7]. Compared to the non-DVS original case, the energy saving is 38% (fps=20) and 50% (fps=15).

7.5 Conclusion

In this chapter we have presented a systematic approach on how to insert a middleware layer between the application and the RTOS to map and order tasks dynamically for multiprocessor platforms. Two experiments have been done on a real dual-TI experimental board. This practical proof is one contribution compared to the normal simulation approach taken by other researchers. A simple DCT example is used to illustrate the overhead introduced by our middleware layer. An H.263 example shows the large impact of our approach on real-life applications, where the deadline miss rate is dramatically reduced. When DVS is considered, an 10% energy saving has been achieved compared to the state-of-the-art approach.

Chapter 8

Conclusions and Future Work

This doctoral research has been performed in the context of the Task Concurrency Management (TCM) project at IMEC, which aims to provide a systematic methodology and design flow for the complex embedded software design for highly concurrent and highly dynamic applications.

The merging of computers, consumer and communication disciplines gives rise to very fast growing markets for personal communication, multimedia and broadband networks. Technology advances lead to platforms with enormous processing capacity that are however not matched with the required increase in system design productivity. One of the most critical bottlenecks is the very dynamic and concurrent behaviors of many nowadays multimedia applications. Normally first specified in software oriented languages (like Java, UML, SDL, C++), these applications have to be executed at real time in a cost/energy-sensitive way, most probably on heterogeneous System-on-Chip platforms. A systematic way of mapping these software specifications onto embedded multiprocessor platforms is required. The fully design-time based solutions, as proposed earlier in the compiler and system synthesis communities, can not handle the problem properly. They can only solve the problem by assuming the worst case situations, which results in very costly designs.

In order to deal with these new dynamic applications where tasks and complex data types are created and deleted at run-time based on non-deterministic events (typically at the rate of tens of ms), a novel system design paradigm is required. TCM tackles this problem by first describing the system with a MTG* based gray-box model [150]. This allows us to expose the key dynamic

and concurrent features of the system while hide all unnecessary details. Then a transformation step can be applied to improve and optimize the system concurrency. After that, a two-phase scheduling approach is proposed to allocate, map and order the tasks of the system onto the multiprocessor platforms. The design-time scheduler explores the design space and pre-stores the exploration results, while the run-time scheduler is used to optimize the system performance/cost according to the system dynamic context and the pre-computed information.

In this thesis, we mainly focus on the algorithms that the run-time scheduler needs to make system-level dynamic tradeoffs and the implementation of such a scheduler on top of normal RTOSes. Two real-life demonstrators are also used to verify the effectiveness of the complete design flow.

8.1 Contributions

The main contributions of this thesis work are the following:

1. In Chapter 3, the complete TCM design methodology has been defined. This is the cooperation work with several other PhD students in the TCM team, in particular Chun Wong, Paul Marchal, Stefaan Himpe and Aggeliki Prayati [180, 183, 175, 169, 106]. In this methodology, an application is first modeled with the MTG* based gray-box model. Then a transformation step is used to further increase and expose the parallelism of the application so that it can be better mapped to the multiprocessor platforms. After that, a hybrid design time and run time scheduling approach is proposed mainly for three reasons. First, this scheme better optimizes the embedded software design. Second, it gives the entire system more runtime flexibility. Third, it reduces runtime computation complexity.

2. In Chapter 4 and Chapter 5, two algorithms are proposed for the run-time scheduling. The first algorithm schedules the applications sequentially while exploring the tradeoffs dynamically [178]. This algorithm is best used for systems with fewer processors (e.g. less than 4) because of its speed and solution quality. When the system comprises a larger number of processors, the second algorithm allows several applications to run simultaneously, each using a part of the platform. This typically leads to better usage of the processors available in the system and results in shorter completion time or less energy cost. However, its computation overhead is higher than the first algorithm.

3. In Chapter 6, two real-life demonstrators are used to verify the effectiveness of our TCM design methodology. The first is a 3D Quality of

Service application and it is simulated on PC [181, 182]. The second is a 3D rendering application implemented on a real XScale board. Dynamic voltage scaling is enabled by the enhanced Linux operating system and the system power consumption is monitored at real time.

4. In Chapter 7, a middleware-like module is developed to map and order tasks dynamically on a multiprocessor platform, as decided by the run-time scheduler [179]. This module provides a generic method to integrate the application, the run-time scheduler and the Real-Time Operating System at a low overhead. Applicability of this module is tested by a real dual-DSP platform with applications such as DCT and H.263.

8.2 Future Work

Many opportunities exist to further extend the TCM research, making it more efficient, applicable to more situations and more easily to use. Below is an overview of some interesting topics for further research.

- Currently, the TCM approach is done step by step, most of the time manually. The result of the previous step has to be interpreted and transformed by the designers, helped with some tools, to make it understandable and usable to the next step. An integrated automatic or self-automatic tool chain is highly desirable to alleviate the effort and the involvement of the users and to improve the quality of the design. Work is ongoing to achieve this in the TCM alpha team at IMEC.

