

# Power Optimizations for the MLCA Using Dynamic Voltage Scaling

Ivan Matosevic, Tarek S. Abdelrahman  
Edwards S. Rogers Sr. Department of  
Electrical and Computer Engineering  
University of Toronto  
{imatos,tsa}@eecg.toronto.edu

Faraydon Karim, Alain Mellan  
STMicroelectronics  
4690 Executive Drive  
San Diego, CA 92121  
{faraydon.karim,alain.mellan}@st.com

## ABSTRACT

Dynamic voltage scaling (DVS) is an effective method for reducing processor power consumption. We present a compiler-based technique for DVS-based power optimizations of multimedia applications in the context of the Multi-Level Computing Architecture (MLCA)—a novel architecture for parallel systems-on-a-chip. Our technique combines dependence analysis of long-running loops with profiling information in order to identify the slack available in the execution of parallel tasks. DVS is then applied to slow down processors executing non-critical-path tasks, reducing power with little or no impact on execution time. We evaluate our technique using realistic multimedia applications and a simulator of the MLCA. The results demonstrate that up to 10% savings in processor power consumption can be achieved with no more than 1.5% increase in execution time. Although our technique is developed in the context of MLCA, we believe that it is applicable in the broader context of task-level parallelism in multimedia applications.

## Categories and Subject Descriptors

C.1 [Processor architectures]: Parallel architectures; C.3 [Special purpose and application-based systems]: Real-time and embedded systems; C.4 [Performance of systems]: Design studies

## General Terms

Algorithms, Performance

## Keywords

Parallel-embedded systems, compiler techniques for power, dynamic voltage scaling, power optimizations, multimedia applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCOPES'05 Dallas, Texas, USA

Copyright 2005 ACM X-XXXXXX-XX-X/XX/XX ...\$5.00.

## 1. INTRODUCTION

The *Multi-Level Computing Architecture* (MLCA) [11] is a novel architecture for parallel systems-on-a-chip (SOCs). It features multiple processing units and a top-level controller that automatically exploits parallelism among coarse-grain units of computation, called *tasks*, using techniques similar to those used by superscalar processors for the extraction of instruction-level parallelism. The MLCA supports a programming model that is similar to sequential programming, making the MLCA an attractive architecture for multimedia and streaming applications.

Power consumption remains one of the critical design constraints in today's embedded systems. These systems more often than not run on batteries, yet improvements in battery technology have failed to keep pace with increased system power consumption, particularly the power consumption of processors [13]. Consequently, techniques for power consumption reduction have attracted considerable interest in recent years.

Dynamic Voltage Scaling (DVS) is one of the main techniques for reducing the power consumption of processors. Using DVS, the supply voltage and operating frequency of a processor can be varied during run-time in order to achieve a trade-off between processor performance and power dissipation. Today, there exists a number of DVS-enabled processors, including the Intel XScale [9], the IBM PowerPC 405LP [17], and the Transmeta Crusoe processor [21].

In this paper, we investigate the use of DVS for the MLCA. More specifically, we consider the use of DVS-enabled processors as processing units in the MLCA and propose a novel profile-driven compiler technique for voltage selection and task scheduling for MLCA applications. The proposed technique targets long-running loops. It combines analysis of the loop dependence graph with profiling information in order to deduce properties of the dynamic task graph that represents the run-time execution of tasks in the loop. Based on these deduced properties, our algorithm computes the optimal voltage level for the execution of each task. The task scheduling scheme is also based on the properties of the task graph and complements the voltage selection algorithm.

Previous work in the area of DVS-based power optimizations for parallel systems, which we survey in Section 5, has focused on real-time applications with small task graphs (up to several tens of tasks), possibly executed periodically. For such applications, the problem of power optimization is reduced to a single instance of the task graph, which can be analyzed using computationally-intensive algorithms. In

contrast, our work targets real MLCA applications, in which task-level parallelism is extracted from programs by means akin to the parallel execution of machine instructions in a superscalar processor. The task graph of such an application is fully defined only at run-time. Even if we introduce simplifying assumptions about the control-flow that enable the task graph to be known at compile time, the task graph can be arbitrarily large and therefore unanalyzable using previously proposed techniques. Furthermore, the run-time task graph of a loop in an MLCA program cannot be partitioned into small subgraphs that could each be analyzed separately. This is because most of the parallelism in these task graphs stems from pipelining loop iterations, resulting in cross-iteration dependences that prevent partitioning of the task graph. Thus, the novelty of our technique lies in identifying regularities in the control-flow of loops in a wide range of multimedia applications, in using these regularities to infer at compile-time the properties of the run-time task graph, and in novel heuristic algorithms for voltage selection and task scheduling capable of handling such task graphs.

