Master Thesis

A Nonlinear Programming Approach for Dynamic Voltage Scaling

by

Shanai Ardi

LITH-IDA/DS-EX--05/003--SE

Februari 2005

Linköping University
Department of Computer and Information Science

Master Thesis

A Nonlinear Programming Approach for Dynamic Voltage Scaling

by

Shanai Ardi

LITH-IDA/DS-EX--05/003--SE

Februari 2005

Supervisor: Alexandru Andrei
Examiner: Prof. Zebo Peng

Defence date

2005-02-15

Publishing date (Electronic version)

2005-02-23

Department and Division

Institutionen för Datavetenskap
583 83 LINKÖPING

Linköpings universitet

Language

× English
Other (specify below)

_____

Report category

Licentiate thesis
× Degree thesis
Thesis, C-level
Thesis, D-level
Other (specify below)

_____

ISBN:

ISRN: LITH-IDA/DS-EX--05/003--SE

Title of series

Series number/ISSN

URL, Electronic version
http://www.ep.liu.se/exjobb/ida/2005/dt-d/003/

Title
A Nonlinear Programming Approach for Dynamic Voltage Scaling

Author(s)
Shanai Ardi

Abstract

Embedded computing systems in portable devices need to be energy efficient, yet they have to deliver adequate performance to the often computationally expensive applications. Dynamic voltage scaling is a technique that offers a speed versus power trade-off, allowing the application to achieve considerable energy savings and, at the same time, to meet the imposed time constraints.

In this thesis, we explore the possibility of using optimal voltage scaling algorithms based on nonlinear programming at the system level, for a complex multiprocessor scheduling problem. We present an optimization approach to the modeled nonlinear programming formulation of the continuous voltage selection problem excluding the consideration of transition overheads. Our approach achieves the same optimal results as the previous work using the same model, but due to its speed, can be efficiently used for design space exploration. We validate our results using numerous automatically generated benchmarks.

Keywords

Low Power Design, Dynamic Voltage Scaling, Nonlinear Programming, AMPL, Application Program Interface.

To
Ali, Effat, Sonai and Behzad

Abstract

Embedded computing systems in portable devices need to be energy efficient, yet they have to deliver adequate performance to the often computationally expensive applications. Dynamic voltage scaling is a technique that offers a speed versus power trade-off, allowing the application to achieve considerable energy savings and, at the same time, to meet the imposed time constraints.

In this thesis, we explore the possibility of using optimal voltage scaling algorithms based on nonlinear programming at the system level, for a complex multiprocessor scheduling problem. We present an optimization approach to the modeled nonlinear programming formulation of the continuous voltage selection problem excluding the consideration of transition overheads. Our approach achieves the same optimal results as the previous work using the same model, but due to its speed, can be efficiently used for design space exploration. We validate our results using numerous automatically generated benchmarks.

Acknowledgement

First and foremost I would like to express my sincere gratitude to my supervisor, Alexandru Andrei, for all his support and encouragement. Without his great and helpful supports this work may not be led in an efficient way. I would also like to thank my examiner Prof. Zebo Peng for his great advices.

I would like to thank my friends Soheil Samii and Mikael Asplund for their helps and sharing office with me.

I also wish to express how grateful I am to my friends who supported me consistently with their helps and kindness. Specially: Jalal Maleki, Prof. Mariam Kamkar, and Prof. Nahid Shahmehri.

1. Introduction

1.1 Basic Concepts

In recent years, the performance versus power consumption trade-off during the embedded systems design process, has received much attention. The reason is the growing number of portable systems, such as portable computers and personal communication devices.

Techniques for saving power are applied at different abstraction levels during the system design process, from the circuit level to the system level. Techniques like transistor sizing and clock gating are examples of device and circuit level techniques. In this work, we will concentrate on optimizations that are performed at the system level.

System-level energy efficient design performs power optimizations at the architecture, operating system, compiler and application layers. Examples of such techniques are architecture selection, memory and cache optimization, mapping, scheduling and voltage scaling [4].

Dynamic voltage scaling (*DVS*) and adaptive body biasing (*ABB*) are two system-level techniques, which allow an energy/performance trade-off during run-time of the applications. These techniques address the dynamic power and leakage power consumption as the two main sources of power dissipation. They are supported by various commercial processors such as Tranemeta Crusoe, Intel XScale and AMD's mobile processors which can operate at several frequencies and supply voltages.

DVS is effective in reducing dynamic power consumption quadratically. Dynamic power has been the primary source of power consumption, but as the technology feature size shrinks, leakage power consumption is becoming an important concern. *ABB* is a technique that adjusts the body bias during system run-time in order to reduce the leakage power [8], [9].

A large number of embedded systems, for example, the mobile phones or various multimedia devices, are running real-time applications. Thus, an important

concern during the design is the correct timing behavior (captured by imposed execution time constraints).
These systems usually consist of an application executing on a multi-processor platform. The application is specified in the form of a task graph. An important aspect is finding a "good" mapping and a schedule for the application. It is important to note that deciding on a particular mapping and scheduling, has a big impact for the energy consumption of the system. These are known NP-complete problems, so exact algorithms are not practical due to their huge running time. Nevertheless, several heuristics have been proposed, like for example the one in [6].

Once such a mapping and schedule have been found, dynamic voltage scaling (*DVS*) and adaptive body biasing (*ABB*) can be applied to further reduce the energy consumption. These techniques exploit the static slack available in the system, by extending the execution time of the power hungry tasks, but making sure that all the time constraints (deadlines) are met.

Voltage selection approaches can be done on-line or off-line. We restrict ourselves to the off-line techniques and present the previous approaches that belong to this category. In off-line techniques the scaled supply voltages are calculated at design time and applied at run-time, according to the precalculated voltage schedule.

1.2. Previous Works

Some previous related works have been done in this area. Ishihara et al. [10] modeled the discrete voltage selection problem using an integer linear programming (*ILP*) formulation. Kwon et al. [11] proposed linear programming (*LP*) solution for the discrete voltage selection problem with uniform and non-uniform switched capacitance. Zhang et al. [12] presented an approach using an *ILP* formulation for the dynamic voltage scaling of heterogeneous multiprocessor systems, considering a continuous supply voltage. These approaches scale the supply voltage only and neglect the leakage power consumption. The effectiveness of combined supply and threshold voltage selection has been analyzed in [5] and [7]. All of these approaches use heuristic methods which cannot guarantee the optimality.

The approach presented by Andrei et al. [3], investigates the voltage selection for a set of tasks, possibly with dependencies, which are scheduled on multi-

processor systems under real-time constraints. Both the continuous and the discrete voltage selection problems are solved, including the consideration of the transition overheads. The authors propose four different voltage selection schemes formulated as convex nonlinear programming (*NLP*) and mixed integer linear programming (*MILP*) problems, without and with the consideration of the transition overheads, in order to solve them optimally.

1.3. Purpose of the Thesis

In this thesis, we focus on the work from [3] and offer an equivalent implementation for *DVS*, which is more time efficient. This aspect is particularly important when dynamic voltage scaling is used as a part of a bigger optimization framework. For example, when using a genetic algorithm for finding a schedule that minimizes the energy consumption in a multiprocessor system, *DVS* is performed in the fitness function evaluation, every time a feasible schedule is found. This calls for a very efficient implementation of the voltage scaling algorithm. Our work is focused on optimizing the modeled nonlinear programming formulation of the continuous voltage selection problem in [3], excluding for the sake of simplicity, the consideration of transition overheads. In the energy model introduced in [3], both dynamic and leakage power are considered. In this thesis we take the dynamic power part and propose a solution without considering the leakage power. Our experiment uses the C-based API of the MOSEK solver [2] and compares the results for the speed of optimization using this API to results from using AMPL [13].

1.4. Outline

The second section presents the architecture and application models used in [3]. An introduction to mathematical programming, AMPL and MOSEK software package are given in Section 3. The Section 4 introduces the problem definition and the algorithm of using the API of MOSEK. The results of comparing optimization speed for the AMPL and the C-based API are given in Section 5. In order to show the efficiency of using the C-based API in iterative applications, a study of using it in genetic algorithm based optimization problem and the results of the study are presented in Section 6 and 7 respectively. Finally Section 8 presents the conclusions and future work.

2. Architectural Model and System Specification

We consider embedded systems as heterogeneous distributed architectures. Such architectures consist of several different processing elements (PEs), such as programmable microprocessors, ASIPs, FPGAs, and ASICs, some of which feature *DVS* and *ABB* capability. These computational components communicate via an infrastructure of communication links (CLs), like buses and point-to-point connections [3].

An example architecture and a task graph that has been mapped onto the architecture are shown in Fig. 2.1. The functionality of data flow intensive applications, such as voice processing and multimedia, can be captured by task graphs $G(T,C)$. Nodes $\tau \in T$ in these directed acyclic graphs represent computational tasks and edges $c \in C$ indicate data dependencies between these tasks (communications) [3].