- We have developed demonstrators for uniprocessor and homogeneous multiprocessor platforms. To further verify our methodology and to provide more space for design exploration, demonstrators on real heterogeneous multiprocessor platforms are needed. The possibility of integrating several different RTOSes on the same multiprocessor platform is also interesting.

- The quality of the overlapping run-time heuristic can still be improved. New techniques are needed to prune the local search space and let the heuristic converge faster. When the global deadline is loose, an extra heuristic may be added to further improve the solution quality.

- For applications running on heterogeneous multiprocessor platforms, many idle time periods may exist on some processors, due to data/control dependencies and/or intra thread frame time constraints. To reduce these idle periods, interleaving of thread nodes from different thread frames is a promising solution. However, this will break the boundary of thread

frames and further work is needed to make sure it will not reduce the benefits which cause us to separate applications into thread frames and thread nodes. Currently Zhe Ma is working on this [101].

- At the run time scheduling stage, we assume we can have the full knowledge of the system dynamic context at that moment. It is also possible we have only partial knowledge of that dynamic context but are still required to make a schedule. In that case, we may schedule the system first based on the partial knowledge and our prediction and then refine the scheduling later when the full knowledge is finally available.

- In this thesis we have only discussed the task scheduling issues. In fact, the task level data access and memory management for highly dynamic applications is also an interesting topic, especially when it is integrated with our TCM approach [106, 3].

Appendix A

List of Publications

Journal Papers

1. P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. "Energy-aware Runtime Scheduling for Embedded Multiprocessor SoCs", *IEEE Design & Test of Computers*, 18(5):46–58, Sept. 2001.

2. P. Marchal, M. Jayapala, S. De Souza, P. Yang, F. Catthoor, and G. Deconinck. "Matador: an exploration environment for system-design", *Journal of Circuits, Systems and Computers*, 11(5):503–535, 2002.

Book Chapter

1. P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. "Cost-efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems", in *Multi-Processor System on Chip*, Morgan Kaufman, 2004.

Conference Papers

1. P. Yang, D. Desmet, F. Catthoor, and D. Verkest. "Dynamic Scheduling of Concurrent Tasks with Cost Performance Trade-off", in *Int. Conf. Compilers, Architectures, and Synthesis for Embedded Systems*, pages 103–109, San Jose, CA, 2000.

2. D. Verkest, P. Yang, C. Wong, and P. Marchal. "Optimisation Problems for Dynamic Concurrent Task-based Systems", in *IEEE/ACM International Conference on Computer-Aided Design*, embedded tutorial, pages 265–268, San Jose, Nov. 2001.

3. C. Wong, P. Marchal, P. Yang, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, and H. De Man. "Task Concurrency Management Methodology to Schedule the MPEG4 IM1 Player on a Highly Parallel Processor Platform", in *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 170–175, 2001.

4. P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. "Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems", in *Proceedings of International Symposium on System Synthesis*, invited paper, pages 112–119, Kyoto, Japan, Oct. 2002.

5. P. Yang and F. Catthoor. "Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems", in *ISSS+CODES*, pages 120–125, Newport Beach, CA, 2003.

6. P. Yang and F. Catthoor. "Dynamic Mapping and Ordering Tasks of Embedded Real-Time Systems on Multiprocessor Platforms", in *Proc. 8th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Amsterdam, the Netherland, 2004.

Appendix B

Abbreviations

API	Application Program Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction Processor
CDFG	Control Data Flow Graph
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DP	Dynamic Programming
DPM	Dynamic Power Management
DSP	Digital Signal Processing
DTSE	Data Transfer and Storage Exploration
DVS	Dynamic Voltage Scaling
EDF	Earliest Deadline First
IC	Integrated Circuit
ILP	Integer Linear Programming
MCKP	Multiple Choice Knapsack Problem
MILP	Mixed Integer Linear Programming
MIPS	Million Intruction Per Second
MPEG	Moving Picture Expert Group
MTG	Multi Task Graph
PBD	Platform Based Design
QoS	Quality of Service
RM	Rate Monotonic
RT	Real Time
RTOS	Real Time Operating System
SoC	System on Chip
TCM	Task Concurrency Management
TF	Thread Frame

TN	Thread Node
VLIW	Very Long Instruction Word
WCET	Worst Case Execution Time

Bibliography

[1] A. Acquaviva, L. Benini, and B. Ricco. Software-Controlled Processor Speed Setting for Low-Power Streaming Multimedia. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 20(11):1283–1292, Nov. 2001.

[2] Acunia. http://www.acunia.com.

[3] D. Atienza, S. Mamagkakis, F. Catthoor, J. Mendias, and D. Soudris. Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In *Proceedings of the Design Automation and Test in Europe*, pages 532–537, Paris, France, Feb. 2004.