We implement and evaluate our technique using 3 realistic multimedia applications and a simulator of the MLCA [11]. The results indicate that our technique is successful at reducing processor power consumption by 5–10%, with minimal increase in execution time (no more than 1.5%). Although our technique specifically targets the MLCA, we believe that it is applicable in the more general context of task-level parallelism in multimedia applications based on pipelining the processing of individual units in the input media stream.

The remainder of this paper is organized as follows. Section 2 gives an overview and our assumptions of the MLCA, the task execution model, and dynamic voltage scaling. Section 3 formulates the problem and presents our technique. Section 4 describes our experimental evaluation. Section 5 surveys related work in the area of DVS-related power optimizations. Finally, Section 6 presents concluding remarks and directions for future work.

## 2. BACKGROUND

### 2.1 The MLCA

The MLCA [11] is a novel 2-level hierarchical architecture, aimed at parallel SOCs and primarily intended for multimedia applications. The lower level consists of multiple *processing units* (PUs), and the upper level of a controller that automatically exploits parallelism among coarse-grain units of computation, or *tasks*. A PU can be a full-fledged processor core, a DSP, a block of FPGA, or any other type of programmable hardware. The top-level controller consists of a *control processor* (CP), a *task dispatcher* (TD), and a *universal register file* (URF). A dedicated interconnection network links the PUs to the URF and memory, as shown in Figure 1(a). In this paper, we assume that the target MLCA architecture features a homogeneous set of PUs with uniform access to a shared memory.

The novelty of the MLCA stems from the fact that the upper level of the hierarchy supports parallel execution of tasks, using the same techniques used in superscalar processors, such as register renaming and out-of-order execution. This leverages existing processor technology to exploit task-level parallelism across PUs, in addition to possible instruction-level parallelism within each task. The similar-

ity of the MLCA to the microarchitecture of a superscalar processor can be seen in Figure 1.

The MLCA supports a programming model that, similar to sequential programming, does not require programmers to specify task synchronization and inter-task communication. It only requires programmers to express an application in terms of a sequential *control program*, which contains *task instructions*, and a set of *task functions*, each a sequential function with a specified number of input and output URF registers. The control program is executed by the CP, and the task functions are executed by the PUs. At run-time, each *task* executes one of the task functions, and corresponds to a task instruction executed by the CP.

The CP fetches and decodes task instructions, each of which specifies the inputs and outputs of the task as registers in the URF. Data dependences among task instructions are detected by identifying the source and sink registers in the URF, in the same way that dependences among instructions are detected in a superscalar processor. The CP renames URF registers as necessary to break false dependences among task instructions. Decoded task instructions are then issued to the TD unit. The default task scheduling scheme in the MLCA system is based on a simple FIFO task queue, which orders the task instructions according to their sequential execution order, and round-robin selection of PUs. Based on the dependences that occur at run-time, tasks can be issued out of order, and may complete and commit their outputs also out of order. Besides the task instructions, the control program can also contain control-flow instructions. Conditional branches are implemented by means of a set of control registers in the CP, which can be written by task instructions.

In order to port a sequential application to the MLCA, it is necessary to partition the sequential program code into task functions and write the control program. Once the task functions are formed, they can be compiled using standard compilers for the architectures used as PUs in the MLCA. The MLCA benchmark applications used in this paper are ported to the MLCA manually. However, research into compiler support for the MLCA that would automate large parts of this process is in progress [2].

The control program is written in an assembler-like language called *HyperAssembly*. An example of a control program is shown in Figure 2(a). It shows a single loop, executing task instructions  $S_1$ – $S_4$  that invoke task functions  $T_1$ – $T_4$ , respectively. The type of access for each register is indicated as read ( $r$ ) or write ( $w$ ) next to the register symbol. Task function  $T_4$  writes to the control register  $CR1$ , and the subsequent conditional branch checks this register.

The tasks in the above example must be executed sequentially, because of the true data dependences  $S_1\delta^t S_2$  and  $S_3\delta^t S_4$ , as well as false dependences  $S_1\delta^o S_3$  and  $S_2\delta^a S_3$ . However, the CP renames registers at run-time to break false dependences and thus allow some parallel execution. The control program after register renaming is shown in Figure 2(b). With both false dependences eliminated,  $T_3$  can be executed in parallel with  $T_1$ , and after  $T_3$  writes its outputs,  $T_4$  can proceed regardless of the status of  $T_1$  and  $T_2$ . Furthermore, since the direction taken by the branch instruction  $S_5$  depends on the result written into  $CR1$  by  $T_4$ , there is a control dependence between  $S_4$  and the subsequently executed task instructions.