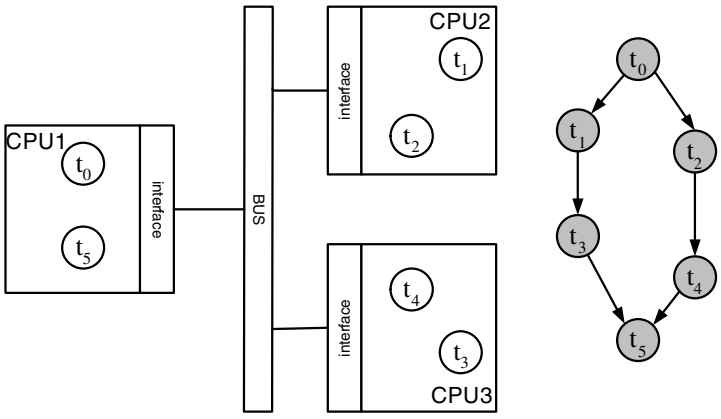


Fig. 2.1. A target architecture with a mapped task graph

A task requires a finite number of clock cycles *NC* to be executed, depending on the PE on which it is mapped. Tasks are annotated with deadlines *dl* which have to be met during application runtime. If two dependent tasks are assigned to different PEs then the communication takes place over a CL, involving a certain amount of communication time and power. According to [3] the task graph is assumed to be mapped and scheduled onto the target architecture, i.e., it is known where and in which order tasks and communication take place. Fig. 2.2 shows a possible execution order of the tasks given in Fig. 2.1.

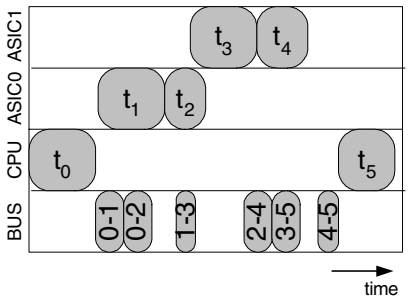


Fig. 2.2. Scheduled tasks and communications

In addition to the precedence relations given by data dependencies between tasks, additional precedence relations $r \in R$ have been introduced in [3]. These dependencies are generated as result of scheduling tasks mapped to the same PE and communications mapped on the same CL.

In Fig. 2.3 the dependencies R are represented as dotted edges. The set of all edges has been defined as $\mathrm{E} = C \cup R$.

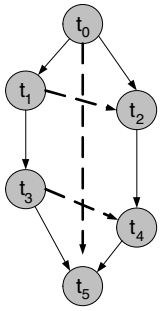


Fig. 2.3. Scheduled task graph (E)

As we mentioned before, dynamic power P_{dyn}, and leakage power P_{leak} are two major sources of power dissipation.

Dynamic power can be represented as:

$$P_{dyn} = C_{eff} \cdot f \cdot V_{dd}^2 \tag{2.1}$$

where C_{eff}, f and V_{dd} denote the effective charged capacitance, operational frequency and circuit supply voltage respectively.

The leakage power can be presented as:

$$P_{leak} = L_g \cdot (V_{dd} \cdot K_3 \cdot e^{K_4 \cdot V_{dd}} \cdot e^{K_5 \cdot V_{dd}} + |V_{bs}| \cdot I_{ju}) \tag{2.2}$$

where V_{bs} is the body bias voltage and I_{ju} represents the body junction leakage current. The fitting parameters K_3, K_4 and K_5 denote circuit technology dependent constants and L_g reflects the number of gates.

The operational frequency can be expressed as:

$$f = \frac{((1 + K_1) \cdot V_{dd} + K_2 \cdot V_{bs} - V_{th1})^{\alpha}}{K_6 \cdot L_d \cdot V_{dd}} \tag{2.3}$$

where α reflects the velocity saturation imposed by the used technology (common values$1.4 \le \alpha \le 2$), L_d is the logic depth, and K_1, K_2, K_6 and V_{th1} are circuit dependent constants [3].

3. Mathematical programming

Mathematical programs are among the most widely used models in operations research and management science. In a mathematical programming problem, one seeks to minimize or maximize a function, subject to constraints on the variables. Mathematical programming refers to the study of these problems; their mathematical properties, the development and implementation of algorithms to solve these problems, and the application of these algorithms to real world problems.

A mathematical program is an optimization problem of the form:

$$\text{Minimize } f(x):\ x \in \mathrm{X},\ g(x) \le 0, h(x) = 0$$

where X is the domain of the real-valued functions, *f*, *g* and *h*. The relations $g(x) \le 0, h(x) = 0$ are called *constraints*, and *f* is called the *objective function*.

A point *x* is feasible if it is in X and satisfies the constraints. A point x^* is optimal if it is feasible and if the value of the objective function is not bigger than that of any other feasible solution: $f(x^*) \le f(x)$ for all feasible *x*.

The sense of the optimization is presented here as minimization, but it could just as well be maximization, with the appropriate change in the meaning of optimal solution: $f(x^*) \ge f(x)$ for all feasible *x*.

Linear programming (*LP*) and integer linear programming (*ILP*) are two well known representatives of mathematical programming. LP assumes that the objective function (*f*) and the constraints (*g and h*) are expressed only as linear functions. The domain of each variable has to be a continuous interval. If all or some of the variables are restricted to the integer domain, the classes of method are called integer linear programming (*ILP*) and mixed integer linear programming (*MILP*), respectively.

The general mathematical programming is nonlinear programming, where the objective function or some of the constraints are nonlinear. A special class of nonlinear programming is nonlinear convex programming which has a convex nonlinear objective function and/or convex nonlinear constraints.

Solving MILP problems was proved to be NP-complete[1]. For LP as well as for convex NLP efficient algorithms with polynomials are available [3].

In order to solve an optimization problem with a solver, and before any optimization routine can be invoked, considerable effort must be expended to formulate the problem and to generate the requisite computational data structures in order to be used in the solver. This can be done using a modeling language or the offered API of the solver (if any). Fig. 3.1 shows the application flow of the use of a solver which offers an API.

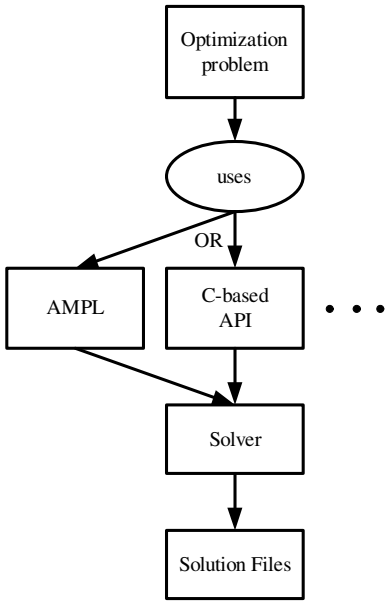


Fig. 3.1. Optimization with a solver

In the following subsections we introduce the AMPL as a modeling language and the MOSEK as a solver.

3.1. AMPL

Practical large-scale mathematical programming involves more than just the application of an algorithm to minimize or maximize an objective function.

[1] For some subclasses, e.g. convex objectives with linear constraints, there exist polynomial algorithms that solve the MILP problems [3].

AMPL is a high-level language designed to resemble the symbolic algebraic notation that many modelers use to describe mathematical programs and it is regular and formal enough to be processed by a computer system [13].

Five major parts of algebraic model—sets, parameters, variables, objectives and constraints—are the five kinds of components in an AMPL model. The definition of these components should be done in a model file (<file>.mod). Once the AMPL translator has read and processed the contents of the model file, it is ready to read the data, which are constant values or inputs, defined in data files (<file>.dat). The primary job of the AMPL translator is to read the model and the data file, and to write a representation of the problem suitable for use by optimization algorithms. The translator must also store enough model information to allow for an understandable listing of the optimal solution. After theses phases and some other intermediate steps, the final output phase makes the translated model available to a solver[14]. The functional diagram of using AMPL is shown in Fig. 3.2.

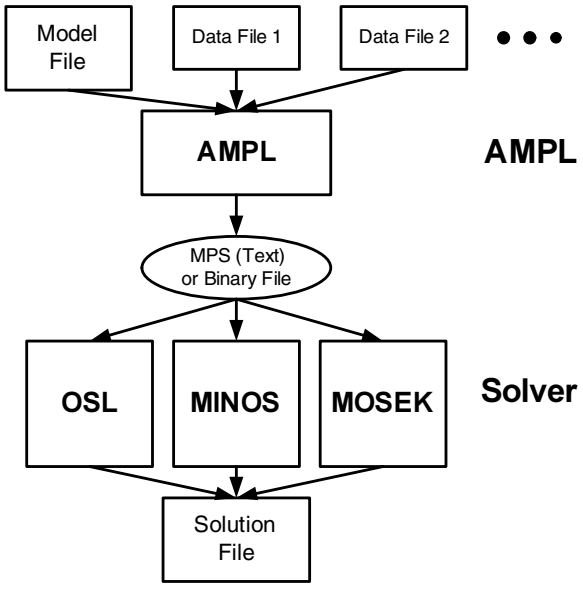


Fig. 3.2. AMPL functional diagram

Most of the solvers allow several ways of specifying the problem including the AMPL language. Some of these solvers accept the specification of the problem in C or other languages using a specific API. AMPL is independent of the solver or

tool used for solving but the specification in C is different for each tool and fairly complicated. Examples for AMPL and C are presented in Appendix C.