[4] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings. Fixed Priority Preemptive Scheduling: an Historical Perspective. *Real-Time Systems*, 8(2):173–198, 1995.

[5] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proceedings of the 5th Euromicro Workshop on Real-Time System*, pages 36–41. Oulu, Finland, 1993.

[6] N. C. Audsley. Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Technical Report YCS 164, Deptartment of Computer Science, University of York, UK, 1991.

[7] A. Azevedo et al. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the Design Automation and Test in Europe*, pages 168–175, 2002.

[8] T. P. Baker. Stack-Based Scheduling of Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[9] F. Balarin et al. *Hardware-Software Co-Design of Embedded Systems: the POLIS Approach*. Kluwer Academic Publishers, 1997.

[10] S. Baruah et al. On the Competitiveness of On-line Real-time Scheduling. In *Proceedings of the IEEE Real-Time System Symposium*, pages 106–115, Dec. 1991.

[11] L. Benini, A. Bogliolo, and G. De Micheli. A Survey of Design Techniques for system-Level Dynamic Power Management. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 8(3):299–316, June 2000.

[12] L. Benini and G. De Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Boston, 1997.

[13] L. Benini and G. De Micheli. System-level Power Optimization Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, Apr. 2000.

[14] L. Benini and G. De Micheli. Powering Networks on Chips. In *Proceedings of International Symposium on System Synthesis*, pages 33–38, Montreal, Quebec, Canada, Oct.1-3 2001.

[15] T. Benner and R. Ernst. An Approach to Mixed Systems Co-Synthesis. In *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 9–14, 1997.

[16] B. A. Blake and K. Schwan. Experimental Evaluation of a Real-time Scheduler for a Multiprocessor System. *IEEE Transactions on Software Engineering*, 17(1):34–44, Jan. 1991.

[17] J. Borel. Design automation in MEDEA: Present and Future. *IEEE Micro*, 19(5):71–79, Sept. 1999.

[18] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. An Instruction-level Functionality-based Energy Estimation Model for 32-bits Microprocessors. In *Proceedings of the 37th Design Automation Conference*, pages 346–350, June 2000.

[19] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A Dynamic Voltage Scaled Microprocessor System. *IEEE J. Solid-State Circuits*, 35(11):1571–1580, Nov. 2000.

[20] G. C. Buttazzo and F. Sensini. Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE Transactions on Computers*, 48(10):1035–1051, Oct. 1999.

[21] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data Driven Signal Processing: An Approach for Energy Efficient Computing. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 347–352, 1996.

[22] A. P. Chandrakasan and R. W. Brodersen. Minimizing Power Consumption in Digital CMOS Circuits. *Proceedings of the IEEE*, 83(4):498–523, Apr. 1995.

[23] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, Apr. 1992.

[24] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC Revolution: A Guide to Platform-Based Design*. Kluwer Academic Publishers, 1999.

[25] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, Oct. 1989.

[26] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic Scheduling of Real-Time Tasks Under Precedence Constraints. *Real-Time Systems*, 2(3):181–194, Sept. 1990.

[27] E.-Y. Chung, L. Benini, A. Bogliolo, and G. De Micheli. Dynamic Power Management for Non-Stationary Service Requests. In *Proceedings of the Design Automation and Test in Europe*, pages 77–81, 1999.

[28] E.-Y. Chung, L. Benini, and G. De Micheli. Dynamic Power Management Using Adaptive Learning Tree. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 274–279, 1999.

[29] E.-Y. Chung, L. Benini, and G. De Micheli. Contents Provider-Assisted Dynamic Voltage Scaling for Low Energy Multimedia Applications. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 42–47, Monterey, CA, USA, Aug. 2002.

[30] T. Claasen. High Speed: Not the Only Way to Exploit the Intrinsic Computational Power of Silicon. In *Proc. Int. Solid-State Circuits Conf.*, pages 22–25, San Fransisco, CA, Feb. 1999.

[31] A. P. Dancy, R. Amirtharajah, and A. P. Chandrakasan. High-Efficiency Multiple-Output DC-DC Conversion for Low-Voltage Systems. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 8(3):252–263, June 2000.

[32] B. P. Dave and N. K. Jha. COHRA: Hardware-Software Co-Synthesis of Hierarchical Distributed Embedded System Architectures. In *Proceedings of the International Conference on VLSI Design*, pages 347–354, 1997.

[33] B. P. Dave and N. K. Jha. COHRA: Hardware-Software Co-Synthesis of Hierarchical Heterogeneous Distributed Embedded Systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 17(10):900–919, Oct. 1998.

[34] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-Software Co-Synthesis of heterogeneous Distributed Embedded Systems. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 7(1):92–104, Mar. 1999.

[35] R. I. Davis. Scheduling Slack Time in Fixed Priority Preemptive Systems. Technical Report YCS 216, Dept. of Comp. Sci., Univ. of York, UK, 1993.

[36] E. A. de Kock. Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study. In *Proceedings of International Symposium on System Synthesis*, pages 68–73, Kyoto, Japan, Oct. 2002.