The number of renaming registers impacts the perfor-



Figure 1: Comparison between the MLCA and a superscalar processor

```

S1: task T1, R1:r,R2:w,R3:w
S2: task T2, R2:r,R3:r,R4:w
S3: task T3, R1:r,R5:r,R2:w,R3:w
S4: task T4, CR1, R2:r,R3:r,R5:w
S5: if (CR1 & 0x02) goto S1
S6: stop

```

(a) Original code

```

S1: task T1, R1:r,R2:w,R3:w
S2: task T2, R2:r,R3:r,R4:w
S3: task T3, R1:r,R5:r,R101:w,R102:w
S4: task T4, CR1, R101:r,R102:r,R5:w
S5: if (CR1 & 0x02) goto S1
S6: stop;

```

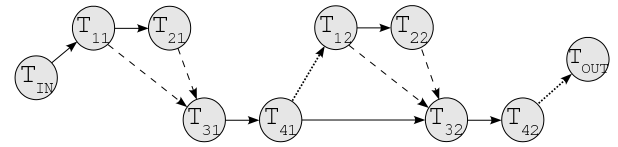
(b) After register renaming

Figure 2: An example of HyperAssembly code and register renaming

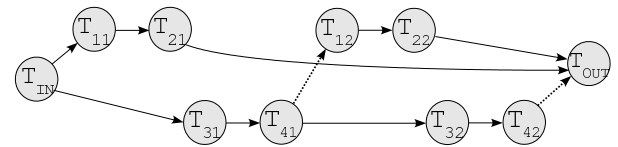
mance of MLCA programs; a larger number of renaming registers allows more false dependences to be eliminated [11]. In this work, we assume that the number of renaming registers is large enough to eliminate all false data dependences, and that the number of processors is the maximum allowed by the scalability of the applications. This is reasonable since we expect that maximization of parallelism will be one of the goals of the system design. Besides, applications running on a system with a number of processors smaller than the maximum tend to have high levels of processor utilization and therefore do not offer significant opportunities for power optimizations using DVS.

## 2.2 Execution Model

The execution of the control program by the CP at runtime can be represented by a directed acyclic graph, which we refer to as the *task graph*. The nodes of the task graph represent individual tasks (i.e. instances of task instructions), and the edges represent dependences between pairs of tasks. We add two additional nodes  $T_{IN}$  and  $T_{OUT}$  to the task graph, corresponding to program entry and exit. Figure 3 shows the task graph of the HyperAssembly program in Figure 2, with and without the register renaming. In this example, we assume that two loop iterations are executed before the exit from the loop. For simplicity, in cases



(a) Without register renaming



(b) With register renaming

Figure 3: Task graph of the HyperAssembly program from Figure 2. Solid lines denote true data dependences, dashed lines false data dependences, and dotted lines control dependences.

when there is more than one dependence between a pair of tasks, we show only one edge in the graph. In the figure, symbol  $T_{Ni}$  denotes the task that executes the task function  $T_N$  in loop iteration  $i$ . From the task graph, it is obvious that the register renaming eliminates the false data dependences, enabling the parallel execution on two processors of the originally sequential task graph.

The nodes of the task graph are labeled by the execution times of tasks (not shown in Figure 3). The minimal execution time of the application is equal to the length of the longest path in the task graph, i.e. the *critical path*. The execution of a program on the MLCA is equivalent to the scheduling of the task graph on the corresponding set of PUs using the task scheduling algorithm employed by the MLCA control unit.

We assume that the compiler system is capable of collecting profile data by running the application or simulating its execution. However, we do not require knowledge of the absolute values of the task execution times, but only the ratios of the average values of execution times of individual task instructions. In the theoretical derivation of our algorithms, we assume that the execution time of each task function does not vary across loop iterations. In practice, this assumption

almost never holds strictly, since the execution times of task functions normally depend on input. We approximate the ideal case by assigning to each task instruction its average execution time measured by profiling the application execution with a set of training inputs. The positive results of the experimental evaluation presented in Section 4 confirm that this approximation is very reasonable.

### 2.3 Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) allows programs to change at run-time the supply voltage and frequency of a processor in order to trade performance for lower power consumption [4]. Programs may lower the supply voltage to reduce the power consumption of the processor (the dynamic power dissipation of the processor is proportional to  $f \cdot V^2$ , where  $f$  is the operating frequency and  $V$  the supply voltage [1]). However, lowering the supply voltage also reduces the frequency of the processor, resulting in degraded performance.