3.2. MOSEK

MOSEK is a software package for the solution of linear, mixed-integer linear, and convex nonlinear mathematical optimization problems. MOSEK can solve linear programs, generalized linear programs involving nonlinear conic constraints and convex nonlinear programs [2]. These problem classes can be solved using an appropriate optimizer built into MOSEK. The MOSEK optimization tools make several interfaces available to the user. The default interfaces are:

- MPS file interface. MOSEK reads the industry standard MPS file format for specifying (mixed integer) linear optimization problems.
- API interface. MOSEK contains Application Program Interface which allows the user to interact with MOSEK from other programming languages such as C, FORTRAN and so forth.
- AMPL interface.

As it was mentioned, MOSEK is capable of solving the general nonlinear convex programs and it introduces the general form of a nonlinear optimization problem as:

$$\text{Minimize} \quad f(x) + c^{\mathrm{T}}x \tag{3.1}$$

$$\text{Subject to} \quad g(x) + Ax - x^{c} = 0 \tag{3.2}$$

$$l^{c} \le x^{c} \le u^{c} \tag{3.3}$$

$$l^{x} \le x \le u^{x} \tag{3.4}$$

where

- $x \in R^{n}$ is a vector of decision variables (n is the number of decision variables).

- $x^c \in R^m$ is a vector of constraint variables (m is the number of constraints).
- $c \in R^n$ is the linear part of objective function.
- $A \in R^{m \times n}$ is the constraint matrix.
- $l^c \in R^m$ is the lower limit on the activity for the constraints.
- $u^c \in R^m$ is the upper limit on the activity for the constraints.
- $l^x \in R^n$ is the lower limit on the activity for the variables.
- $u^x \in R^n$ is the upper limit on the activity for the variables.
- $f : R^n \to R$ is a nonlinear function.
- $g : R^n \to R^m$ is a nonlinear function.

This implies that the i^{th} constraint essentially has the form

$$l_i^c \le g_i(x) + \sum_{j=1}^{n} a_{ij} x_j \le u_i^c$$

In general MOSEK can only handle convex optimization problems. This implies that *f(x)* and *g(x)* should be twice differentiable for all *x*. One of the offered solutions by MOSEK to solve the nonlinear convex problems is to change the nonlinear functions into separable functions. MOSEK proposes a simplified method for solving the separable nonlinear convex problems.

In order to be able to use MOSEK to solve a separable convex problem, it must satisfy three important requirements:

1. Separability: This requirement implies that all nonlinear function can be written in the form:

$$f(x) = \sum_{j=1}^{n} f^j(x_j) \quad \text{and} \quad g(x) = \sum_{j=1}^{n} g_i{}^j(x_j)$$

Hence, the nonlinear functions should be written as a sum of functions which only depends on one variable.

2. Differentiability: All functions should be twice differentiable for all x_j satisfying $l_j{}^x \le x_j \le u_j{}^x$ if x_j occurs in at least one nonlinear function.

3. Convexity: The objective function should be a convex function.

The method used by MOSEK to solve these problems has been not mentioned in its documentation but regardless of the method, theorem 3.1 illustrates the first steps in optimization and problem definition in MOSEK.

Theorem 3.1
Let f and g be twice continuously differentiable defined on a neighborhood of a point x_o for which $g(x_o) = 0$ and suppose there exists a number k such that: $\nabla f(x_o) - k \cdot \nabla g(x_o) = 0$ and the matrix $L(x_o) = H(x_o) - G(x_o)$ is positive definite where H is the Hessian for f and G is the Hessian for g. Then x_o is the relative minimum for f subject to $g(x) = 0.$ (∇f means gradient of f).

According to this theorem, MOSEK needs to know the Gradient and Hessian of the nonlinear functions in the objective and in the constraints. The definition of the convexity has been introduced in Appendix A and the details about Gradient and Hessian is presented in Appendix B. More information on MOSEK can be found in [2].

4. Proposed Solution

4.1. Problem Formulation

We consider a set of tasks $\mathrm{T} = \{\tau_i\}$ with precedence constraints which have been mapped and scheduled on a set of variable voltage processors. For each task τ_i, its deadline dl_i, its number of clock cycles to be executed NC_i and switched capacitance C_{eff_i} are given. Each processor can vary its supply voltage V_{dd} within certain continuous ranges. The power dissipation (dynamic) and the cycle time (processor speed) depend on the selected voltage. Tasks are executed cycle by cycle, and each cycle can execute at a different supply voltage. The goal is to find voltage assignment for each task such that the individual task deadlines are met and the total energy consumption is minimal.

According to [3] the continuous voltage scaling problem, excluding the transition overheads, can be modeled as the following nonlinear problem formulation:

Minimize

$$\sum_{k=1}^{|\mathrm{T}|} (NC_k \cdot C_{eff_k} \cdot V_{dd_k}^2 + L_g (K_3 \cdot V_{dd_k} \cdot e^{K_4 \cdot V_{dd_k}} \cdot e^{K_5 \cdot V_{bs_k}} + I_{Ju} \cdot \left| V_{bs_k} \right|) \cdot t_k) \tag{4.1}$$

Subject to $t_k = NC_k \cdot \dfrac{(K_6 \cdot L_d \cdot V_{dd_k})}{((1+K_1) \cdot V_{dd_k} + K_2 \cdot V_{bs_k} - V_{th1})^{\alpha}}$ (4.2)

$$D_k + t_k \le D_l \qquad \forall (k,l) \in \mathrm{E} \tag{4.3}$$

$$D_k + t_k \le dl_k \qquad \forall \tau_k \text{ that have a deadline} \tag{4.4}$$

$$D_k \ge 0 \tag{4.5}$$

$$V_{dd_{\min}} \le V_{dd_k} \le V_{dd_{\max}} \quad \text{and} \quad V_{bs_{\min}} \le V_{bs_k} \le V_{bs_{\max}} \tag{4.6}$$

In Eq. (4.1) both dynamic and leakage powe are involved as power dissipation. In this thesis we focus on dynamic power as power dissipation and we neglect the leakage power in Eq. (4.1). If we ignore the scaling of V_{bs} and focus only on dynamic power and scaling of $V_{dd,}$ the value of V_{bs} is assumed to be zero. The energy function will be as below:

Energy: $$\sum_{k=1}^{|T|}\left(NC_k \cdot C_{eff_k} \cdot V_{dd_k}^2\right) \tag{4.7}$$

$$\text{Subject to } t_k = NC_k \cdot \frac{(K_6 \cdot L_d \cdot V_{dd_k})}{((1+K_1)\cdot V_{dd_k} - V_{th1})^{\alpha}} \tag{4.8}$$

$$D_k + t_k \le D_l \qquad \forall (k,l) \in E \tag{4.9}$$

$$D_k + t_k \le dl_k \qquad \forall \tau_k \text{ that have a deadline} \tag{4.10}$$

$$D_k \ge 0 \tag{4.11}$$

$$NC_k \cdot \lambda_1 \le t_k \le NC_k \cdot \lambda_2 \tag{4.12}$$

The variables that need to be optimized in Eq. (4.7) are the task execution times (t_k), the task start times (D_k) as well as the supply voltage V_{dd_k}. The values of λ_1 and λ_2 are calculated according to the values in Eq. (4.2) and Eq. (4.6) to define the bounds for t_k.

The total energy consumption has to be minimized. The minimization has to comply with following relations and constraints. The task execution time has to be equivalent to the number of clock cycles of the task multiplied by the circuit delay for a particular V_{dd} setting as expressed by Eq. (4.8).

Given the execution time of the tasks, it becomes possible to express the precedence constraints between tasks, i.e. a task τ_l can only start its execution after all its predecessor tasks τ_k have finished their execution (D_k + t_k). Predecessors of task τ are all tasks for which there exists an edge $(k,l) \in E$. Similarly, tasks with deadlines dl_k have to be completed before their deadlines are exceeded Eq. (4.10).

Task start times have to be positive and imposed range for t_k should be respected (Eq. (4.12)). The objective Eq. (4.7) and the task execution time Eq. (4.8) are convex functions. Hence, the problem belongs to the class of general convex nonlinear optimization problems.
As it has been mentioned in Section 3, MOSEK provides a package to solve nonlinear convex problems. The available documentation suggests changing the nonlinear functions, in the objective and in the constraints, to the separable functions and then using the provided package.

For a given specific mapped and scheduled task graph, the energy function should be minimized subject to the constraints. One solution for this problem can be using the AMPL language and solve it with a proper solver. Another alternative is using the offered API by MOSEK. As it has been mentioned before the AMPL language is very close to the mathematical formulation and it is easy to write and model. While using AMPL and its solver, the process of optimization is an outer process, and for each iteration, a new process for the solver has to be created. The basic process should communicate with the result file for each row at the time. This imposes a time penalty. This solution has been used by the authors of [3] and it seems more time consuming than specifying the problem with, for example C code using a specific API.