[37] H. De Man. On Nanoscale Integration and Gigascale Complexity in the Post .com World. In *Proceedings of the Design Automation and Test in Europe*, 2002. keynote speech.

[38] M. L. Dertouzos and A. K. lau Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, Dec. 1989.

[39] R. P. Dick and N. K. Jha. CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 62–68, 1998.

[40] R. P. Dick and N. K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 17(10):920–935, Oct. 1998.

[41] R. P. Dick and N. K. Jha. MOCSYN: Multiobjective Core-Based Single-Chip System Synthesis. In *Proceedings of the Design Automation and Test in Europe*, pages 263–270, Mar. 1999.

[42] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: Task Graphs for Free. In *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 97–101, 1998.

[43] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run-time. In *Proc. of Programming Language Design and Implementation*, pages 229–241, 1999.

[44] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems. In *Proceedings of the Design Automation and Test in Europe*, pages 132–138, 1998.

[45] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano. Power Estimation of Embedded Systems: A Hardware/Software Codesign Approach. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 6(2):267–275, June 1998.

[46] P. Gerin, S. Yoo, G. Nicolescu, and A. Jerraya. Scalable and Flexible Cosimulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures. In *Proc. Asia South Pacific Design Automation Conference*, pages 63–68, 2001.

[47] S. Gertphol et al. A Metric and Mixed-Integer-Programming-Based Approach for Resource Allocation in Dynamic Real-Time Systems. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2002.

[48] T. Givargis, J. Henkel, and F. Vahid. Interface and Cache Power Exploration for Core-Based Embedded System Design. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 270–275, Oct. 1999.

[49] J. Goodman, A. Chandrakasan, and A. P. Dancy. Design and Implementation of a Scalable Encryption Processor with Embedded Variable DC/DC Converter. In *Proceedings of the 36th Design Automation Conference*, pages 855–860, 1999.

[50] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proc. the First Annual Int. Conf. on Mobile Computing and Networking*, pages 13–25, New York, NY, USA, 1995.

[51] F. Gruian and K. Kuchcinski. Uncertainty-Based Scheduling: Energy-Efficient Ordering for Tasks with Variable Execution Time. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 465–468, Aug.25-27 2003.

[52] D. Grunwald, C. B. Morrey III, et al. Policies for Dynamic Clock Scheduling. In *Proc. Symp. Operating Systems Design and Implementation*, pages 13–25, Oct. 2000.

[53] S. Ha and E. A. Lee. Compile-Time Scheduling of Dynamic Constructs in Dataflow Program Graphs. *IEEE Transactions on Computers*, 46(7):768–778, July 1997.

[54] S. Hartmann. Project scheduling with multiple modes: A genetic algorithm. *Annals of Operations Research*, 102:111–135, 2001.

[55] T. Hegazy and B. Ravindran. Using Application Benefit for Proactive Resource Allocation in Asynchronous Real-Time Distributed Systems. *IEEE Transactions on Computers*, 51(8):945–962, Aug. 2002.

[56] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 217–222, Seoul, Korea, Aug.25-27 2003.

[57] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power Optimization of Variable Voltage Core-Based Systems. In *Proceedings of the 35th Design Automation Conference*, pages 176–181, San Francisco, CA, 1998.

[58] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power Optimization of Variable Voltage Core-Based Systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(12):1702–1714, Dec. 1999.

[59] I. Hong, M. Potkonjak, and M. B. Srivastava. On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 653–656, San Jose, CA, 1998.

[60] I. Hong, G. Qu, M. Potkonjak, and M. B. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. In *Proceedings of the IEEE Real-Time System Symposium*, pages 178–187, 1998.

[61] C.-J. Hou and K. G. Shin. Load Sharing with Considerations of Future Arrivals in Heterogeneous Distributed Real-time Systems. In *Proceedings of the IEEE Real-Time System Symposium*, pages 94–103, Dec. 1991.

[62] S. Hua, G. Qu, and S. S. Bhattacharyya. Energy Reduction Techniques for Multimedia Applications with Tolerance to Deadline Misses. In *Proceedings of the 40th Design Automation Conference*, pages 131–136, Anaheim, CA, USA, June 2003.

[63] C. Hughes, V. Pai, P. Ranganathan, and S. Adve. Rsim: Simulating Shared-Memory Multiorcessors with ILP Processors. *IEEE Computer*, 35(2):40–49, Feb. 2002.

[64] H. Hulgaard and S. M. Burns. Bounded Delay Timing Analysis of a Class of CSP Programs. *Formal Methods in System Design*, 11:265–294, 1997.

[65] C. Im, H. Kim, and S. Ha. Dynamic Voltage Scheduling Technique for Low-Power Multimedia Applications Using Buffers. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 34–39, 2001.

[66] T. Ishihara and H. Yasuura. Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 197–202, 1998.

[67] A. Iyer and D. Marculescu. Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors. In *International Symposium on Computer Architecture*, pages 158–168, 2002.

[68] N. K. Jha. Low Power System Scheduling and Synthesis. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 259–263, 2001.

[69] C. Joshi, A. Kumar, and M. Balakrishnan. A New Performance Evaluation Approach for System Level Design Space Exploration. In *Proceedings of International Symposium on System Synthesis*, pages 180–185, Kyoto, Japan, Oct.2-4 2002.

[70] I. Kadayif, M. Kandemir, and M. Karakoy. An Energy Saving Strategy Based on Adaptive Loop Parallelization. In *Proceedings of the 39th Design Automation Conference*, pages 195–200, New Orleans(LA), USA, June10-14 2002.

[71] I. Kadayif, M. Kandemir, and M. U. Sezer. An Integer Linear Programming Based Approach for Parallelizing Applications in On-Chip Multiprocessors. In *Proceedings of the 39th Design Automation Conference*, pages 703–708, New Orleans(LA), USA, June10-14 2002.

[72] W. Kim, J. Kim, and S. L. Min. A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis. In *Proceedings of the Design Automation and Test in Europe*, pages 788–794, 2002.

[73] R. Kolisch and S. Hartmann. *Project scheduling: Recent models, algorithms and applications*, chapter Heuristic Algorithms for Solving the Resource-Constrainted Project Scheduling Problem: Classification and Compuptational Analysis, pages 147–178. Kluwer Academic Publishers, Amsterdam, the Netherlands, 1999.

[74] R. Kolisch and R. Padman. An integrated perspective of project scheduling. Technical Report 463, Manuskripte aus den Instituten fur Betriebswirtschaftslehre der Universitat Kiel, 1997.

[75] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[76] W.-C. Kwon and T. Kim. Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors. In *Proceedings of the 40th Design Automation Conference*, pages 125–130, Aneheim (CA), USA, June2-6 2003.

[77] M. Lajolo, A. Raghunathan, S. Dey, and L. Lavagno. Efficient Power Coestimation Techniques for System-on-Chip Design. In *Proceedings of the Design Automation and Test in Europe*, pages 27–34, Mar. 2000.

[78] C. Lee, M. Potkonjak, and W. Wolf. Synthesis of Hard Real-Time Application Specific Systems. *Design Automation for Embedded System*, 4(4):216–242, Oct. 1999.

[79] C.-G. Lee, J. Hahn, et al. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[80] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[81] S. Lee and T. Sakurai. Run-Time Voltage Hopping for Low-Power Real-Time Systems. In *Proceedings of the 38th Design Automation Conference*, pages 806–809, Los Angeles, CA, 2000.

[82] 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 *Proceedings of International Symposium on Low Power Electronic Device*, pages 84–87, Monterey, CA, USA, Aug. 2002.

[83] T.-M. Lee, J. Henkel, and W. Wolf. Dynamic Runtime Re-Scheduling Allowing Multiple Implementations of a Task for Platform-based Design. In *Proceedings of the Design Automation and Test in Europe*, pages 296–301, 2002.

[84] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proceedings of the IEEE Real-Time System Symposium*, pages 110–123, 1992.

[85] J. P. Lehoczky, L. Sha, and Y. Ding. The Rate Monotone Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the IEEE Real-Time System Symposium*, pages 166–171, Dec. 1989.

[86] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the IEEE Real-Time System Symposium*, pages 261–270, 1987.

[87] J. A. Leijten, J. L. van Meerbergen, A. H. Timmer, and J. A. Jess. Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor. In *Proceedings of the Design Automation and Test in Europe*, pages 125–131, 1998.

[88] J.-T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[89] T. Li and L. K. John. Routine Based OS-aware Microprocessor Resource Adaptation for Run-time Operating System Power Saving. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 241–246, Seoul, Korea, Aug.25-27 2003.

[90] Y. Li and J. Henkel. A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems. In *Proceedings of the 35 Design Automation Conference*, pages 188–193, June 1998.

[91] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[92] J. Liu, P. H. Chou, and N. Bagherzadeh. Combined Functional Partitioning and Communication Speed Selection for Networked Voltage-Scalable Processors. In *Proceedings of International Symposium on System Synthesis*, pages 14–19, Kyoto, Japan, Oct. 2002.

[93] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, May 1985.

[94] Y.-H. Lu, L. Benini, and G. De Micheli. Low-Power Task Scheduling for Multiple Devices. In *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 39–43, 2000.

[95] Y.-H. Lu, L. Benini, and G. De Micheli. Operating-System Directed Power Reduction. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 37–42, 2000.

[96] Y.-H. Lu, L. Benini, and G. De Micheli. Power-aware Operating System for Interactive System. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 10(2):119–34, Apr. 2002.

[97] Y.-H. Lu, T. Simunic, and G. De Micheli. Software Controlled Power Management. In *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 157–161, 1999.