We use the AMPL implementation of [3] and compare the results to API solution. MOSEK provides a library in order to use in any application coded in C. An optimization using an API can be found as the inner loop in another optimization process and this can save time of executing a new process every time and communicating with it via files. We solve the specified problem using C-based API to compare the efficiency of using an API than AMPL. Fig. 4.1. shows the flowchart of the process in our optimization problem.

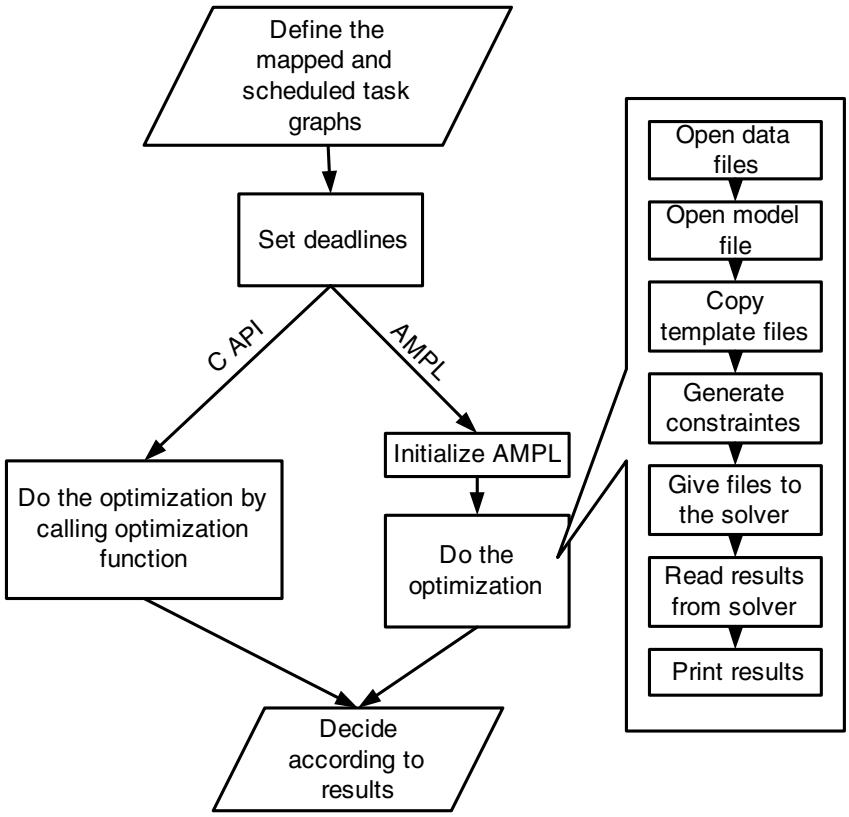


Fig. 4.1. The optimization process

4.2. Energy Function

The energy function and the tasks execution time have been defined by Eq. (4.7) and Eq. (4.8) respectively. According to the used technology, we assume $\alpha = 2$ (α reflects the velocity saturation) in Eq. (4.8). If we extract the V_{dd} value based on its relation to t_k, using Eq. (4.7) and Eq. (4.8), the optimization function will be changed as below:

$$\sum_{k=1}^{|T|}(NC_k \cdot C_{eff_k} \cdot \frac{V_{th1}^2}{(1+K_1)^2} + \frac{2 \cdot NC_k^2 \cdot C_{eff_k} \cdot V_{th1} \cdot K_6 \cdot L_d}{t_k \cdot (1+K_1)^3} + \frac{NC_k^3 \cdot C_{eff_k} \cdot K_6^2 \cdot L_d^2}{2.(1+K_1)^4 \cdot t_k^2}$$

$$+ \frac{1}{t_k} \cdot \sqrt{\frac{NC_k^4 \cdot C_{eff_k}^2 \cdot (V_{th1} \cdot K_6 \cdot L_d)^2}{(1+K_1)^6} + \frac{4 \cdot NC_k^3 \cdot C_{eff_k}^2 \cdot V_{th1}^3 \cdot K_6 \cdot L_d \cdot t_k}{(1+K_1)^5}}$$

$$+ \frac{1}{t_k^2} \cdot \sqrt{\frac{NC_k^6 \cdot C_{eff_k}^2 \cdot K_6^4 \cdot L_d^4}{4 \cdot (1+K_1)^8} + \frac{NC_k^5 \cdot C_{eff_k}^2 \cdot V_{th1} \cdot K_6^3 \cdot L_d^3 \cdot t_k}{(1+K_1)^7}}) \qquad (4.13)$$

Subject to $D_k + t_k \le D_l \quad \forall (k,l) \in E \qquad (4.14)$

$D_k + t_k \le dl_k \quad \forall \tau_k$ that have a deadline $\qquad (4.15)$

$D_k \ge 0 \qquad (4.16)$

$NC_k \cdot \lambda_1 \le t_k \le NC_k \cdot \lambda_2 \qquad (4.17)$

So t_k (task execution time) and D_k (task starting time) are the variables of this function. The objective has t_k as its variable and it can be written as sum of nonlinear functions, which are dependent on only one variable.
This means that the problem can be solved as separable convex optimization.
Eq. (4.13) can be simplified as below:

$$f(t) = \sum_{k=1}^{|T|}(A + \frac{a_1}{b_1 \cdot t_k} + \frac{a_2}{b_2 \cdot t_k^2} + \frac{\sqrt{a_3 + b_3 \cdot t_k}}{t_k} + \frac{\sqrt{a_4 + b_4 \cdot t_k}}{t_k^2}) \qquad (4.18)$$

This function is twice differentiable. It contains the addition of four nonlinear functions each depending on one variable. According to the definition of convexity in Appendix A, this function is convex. What we need is defining the gradient and Hessian of the function and apply the MOSEK rules to it to be able to solve it.

As it has been mentioned in Section 2.1, the functionality of data flow intensive applications can be captured by task graphs $G(T,C)$. The constraints in the

optimization problem should be derived from the graph to define the task dependencies. The main idea behind this constraint production is that each node, representing a task, is annotated with deadline *dl* which has to be met during the application runtime. The dependencies are represented as edges which define the execution order and add more constraints. In our problem we specify these constraints with numbered nodes and edges and introduce them according to MOSEK rules as the coefficiencies of variables into constraint matrix and MOSEK uses it for optimization.

4.3. Numerical Values

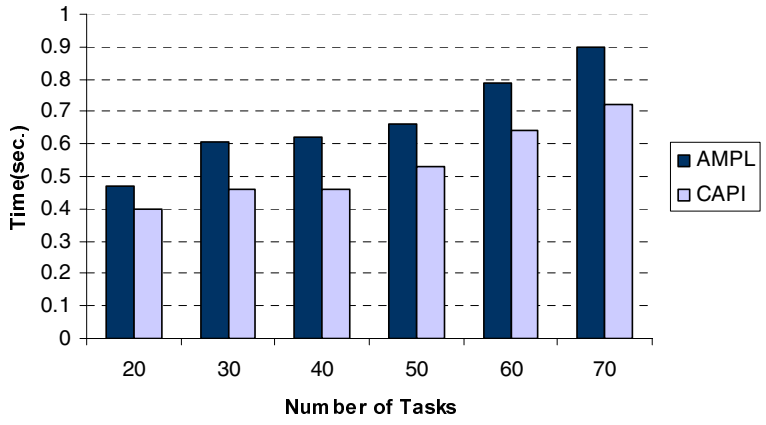
There are constant values used in energy functions that depend on the assumed process. In this thesis, these values corresponding to the 0.18 μm *CMOS* Crusoe processor were calculated using published data on the processor. These parameters (Table 4.1) were adapted from the Berkeley predicted models for 0.18 μm process [7].

VARIABLE	***VALUE***	***VARIABLE***	***VALUE***
K_1	0.053	K_6	51×10^{-12}
K_2	0.140	V_{th1}	0.359
K_3	3.0×10^{-9}	I_{ju}	2.40×10^{-10}
K_4	1.63	L_d	37
K_5	3.65	L_g	4×10^{6}

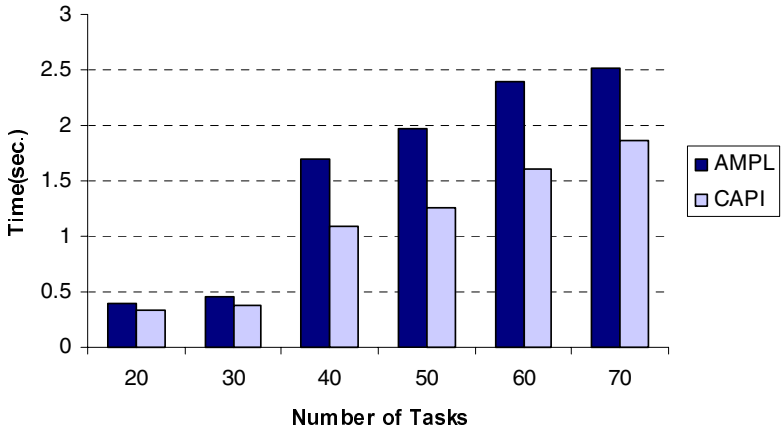
Table 4.1. Constants for the Crusoe 5600 processor in the 0.18 μm process

5. Results

The optimization problem has been implemented using the MOSEK tools. We conduct the experiment for 100 randomly generated task graphs with different number of tasks. The results of the t_k and D_k in optimal case and minimized value of energy in Eq. (4.13) with AMPL and CAPI both using MOSEK as solver, are the same, up to good precision (10^{-10}). The results of the experiment are shown in Fig. 5.1.