[98] J. Luo and N. Jha. Battery-Aware Static Scheduling for Distributed Real-Time Embedded Systems. In *Proceedings of the 38th Design Automation Conference*, pages 444–449, June18-22 2001.

[99] J. Luo and N. Jha. Static and Dynamic Variable Voltage Scheduling Algorithms for Real-Time Heterogeneous Distributed Embedded Systems. In *7th ASPDAC and 15th Int'l Conf. on VLSI Design*, pages 719–726, Jan. 2002.

[100] T. C.-L. Ma and K. G. Shin. A User-Customizable Energy-Adaptive Combined Static/Dynamic Scheduler for Mobile Applications. In *Proceedings of the IEEE Real-Time System Symposium*, pages 227–236, 2000.

[101] Z. Ma, C. Wong, E. Delfosse, J. Vounckx, F. Catthoor, S. Himpe, and G. Deconinck. Task Concurrency Analysis and Exploration of Visual Texture Decoder on a Heterogeneous Platform. In *IEEE Workshop on Signal Processing Systems (SIPS)*, pages 245–250, Soul, South Korea, 2003.

[102] P. Magarshack and P. G. Paulin. System-on-chip beyond the nanometer wall. In *Proceedings of the Design Automation and Test in Europe*, pages 419–424, 2003.

[103] S. Malik and M. Martonosi. Static Timing Analysis of Embedded Software. In *Proceedings of the 34th Design Automation Conference*, pages 147–152, 1997.

[104] A. Manzak and C. Chakrabarti. Voltage Scaling for Energy Minimization with QoS Constraints. In *ICDD*, pages 438–443, 2001.

[105] P. Marchal, J. Gomez, L. Pinuel, D. Bruni, L. Benini, F. Catthoor, and H. Corporaal. SDRAM-energy-aware Data Allocation for Dynamic Multi-media Applications on Multiprocessor Platforms. In *Proceedings of the Design Automation and Test in Europe*, pages 516–521, Munich, Germany, Mar. 2002.

[106] P. Marchal, M. Jayapala, S. D. Souza, P. Yang, F. Catthoor, and G. Deconinck. Matador: an exploration environment for system-design. *Journal of Circuits, Systems and Computers*, 11(5):503–535, 2002.

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

[108] G. Martin and H. Chang, editors. *Winning the SoC Revolution: Experiences in Real Design*. Kluwer Academic Publishers, 2003.

[109] MediaBench. http://cares.icsl.ucla.edu/MediaBench.

[110] P. Mejia-Alvarez, E. Levner, and D. Mosse. Power-Optimized Scheduling Server for Real-Time Tasks. In *Proceedings of the 8th IEEE Real-Time and Embedded technology and Applications Symposium*, pages 239–250, 2002.

[111] B. Mochocki, X. S. Hu, and G. Quan. A Realistic Variable Voltage Scheduling Model for Real-Time Applications. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 726–731, 2002.

[112] A. K. Mok and M. L. Dertouzos. Multiprocessor Scheduling in a Hard Real-time Environment. In *Proc. 7th Texas Computing Conf. Computing Systems*, 1978.

[113] T. Mudge. Power: A first class design constraint for future architectures. In *HiPC 2000*, pages 215–224, Bangalore, India, Dec. 2000.

[114] T. Okuma, T. Ishihara, and H. Yasuura. Real-Time Task Scheduling for a Variable Voltage Processor. In *Proceedings of International Symposium on System Synthesis*, pages 24–29, 1999.

[115] T. Okuma, H. Yasuura, and T. Ishihara. Software Energy Reduction Techniques for Variable-Voltage Processors. *IEEE Design & Test of Computers*, 18(2):31–41, March-April 2001.

[116] V. Pareto. *Manuale di Economia Politica*. Piccola Biblioteca Scientifica, Milan, 1906. Translated into English by Ann S. Schwier (1971), Manual of Political Economy, MacMillan, London.

[117] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 76–81, 1998.

[118] T. Pering, T. Burd, and R. Brodersen. Voltage Scheduling in the IpARM Microprocessor System. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 96–101, Rapallo, Italy, 2000.

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

[120] PocketGL. http://www.pocketpcdn.com/libraries/pocketgl.html.

[121] P. Pop, P. Eles, and Z. Peng. Bus Access Optimization for Distributed Embedded Systems Based on Schedulability Analysis. In *Proceedings of the Design Automation and Test in Europe*, pages 567–574, 2000.

[122] T. Pop, P. Eles, and Z. Peng. Design Optimization of Mixed Timer/Event-Triggered Distributed Embedded Systems. In *First IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign & System Synthesis (CODES+ISSS)*, pages 31–36, Oct.1-3 2003.

[123] T. Pop, P. Eles, and Z. Peng. Schedulability Analysis for Distributed Heterogeous Time/Event Triggered Real-Time Systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 257–266, 2003.

[124] J. Pouwelse, K. Langendoen, and H. Sips. Energy Priority Scheduling for Variable Voltage Processors. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 28–33, 2001.

[125] A. Prayati, C. Wong, P. Marchal, et al. Task Concurrency Management Experiment for Power-efficient Speed-up of Embedded MPEG4 IM1 Player. In *International Conference on Parallel Processing*, pages 453–460, 2000.

[126] G. Qu and M. Potkonjak. System Synthesis of Synchronous Multimedia Applications. In *Proceedings of International Symposium on System Synthesis*, pages 128–133, San Jose, CA, USA, Dec. 1999.

[127] G. Qu and M. Potkonjak. Achieving utility arbitrarily close to the optimal with limited energy. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 125–130, Rapallo, Italy, Aug. 2000.

[128] G. Qu and M. Potkonjak. Energy Minimization with Guaranteed Quality of Service. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 43–48, Rapallo, Italy, Aug. 2000.

[129] G. Quan and X. Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. In *Proceedings of the 38th Design Automation Conference*, pages 828–833, 2001.

[130] G. Quan and X. S. Hu. Minimum Energy Fixed-Priority Scheduling for Variable Voltage Processors. In *Proceedings of the Design Automation and Test in Europe*, pages 782–787, 2002.

[131] J. Rabaey. Design in the Late-Silicon Age. In *DesignCon*, 2004. keynote speech.

[132] D. Rakhmatov, S. Vrudhula, and C. Chakrabarti. Battery-conscious task sequencing for portable devices including voltage/clock scaling. In *Proceedings of the 39th Design Automation Conference*, pages 189–194, New Orleans(LA), USA, June10-14 2002.

[133] K. Ramamritham. Allocation and Scheduling of Complex Periodic Tasks. In *10th Int. Conf. on Distributed Computing Systems*, pages 108–115, Paris, France, June 1990.

[134] K. Ramamritham, G. Fohler, and J. M. Adan. Issues in the Static Allocation and Scheduling of Complex Periodic Tasks. In *10th IEEE Workshop on Real-Time Operating Systems and Software*, pages 11–16, May 1993.

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

[136] K. Ramamritham, J. A. Stankovic, and P. Shiah. Efficient Scheduling Algorithams for Real-time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, Apr. 1990.

[137] S. Ramos-Thuel and J. P. Lehoczky. On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems. In *Proceedings of the IEEE Real-Time System Symposium*, pages 160–171, 1993.

[138] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model Composition for Scheduling Analysis in Platform Design. In *Proceedings of the 39th Design Automation Conference*, pages 287–292, New Orleans(LA), USA, June10-14 2002.

[139] M. Rosenblum, E. Bugnion, S. Devine, and S. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[140] A. Sangiovanni-Vincentelli and G. Martin. Platform-Based Design and Software Design Methodology for Embedded Systems. *IEEE Design & Test of Computers*, 18(6):23–33, November-December 2001.

[141] M. T. Schmitz and B. M. Al-Hashimi. Considering Power Variations of DVS Processing Elements for Energy Minimisation in Distributed Systems. In *Proceedings of International Symposium on System Synthesis*, pages 250–255, 2001.

[142] K. Schwan and H. Zhou. Dynamic Scheduling of Hard Real-Time Tasks and Real-Time Threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, Aug. 1992.

[143] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In *Proc. International Symposium on Microarchitecture*, pages 356–367, Feb. 2002.

[144] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Int. Symp. High-Performance Computer Architecture*, pages 29–40, 2002.

[145] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.

[146] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, Jan. 1994.

[147] L. Shang and N. K. Jha. Hardware-Software Co-Synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs. In *7th ASPDAC and 15th Int'l Conf. on VLSI Design*, pages 345–352, Jan. 2002.

[148] C. Shen, K. Ramamritham, and J. A. Stankovic. Resource Reclaiming in Multiprocessor Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):382–397, Apr. 1993.

[149] S.Himpe. MTG* and Gray-box: A Modern Bible for TCM Methodologies. Technical report, Katholieke Universiteit Leuven, July 2003.

[150] S.Himpe, G.Deconinck, F.Catthoor, and J.Meerbergen. MTG* and Grey-Box: Modeling Dynamic Multimedia Applications with Concurrency and Nondeterminism. In *Proc. Forum on Design Languages(FDL)*, Marseille, France, Sept. 2002.

[151] D. Shin, J. Kim, and S. Lee. Low-Energy Intra-Task Voltage Scheduling Using Static Timing Analysis. In *Proceedings of the 38th Design Automation Conference*, pages 438–443, 2001.

[152] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proceedings of the 36th Design Automation Conference*, pages 134–139, 1999.

[153] Y. Shin, K. Choi, and T. Sakurai. Power Optimization of Real-Time Embedded Systems on Variable Speed Processors. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 365–368, 2000.

[154] T. Simunic. *Energy Efficient System Design and Utilization*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2001.

[155] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic Voltage Scaling and Power Management for Portable Systems. In *Proceedings of the 38th Design Automation Conference*, pages 524–529, 2001.