(a)



(b)

Fig. 5.1. Results (a) Tasks mapped in 3 processors (b) Task mapped in 4 processors

The results are average values and are proposed for different number of tasks once with scheduling in 3 processors and once in 4 processors. According to these results, optimization using CAPI is on average around 30% faster than optimization using AMPL. This is the average time that it takes to optimize the problem. In our experiment the best case has happened in one of the cases of 50 tasks mapped on 4 processors and our C-based API was 89% faster.

This improvement will be more useful when the optimization is done in iterations and the final result depends on the comparison of several optimization processes.
As an example, in Section 6, we introduce another experiment in Genetic Algorithm to show the efficiency of C-based API in more iteration.
In order to show the efficiency of MOSEK C-based API vs. AMPL we will refer to [15].

6. Genetic Algorithm

Genetic algorithms are widely used for solving practical search exploration and optimization problems. A genetic algorithm (GA) is a technique that mimics biological evolution as a problem-solving strategy. Given a specific problem to solve, the input to the GA is a set of potential solution to that problem, encoded in some fashion, and a metric called fitness function that allows each candidate solutions to be quantitatively evaluated [16].

The GA evaluates each candidate according to the fitness function. Here *fitness* is the suitability of a given member of the candidate population to its environment where in nature the fitness relates to the ability of this member to survive and to reproduce.
In a pool of randomly generated candidates, of course, most will not be feasible at all, and these will be deleted. However purely by chance, a few may hold promise – they may have the chance of being mated. These promising candidates are kept and allowed to reproduce [16]. In the other word highly fit individuals are more likely to be selected than unfit members in reproduction. These winning individuals are selected and copied over into the next generation with random changes, to form a new pool of candidate solutions, and are subjected to second round of fitness evaluation. The expectation is that the average fitness of the population will increase each round [16].

The name *genetic algorithm* originates from the analogy between the representations of a complex structure by means of a vector of components, and the idea, familiar to biologists, of the genetic structure of a chromosome [17]. In nature all living organisms contain a set of genetic data, termed a “genome”. This genetic data encodes all of the physical characteristics of the organism. The string which carries these genomes is called chromosome. Previously mentioned individuals in population are chromosomes. A genetic algorithm works by maintaining a population of chromosomes— potential parents—whose *fitness* value have been calculated. Each chromosome encodes a solution to the problem, and its fitness value is related to the value of the objective functions for the solution.

6.1. Crossover and Mutation

Crossover is an operation in GA in which, two (or more) individuals are involved. In crossover, highly fit individuals are more likely to be selected to mate and produce children than unfit members. In this manner, highly fit vectors are allowed to breed, with the hope that they will produce more fit offspring. Although the crossover operation may take many forms, it typically involves splitting each parent chromosome at a randomly-selected point within the interior of the chromosome, and rearranging the fragments so as to produce offsprings of similar characteristics. The effect of crossover is to build upon the success of the past, and explore new areas of research space [18].

Another operation is mutation which helps to diversify the population. During the mutation only one individual is involved and the idea behind it is to restore genetic diversity lost during the application of reproduction and crossover. After many generation of evolution via the repeated application of reproduction, crossover and mutation, the individuals in the population will often look alike [15]. At this point GA typically terminates because additional evolution will produce little improvement in fitness.

6.2. Problem Formulation

This problem formulation is according to the problem definition in [15] and we use the implementation done in this work as backbone of our experiments. Given a set of tasks T with precedence constraints, captured by an acyclic graph *G(T,C)*; a set of processors PE and a function $\mathrm{M}:\mathrm{T}\rightarrow\mathrm{P}$ for mapping the tasks on the processors. A task $\tau_i \in \mathrm{T}$ is characterized by the number of clock cycles to be executed NC_i, the switched capacitance C_{eff_i} , and a deadline dl_i that has to be met. The main goal is to find a feasible schedule where all the tasks under the given precedence constraints and mapping, meet their deadlines, and the energy consumption of the system is minimized. This problem is NP-complete, so finding the exact solution is not computationally feasible. A genetic algorithm based search is employed to find the feasible schedules with close to minimal energy consumption. The fitness function for each individual is the energy function which will be minimized once with C-based API and once with AMPL.

In optimization problems using GA, calculating the fitness function is done repeatedly for all new individuals and the faster optimization process will decrease the optimization time significantly. The flowchart in Fig. 6.1 illustrates this aspect:

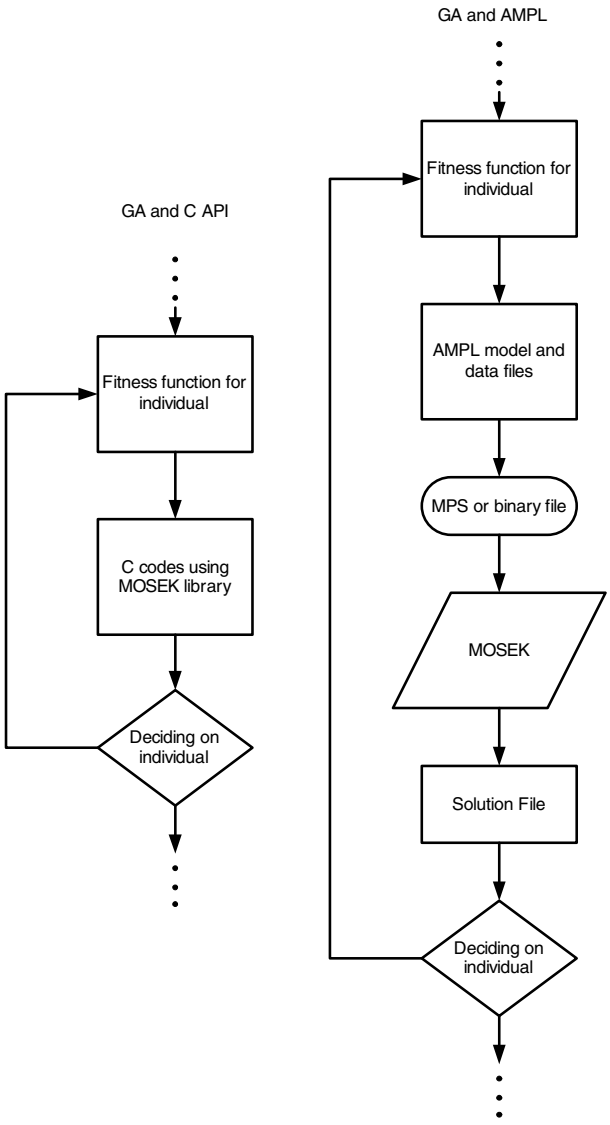


Fig. 6.1. GA using AMPL and C-based API

6.3. Genetic Encoding

In the approach of [15], the actual order of task execution on each processor is encoded in the genome. The system schedule can be built with task dependencies and execution times. For example, the system from Fig. 6.2 shows the scheduled tasks as $P_1[t_1, t_3, t_4, t_6, t_{10}]$, $P_2[t_2, t_7, t_9, t_{11}, t_{12}]$ and $P_3[t_5, t_8]$. These tasks are encoded like Fig. 6.3.

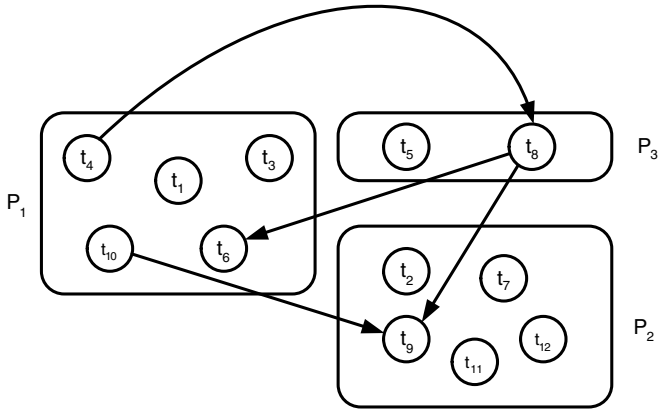


Fig. 6.2. An example application

The chromosome is divided in three zones corresponding to the three existing processors.

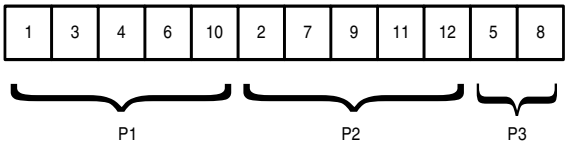


Fig. 6.3. Gene encoding

Each zone has as many genes as the number of tasks mapped on the corresponding processor.