[156] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Event-Driven Power Management. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 20(7):840–857, July 2001.

[157] T. Simunic, L. Benini, and G. D. Micheli. Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. In *Proceedings of the 36th Design Automation Conference*, pages 867–872, June 1999.

[158] T. Simunic, H. Vikalo, P. Glynn, and G. De Micheli. Energy Efficient Design of Portable Wireless Systems. In *Proceedings of International Symposium on Low Power Electronic Device*, pages 49–54, 2000.

[159] A. Sinha and A. P. Chandrakasan. Energy Efficient Real-Time Scheduling. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 458–463, 2001.

[160] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. *Real-Time Systems*, 1:27–60, 1989.

[161] M. Spuri and G. C. Buttazzo. Efficient Aperiodicc Service under Earliest Deadline Scheduling. In *Proceedings of the IEEE Real-Time System Symposium*, pages 2–11, 1994.

[162] M. Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10(2):179–210, 1996.

[163] W. Stalling. *Operating Systems: Internals and Design Principles*. Prentice Hall, 1998.

[164] I. Stoica et al. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the IEEE Real-Time System Symposium*, pages 288–299, 1996.

[165] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall Inc., 1995.

[166] F. Thoen and F. Catthoor. *Modeling, Verification and Exploration of Task-level Concurrency in Real-Time Embedded Systems*. Kluwer Academic Publishers, 1999.

[167] V. Tiwari, S. Malik, A. Wolfe, and M. Lee. Instruction-Level Power Analysis. *Journal of VLSI Signal Processing*, (1):223–238, 1996.

[168] B. Vassileios. Evaluation of Real Time Operating Systems. Technical report, ADT, IMEC, 2001.

[169] D. Verkest, P. Yang, C. Wong, and P. Marchal. Optimisation problems for dynamic concurrent task-based systems. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 265–268, San Jose, Nov. 2001.

[170] B. Walsh, R. van Engelen, K. Gallivan, J. Birch, and Y. Shou. Parametric Intra-Task Dynamic Voltage Scheduling. In *Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, 2003.

[171] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proc. Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[172] Windver. VSPWorks. www.windriver.com/products/vspworks/index.html.

[173] F. Wolf and R. Ernst. Intervals in Software Execution Cost Analysis. In *Proceedings of International Symposium on System Synthesis*, pages 130–135, 2000.

[174] C. Wong. *Design-Time Sub-Task Scheduling for Embedded Multimedia and Telecom Systems*. PhD thesis, Katholieke Universiteit Leuven, Dep. Elec. Eng., Sept. 2003.

[175] C. Wong, P. Marchal, P. Yang, A. Prayati, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, and H. De Man. Task Concurrency Management Methodology to Schedule the MPEG4 IM1 Player on a Highly Parallel Processor Platform. In *Proceedings of the International Workshop on Hardware/Software Codesign(CODES)*, pages 170–175, 2001.

[176] XScale 80200. http://www.intel.com/design/pca/applicationsprocessors.

[177] J. Xu and D. L. Parnas. Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3):360–369, Mar. 1990.

[178] P. Yang and F. Catthoor. Pareto-Optimization-Based Run-Time Task Scheduling for Embedded Systems. In *ISSS+CODES*, pages 120–125, Newport Beach, CA, 2003.

[179] P. Yang and F. Catthoor. Dynamic Mapping and Ordering Tasks of Embedded Real-Time Systems on Multiprocessor Platforms. In *Proc. 8th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Amsterdam, the Netherland, 2004.

[180] P. Yang, D. Desmet, F. Catthoor, and D. Verkest. Dynamic Scheduling of Concurrent Tasks with Cost Performance Trade-off. In *Int. Conf. Compilers, Architectures, and Synthesis for Embedded Systems*, pages 103–109, San Jose, CA, 2000.

[181] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. In *Proceedings of International Symposium on System Synthesis*, pages 112–119, Kyoto, Japan, Oct. 2002.

[182] P. Yang, P. Marchal, C. Wong, S. Himpe, F. Catthoor, P. David, J. Vounckx, and R. Lauwereins. *Multi-Processor System on Chip*, chapter Cost-efficient Mapping of Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. Morgan Kaufman, 2004.

[183] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware Runtime Scheduling for Embedded Multiprocessor SoCs. *IEEE Design & Test of Computers*, 18(5):46–58, Sept. 2001.

[184] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[185] T. T. Ye, L. Benini, and G. De Micheli. Analysis of Power Consumption on Switch Fabrics in Network Routers. In *Proceedings of the 39th Design Automation Conference*, pages 524–529, 2002.

[186] Y. Zhang, X. S. Hu, and D. Z. Chen. Task Scheduling and Voltage Selection for Energy Minimization. In *Proceedings of the 39th Design Automation Conference*, pages 183–188, 2002.

[187] D. Ziegenbein, J. Uerpmann, and R. Ernst. Dynamic Response Time Optimization for SDF Graphs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 135–40, 2000.