6.4. Fitness Function

The goal is to find feasible schedules with low energy consumption. Thus the fitness must capture both of these aspects: the timing and the energy consumption. The fitness of feasible scheduling is given by the energy computed after performing voltage scaling. This fitness function smoothly integrates and guides the search for schedules that are both feasible and energy efficient. In our contribution we calculate the fitness for the given schedule, once with C-based API and once with AMPL using MOSEK and we calculate the time that it takes for each of them to optimize the energy function in Eq. (4.13).

6.5. Assumptions

Initial population- An initial population has to be supplied when starting the optimization process. It is generally better to have a diverse population. If the initial population is randomly chosen, it is likely that many of the schedules cannot be feasible. So we need to find the feasible schedules in optimization. As a middle approach, here, the initial population consists of several instances of a feasible schedule, produced by another algorithm (for example using a list scheduler) and of some randomly generated schedules. These randomly generated schedules may not be feasible.

Mutation- In our mutation technique the order of execution of two tasks that are mapped on the same processor, is swapped. We randomly select one processor and then again select randomly two tasks that are swapped. This random mutation could introduce a cycle in the mapped and scheduled task graph. After each mutation, the newly created chromosome is checked. If it contains a cycle the original chromosome is restored and a different mutation is tried.

Crossover- As it has been mentioned, crossover is used to create new individuals in the population, based on some common characteristics of the existing individuals. The technique used here is a novel edge recombination technique. From two randomly selected individuals of the current population ($Parent_1$ and $Parent_2$), two children ($Child_1$ and $Child_2$) are produced using the crossover as in the example in Fig. 6.4.
The edge recombination technique is used in order to preserve some of the properties of the parents. First of all, a region is selected randomly to perform the crossover in that region in both parents, (P2 in the example) then the edge

recombination is performed on the genes from the selected region and the results are copied to the corresponding region in the two children. For the other regions, the first child inherits the first parent and the second child second parent. A drawback of this approach is that it may create identical children to one of their parents.

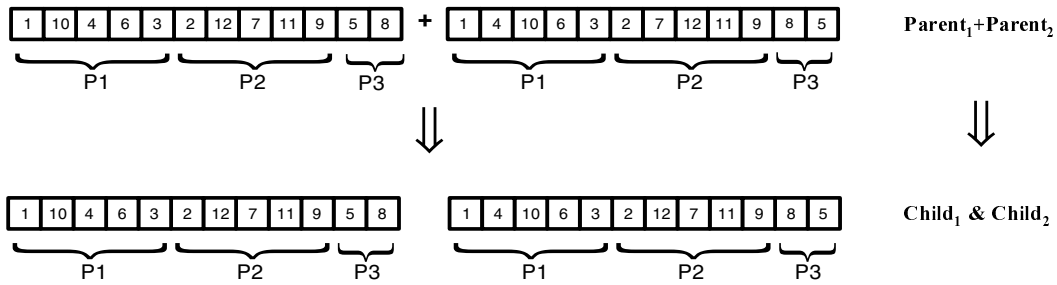


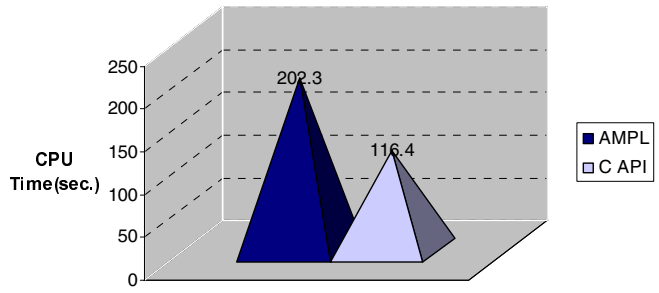
Fig. 6.4. Crossover Examples

6.6. Termination Criteria

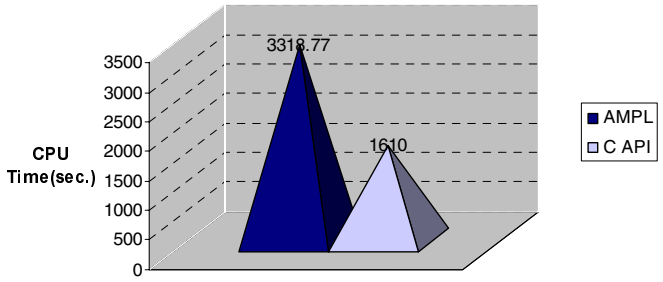
Because of the nature of the problem, finding the best schedule from the energy point of view is not easy. Thus, the algorithm cannot iterate until the optimal schedule is found. A typical genetic algorithm will run forever. We need to define when the algorithm should terminate. In our implementation the genetic algorithm finishes when a number of generations without a given improvement have been produced. Thus a no-improvement factor defines the maximum number of the iterations and refers to the number of generations without improvement.

7. Results for Experiment with GA

In this section we present the results of applying the genetic algorithm to task scheduling problem.
The experiments have been done for GA using AMPL and C-based API separately, to optimize the fitness function for each individual. We have conducted one experiment of 100 examples for 40 tasks, mapped in 4 processors. As it has been mentioned, we define a "No-improvement factor" which sets a maximum value for the number of generations without improvement. When this value is reached the algorithm will terminate. The no-improvement factor is set to 5 in first experiment. Another experiment for the same number of tasks and processors has been done with no-improvement factor 100. The results are shown in Fig. 7.1.



(a)

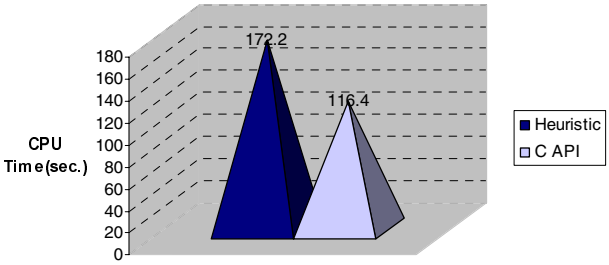


(b)

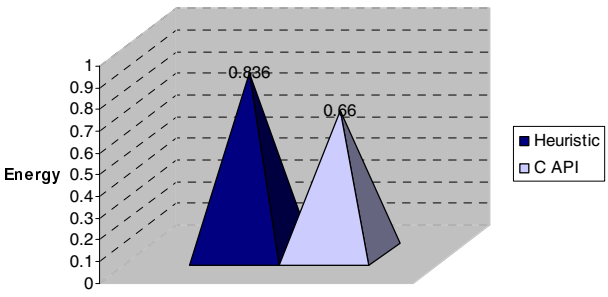
Fig. 7.1. Optimization times (a) No-Imp. Factor 5, (b) No-Imp. Factor 100.

In the experiments the initial population size was set to 100 and the number of generations was 50000. According to the experiments C-based API is 42% faster than AMPL. In another experiment we increase the no-improvement factor to 100. The result is shown in Fig. 7.1 (b). In this experiment our proposed solution solves the problem 51% faster. One reason for this increase comparing to first experiment is that AMPL allocates a new place in memory for each optimization process. When the no-improvement factor is set to a big value this means more iterations and the memory allocation will take time in AMPL case.

We have conducted another experiment to compare the results of C-based API with the heuristic introduced in [6]. In [6] a heuristic is used to decide on a particular mapping and scheduling. The energy value has been calculated before and after optimizing and the percentage of energy reduction has been measured for GA once for proposed method in [6] and once using the model of [3] and C-based API in optimization. The results are shown in Fig. 7.2.



(a)



(b)

Fig. 7.2. Heuristic and C-based API (a)Time (b)Energy after optimization

According to the results of the experiments, our solution is faster than the heuristic. The energy reduction in the heuristic case is 16.4% and in our solution is 33.5%.

8. Conclusion and Future Work

The growing class of portable systems, such as portable computers and personal communication devices, demand high performance and low power consumption. *DVS* and *ABB* are system level techniques which are employed to reduce the power consumption. Dynamic voltage scaling is a technique that offers a speed versus power trade-off, allowing the application to achieve considerable energy savings and, in the same time, to meet the imposed time constraints.

In this thesis, we explored the possibility of using optimal voltage scaling algorithms in a complex multiprocessor scheduling problem and we offered an equivalent implementation for optimization used in *DVS*, which is more time efficient than optimization used in [3].

Our work is focused on optimizing the modeled nonlinear problem formulation of continuous voltage selection problem in [3], excluding the consideration of transition overheads. Our experiment used the C-based API of the MOSEK solver [2] for optimization and the results were compared to the results for the speed of optimization from using AMPL.

The results show that the C-based API implementation was faster than AMPL-based implementation. This was particularly important when we used the genetic algorithm for finding a schedule that minimizes the energy consumption in multiprocessors. In this algorithm, the fitness function was calculated and optimized by our solution.

In another experiment we compared the efficiency of using the C-based API versus a heuristic proposed in [6]. The results show that using the proposed model in [3] and C-based API, is faster than heuristic and energy reduction after optimization is better than heuristic.

The energy function used in this optimization problem, contains only the dynamic power and V_{dd} scaling is the applied technique. So the energy function falls into the category of convex nonlinear objectives with linear constraints.
Future works can be done assuming both dynamic and leakage power. In this case combined *DVS* and *ABB* can be assumed as applied techniques. Thus optimization problem is changed to a convex nonlinear objective with nonlinear constraints which can be solved in future works.

APPENDIX A

Convexity

Convex set – A set S is convex if any point on the line segment connecting any two points in the set is also in S. Fig. a illustrates this property in two dimensions. An important issue in nonlinear programming is whether the feasible region is convex. The definition can be written as:

$$C \subseteq R^n \quad \text{is convex if}$$

$$x, y \in C, \quad \theta \in [0,1] \Rightarrow \theta \cdot x + (1-\theta) \cdot y \in C$$

When the all of the constraints in a problem are linear or convex, the feasible region is a convex set. If the objective function is a convex function and the feasible region defines a convex set, every local minimum is a global minimum. If the objective function is not a convex function, local minima may or may not be global minimum. A nonlinear programming algorithm may terminate at solution that is not a global minimum.

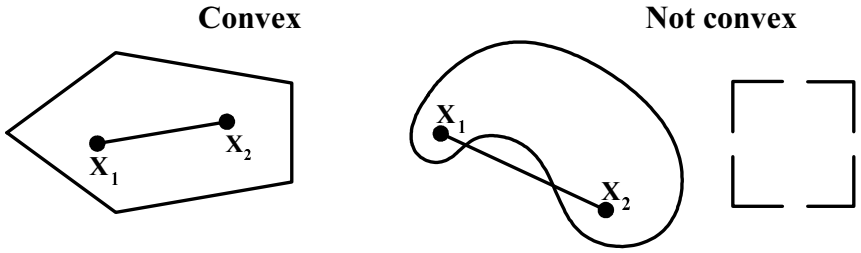


Fig. a. Examples

Convex function – When a straight line is drawn between any two points on a convex function, the line lies on or above the function. Fig. b shows a one-dimensional convex function.

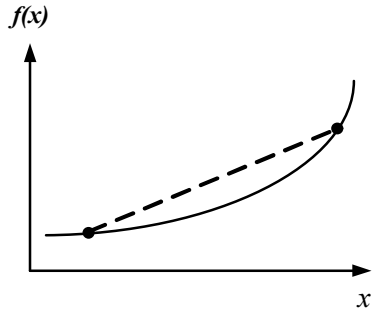


Fig. b. Convex function

In multiple dimensions a convex function has the following property. For every pair of solutions x_1 and x_2 :

$f : R^n \rightarrow R$ is convex if :

$f(\lambda \cdot x_1 + (1-\lambda) \cdot x_2) \leq \lambda \cdot f(x_1) + (1-\lambda) \cdot f(x_2)$ for all $0 < \lambda < 1.$

If f is convex, $-f$ is concave.

APPENDIX B

Gradient

Given a function f of n variables x_1, x_2, …,x_n, we define the partial derivative relative to variable x_i, written as $\frac{\partial f}{\partial x_i}$ to be the derivative of f with respect to x_i treating all variables except x_i as constant. The gradient of f at x, written as $\nabla f(x)$, is the vector:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x) \\ \frac{\partial f}{\partial x_2}(x) \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial f}{\partial x_n}(x) \end{bmatrix}$$

Hessian

Second partials $\frac{\partial f}{\partial x_i \partial x_j}(x)$ are obtained from *f(x)* by taking the derivative relative to x_i (this yields the first partial $\frac{\partial f}{\partial x_i}(x)$) and then taking the derivative of $\frac{\partial f}{\partial x_i}(x)$ relative to x_j. So second partials can be arranged into the Hessian matrix:

$$H(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1 \partial x_1}(x) & \frac{\partial f}{\partial x_1 \partial x_2}(x) & \cdot\ \cdot\ \cdot & \frac{\partial f}{\partial x_1 \partial x_n}(x) \\ \frac{\partial f}{\partial x_2 \partial x_1}(x) & \frac{\partial f}{\partial x_2 \partial x_2}(x) & \cdot\ \cdot\ \cdot & \frac{\partial f}{\partial x_2 \partial x_n}(x) \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{\partial f}{\partial x_n \partial x_1}(x) & \frac{\partial f}{\partial x_n \partial x_2}(x) & \cdot\ \cdot\ \cdot & \frac{\partial f}{\partial x_n \partial x_n}(x) \end{bmatrix}$$

APPENDIX C

Example: Minimize $\sum_{k=1}^{4}\left(A_k \cdot B_k \cdot x^2\right)$

Subject to: $\sum_{k=1}^{4}(y_k + x_k \le C_k)$

$\sum_{k=1}^{4}(y_{k+1} - y_k - x_k) \ge 0$

$\sum_{k=1}^{4} y_k \ge 0$ and $A_k \le \sum_{k=1}^{4} x_k \le 2 \cdot A$

C code using C-based API:

```
#include "scopt.h"
#define NUMOPRO 16 /* Number of nonlinear expressions in the objective. */
#define NUMOPRC 0 /* Number of nonlinear expressions in the constraint. */
#define NUMVAR  8   /* Number of variables.     */
#define NUMCON   7  /* Number of constraints.   */
#define NUMANZ  17  /* Number of nonzeros in A. A is the matrix that defines the
coefficiencies of variables in the constraints. A has the number of columns equal to
number of variables and number of rows equal to number of constraints. */

static void MSKAPI printstr(void *handle,char str[])
{
  printf("%s",str);
} /* To be able to get MOSEK messages. */

int main()
{
  char       buffer[MSK_MAX_STR_LEN];
  double     oprfo[NUMOPRO],oprgo[NUMOPRO],oprho[NUMOPRO],
             oprfc[NUMOPRC],oprgc[NUMOPRC],oprhc[NUMOPRC],
             c[NUMVAR],aval[NUMANZ],
             blc[NUMCON],buc[NUMCON],blx[NUMVAR],bux[NUMVAR];
    int      r,numopro,numoprc,i,k,p,j,
             numcon=NUMCON,numvar=NUMVAR,
          opro[NUMOPRO],oprjo[NUMOPRO],
```

```
        oprc[NUMOPRC],opric[NUMOPRC],oprjc[NUMOPRC],
        aptrb[NUMVAR],aptre[NUMVAR],asub[NUMANZ],
        bkc[NUMCON],bkx[NUMVAR];
MSKenv_t  env;
MSKtask_t task;
schand_t  sch;



/* Constant values.*/

static int A[4]={181296,224811,105654,398712};

static double C[4]={0.0391,0.0876,0.1104,0.1965};

static double B[4]={1e-09,2e-09,3e-09,4e-09};

static double OUT[8]={0};

k=0;
for (p=0; p<4; p++)

 {
  OUT[k] = C[p];
  OUT[k+1] = 0;
  k=k+2;
  }
for (i=0; i<4; i++)
 {


 /* To specify nonlinear terms in the objective. */

 numopro  = NUMOPRO;
 opro[i]  = MSK_OPR_THREE; /* Defined in scopt.h. In this file we define the
  nonlinear function in form of f(x+h)^g and the values for f, g and h should be
  defined.*/
 oprjo[i] = i;
 oprfo[i] = A[i]*B[i];
 oprgo[i] = 2;
```

```
 oprho[i] = 0.0;
```

/* Specifying matrix A*/

/* **Variable x_o.**/

```
    c[0] = 0.0;
    aptrb[0] = 0;    aptre[0] = 2;
    asub[0]  = 0;    aval[0]  = -1.0;
    asub[1]  = 1;    aval[1]  = -1.0;
```

/* **The upper and lower bounds for variable x_o.**/

```
    bkx[0] = MSK_BK_RA;
    blx[0]  = 2*A[0];
    bux[0] = A[0];
```

/* **Variable x_1.**/

```
    c[1] = 0.0;
    aptrb[1] = 2 ;      aptre[1] = 4;
    asub[2]  = 2;       aval[2]  = -1.0;
    asub[3]  = 3;       aval[3]  = -1.0;
```

/* **The upper and lower bounds for variable x_1.**/

```
    bkx[1] = MSK_BK_RA;
    blx[1]  = 2*A[1];
    bux[1] = A[1];
```

/* **Variable x_2.**/

```
    c[2] = 0.0;
    aptrb[2] = 4 ;      aptre[1] = 6;
    asub[4]  = 4;       aval[2]  = -1.0;
    asub[5]  = 5;       aval[3]  = -1.0;
```

/* **The upper and lower bounds for variable x_2.**/

```
        bkx[2] = MSK_BK_RA;
        blx[2]  = 2*A[2];
        bux[2] = A[2];
```

/* **Variable x_3.***/

```
        c[3] = 0.0;
        aptrb[3] = 6;     aptre[3] = 7;
        asub[6]  = 6;     aval[6]  = -1.0;
```

/* **The upper and lower bounds for variable x_3.***/

```
        bkx[3] = MSK_BK_RA;
        blx[3] = 2*A[3];
        bux[3] = A[3];
```

/* **Variable D_o.***/

```
        c[4] = 0.0;
        aptrb[4] = 7;        aptre[m] = 9;
        asub[7]  = 0;        aval[7]  = -1.0;
        asub[8]  = 1;        aval[8]  = -1.0;
```

/* **The upper and lower bounds for variable D_o.***/

```
        bkx[4] =  MSK_BK_LO;
        blx[4] = 0.0;
        bux[4] = MSK_INFINITY;
```

/* **Variable D_1.***/

```
        c[5] = 0.0;
        aptrb[5] = 9;          aptre[5] = 12;
        asub[9] = 1;           aval[9] = 1.0;
        asub[10] = 2;          aval[10] = -1.0;
        asub[11] = 3;          aval[11] = -1.0;
```

/* **The upper and lower bounds for variable D_1.***/

```
bkx[5] =  MSK_BK_LO;
blx[5] = 0.0;
bux[5] = MSK_INFINITY;
```

/* **Variable D_2.**/

```
c[6] = 0.0;
aptrb[6] = 12;          aptre[6] = 15;
asub[12] = 3;           aval[12] = 1.0;
asub[13] = 4;           aval[13] = -1.0;
asub[14] = 5;           aval[14] = -1.0;
```

/* **The upper and lower bounds for variable D_2.**/

```
bkx[6] =  MSK_BK_LO;
blx[6] = 0.0;
bux[6] = MSK_INFINITY;
```

/* **Variable D_3.**/

```
c[7] = 0.0;
aptrb[7] = 15 ;         aptre[m] = 17 ;
asub[15]  = 5;          aval[15]  = 1.0;
asub[16]  = 6;          aval[16]  = -1.0;
```

/* **The upper and lower bounds for variable D_3.**/

```
bkx[7] =  MSK_BK_LO;
blx[7] = 0.0;
bux[7] = MSK_INFINITY;
```

/*Specify bounds for the constraints*/

```
for (k=0;k>8;k++){
bkc[k] = MSK_BK_LO;
blc[k] = -OUT[k];
buc[k] = MSK_INFINITY;}
```

```
 /* Making the mosek environment. */

 r = MSK_makeenv(&env,NULL,NULL,NULL);

/* Checking whether the return code is ok. */

if ( r==MSK_RES_OK )
 {
  /* Directs the log stream to the user
     specified procedure 'printstr'. */
  MSK_linkfunctoenvstream(env,MSK_STREAM_LOG,NULL,printstr);
 }

 if ( r==MSK_RES_OK )
 {

  /* Initializing the environment. */

  r = MSK_initenv(env);
 }

 if ( r==MSK_RES_OK )

 {

  /* Making the optimization task. */

  r = MSK_makeemptytask(env,&task);


  if ( r==MSK_RES_OK )
    MSK_linkfunctotaskstream(task,MSK_STREAM_LOG,NULL,printstr);

  if ( r==MSK_RES_OK )
  {
    r = MSK_inputdata(task,
                      numcon,numvar,
                      numcon,numvar,
                      c,0.0,
                      aptrb,aptre,
```

```
                asub,aval,
                bkc,blc,buc,
                bkx,blx,bux);
}

if ( r== MSK_RES_OK )
{

 /* Set upping of nonlinear expressions. */

 r = MSK_scbegin(task,
                numopro,opro,oprjo,oprfo,oprgo,oprho,
                numoprc,oprc,opric,oprjc,oprfc,oprgc,oprhc,
                &sch);

 if ( r==MSK_RES_OK && 0 )
 {
  MSK_putintparam(task,MSK_IPAR_WRITE_GENERIC_NAMES,MSK_ON);
  r = MSK_scwrite(task,sch,"scopt");
 }

 if ( r==MSK_RES_OK )
 {
  printf("Start optimizing\n");

  r = MSK_optimize(task);

 }

   if ( r==MSK_RES_OK )
 {
    double xx[10];
       int j;
    MSK_getsolutionslice(task,
                    0,
                    MSK_SOL_ITEM_XX,
                    0,
                    NUMVAR,
                    xx);
```

```
            printf("Primal solution\n");
            for(j=0; j<NUMVAR; ++j)
            printf("x[%d]: %e\n",j,xx[j]);

      }

      /* The nonlinear expressions are no longer needed. */

  MSK_scend(task,&sch);
     }
    MSK_deletetask(&task);
  }
  MSK_deleteenv(&env);

  printf("Return code: %d\n",r);
  if ( r!=MSK_RES_OK )
  {
    MSK_getcodedisc(r,buffer,NULL);
    printf("Description: %s\n",buffer);
  }
}
```

Same example with AMPL:

Data file:

Example.dat

```
param n := 4;
param A := 1 181296 2 224811 3 105654 4 398712 5 435567;
param B := 1 1e-09 2 2e-09 3 3e-09 4 4e-09 5 5e-09;
param C := 1 0.0391 2 0.0876 3 0.1104 4 0.1965 5 0.2876;
```

Model file:

Example.mod

```
param n;
param A {i in 1..n};
param B {i in 1..n};
param C {i in 1..n};



var x {i in 1..n};
var y {i in 1..n};


minimize energy: sum {i in 1..n} (A[i]*B[i]*x^2);


subject to c1 {i in 1..n}: y[i]+x[i] <= C[i];
subject to c2 {i in 1..n}: y[i] >= 0;
subject to c3 {i in 1..n}: x[i] >= 2*A[i];
subject to c4 {i in 1..n}: x[i] <= A[i];
subject to c5: (y[2] - y[1] - x[1]) >= 0;
subject to c6: (y[3] - y[2] - x[2]) >= 0;
subject to c7: (y[4] - y[3] - x[3]) >= 0;
```

Input to solver:

Exampl.ampl

```
model Exampl.mod;
data Exampl.dat;
options solver mosek;
options  mosek_options 'MSK_DPAR_INTPNT_NL_TOL_PFEAS=1e-12


print x[1];
print x[2];
print x[3];
print x[4];
print y[1];
print y[2];
print y[3];
print y[4];
```

References

[1] Intel Corporation Website, Jan. 2005: http://www.intel.com/design/intelxscale/.

[2] MOSEK Website, Sep. 2004 – Jan. 2005 http://www.mosek.com.

[3] A. Andrei, M. Schmits, P. Eles, Z. Peng, and B.M. Al-Hashimi, "Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constraied systems," in *Proc. Design Automation & Test in Europe Conf.*, Vol.1, pp.518-523, Feb. 2004.

[4] O.S. Unsal and I. Koren, "System-level power-aware design techniques in real-time systems," in *Proc. of the IEEE.* Vol.91, Iss.7, pp.1055-1069, July 2003.

[5] D. Duarte, N. Vijaykrishna, M. j. Irwin, H. S. Kim, and G. Mcfarland, "Impact of scaling on the effectiveness of dynamic power reduction schemes," in *Proc. Int. Conf. Computer Design*, pp.382-387, Sept. 2002.

[6] M. T. Schmitz, B. M. Al-Hashimi, P. Eles, *System-level Design Techniques for Energy-efficient Embedded Systems*, Kluwer Academic Publishers, 2004.

[7] S. M. Martin. K. Flaunter, T. Mudge, and D. Blaauw, "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workload," in *Proc. Int. Conf. Computer Aided Design*, pp.721-725, Nov. 2002.

[8] K. Nose, M. Hirabayashi, H. Kawaguchi, S. Lee, and t. Sakurai, "V_{th} hopping scheme for 82% power saving in low-voltage processors," in *Proc. Custom Integrated Circuits Conf.* pp.93-98, May. 2001.

[9] C. H. Kim and K. Roy, "Dynamic V_{th} scaling scheme for active leakage power reduction," in *Proc. Design Automation & Test in Europe Conf.* pp.163-167, Mar. 2002.

[10] T. Ishihara and H. Yasuura. “Voltage scheduling problem for dynamically variable voltage processors,” in *Proc. Symp. Low Power Electronics and Design (ISLPED’98)*, pp.197-202, 1998.

[11] W. Kwon, and T. Kim, “Optimal voltage allocation techniques for dynamically variable voltage processors,” in *Proc. IEEE DAC’03*, pp.125-130, June 2003.

[12] Y. Zhang, X. Hu and D. Chen. “Task scheduling and voltage selection for energy minimization,” in *Proc. IEEE DAC’02,* June 2002.

[13] R. Fourer, D. M. Gay and B.W. Kernighan, *AMPL. A modelling language for mathematical programming*, Duxbury Press, Blemont, CA, 1997.

[14] D. Holms, “AMPL (A mathematical programming language)”, Documentation (Version 2), The University of Michigan, Aug. 1995.

[15] A. Andrei, “Energy efficient Real-Time scheduling of multiprocessor systems using a genetic algorithm”, Technical report, Linköping University, 2004.

[16] A. Marczyk, “Genetic Algorithms and Evolutionary Computation,” *The Talk. Origins Archive*, Dec. 2004; http://www.talkorigins.org/faqs/genalg/genalg.html.

[17] C.R. Revees, *Modern heuristic techniques for combinatorial problems*, Oxford, Blackwell, 1993.

[18] R. Baker, “Genetic Algorithms in Search and Optimization,” *Financial Engineering News*, Dec.2004; http://www.fenews.com/fen5/ga.html.