In recent years, a number of DVS-enabled processor designs has appeared. Examples are the Intel XScale [9, 5], IBM’s PowerPC 405LP [17], and Transmeta’s Crusoe [21]. These processors support several discrete voltage levels with different frequencies and rates of power consumption. The transition between levels can be performed at run-time by executing specific machine instructions. We assume that the DVS capabilities of the PUs in the MLCA system are implemented similarly.

The relation between the processor operating frequency and the execution time of a given sequence of instructions is non-trivial, mainly because at lower frequencies the number of stall cycles will usually be lower, thus reducing the performance loss. However, as a simplifying assumption, we *conservatively* assume that the execution time of each task is inversely proportional to the processor frequency.

Switching between voltage levels at run-time incurs certain overheads in time and energy, which generally depend on the levels between which the processor is transferring. Unlike most previous work on DVS-based power optimizations, which we survey in Section 5, we take the effect of the transition overheads into consideration. Also, we assume that the time and energy overheads are constant and independent of voltage levels. This simplification does not significantly affect the accuracy of our technique; Mochocki et al. [16] suggest that the assumption of constant transition overheads is reasonable.

Besides the DVS, another power management feature of modern embedded processors is the possibility of entering a lightweight idle mode implemented using clock gating [3]. This idle mode does not reduce the processor power to a negligible level, but it can be entered and exited with a negligible time overhead. For example, the Intel 80200 processor—based on the XScale core—takes several tens of clock cycles to exit the idle mode [9]. We assume that each processor in the MLCA system spends time in such idle mode whenever it is not executing a task.

## 3. VOLTAGE SELECTION AND TASK SCHEDULING ALGORITHMS

Our technique identifies the critical path in an execution task graph and prolongs the execution time of tasks outside of the critical path using DVS in a manner that does not introduce a new longer critical path, thus achieving power

savings with little or no impact on the application performance. Our assumption is that the processor frequency in the MLCA system is chosen so as to meet the real-time requirements of the application while taking full advantage of the maximum achievable parallel speedup. Thus, power savings cannot be achieved by permanently reducing the voltage and frequency of the processors, because this would violate the real-time requirements of the application. Instead, our technique attempts to achieve power savings in a manner that either does not slow down the application or incurs a small slowdown while achieving power savings higher than those that could be gained by permanently slowing the processors by the same factor.

We define the *slack* of a task as the maximum time by which the execution time of the task can be prolonged without affecting the overall application execution time. We define the total slack in the application as the total additional execution time available for distribution across tasks whose execution time can be prolonged. The task graph shown in Figure 3(b) is used to illustrate the distribution of slack using DVS. We assume that the execution time of task functions  $T_1$  and  $T_2$  is  $1000\mu s$ , while the execution time of task functions  $T_3$  and  $T_4$  is  $1500\mu s$ . Path  $(T_{IN}, T_{31}, T_{41}, T_{32}, T_{42}, T_{OUT})$  is the critical path in this task graph. If the task graph is scheduled on two processors, the minimal execution time is  $6000\mu s$ . However, the execution time of the series of tasks  $(T_{11}, T_{21}, T_{12}, T_{22})$ , which are running in parallel with the critical path, is only  $4000\mu s$ . Therefore, in the task graph there exists a slack of  $2000\mu s$ , which can be distributed over tasks  $T_{11}$ ,  $T_{21}$ ,  $T_{12}$ , and  $T_{22}$  without increasing the overall execution time. If the execution time of each of these tasks is increased by up to  $500\mu s$ , the minimal total execution time of the task graph on two processors is still  $6000\mu s$ . This minimal execution time can be achieved, for example, by scheduling the tasks from the critical path as an uninterrupted sequence on one processor, and the remaining four tasks on the other one. The increase in execution time of the four tasks can be realized using DVS by lowering the supply voltage, thus achieving a reduction in power consumption without increasing the execution time.

The basic idea of using DVS to slow down the execution of non-critical tasks in a task graph has been explored in earlier work [1, 7, 18, 20, 24]. However, this previous work has focused on small task graphs that are known at compile-time, with the number of tasks on the order of several tens. The challenge we address in our work is applying the DVS technique to MLCA programs. The task graphs of these programs are known only at run-time, and since the number of tasks is equal to the number of executed task instructions, they are large enough to preclude the computationally-intensive analysis common to previous work. Furthermore, these task graphs cannot be partitioned into small subgraphs that could be each analyzed independently, because the main source of parallelism in them is pipelining of the loop iterations. This results in loop-carried dependences which prevent the partitioning of the task graph.

We exploit the characteristics of multimedia applications to address the above challenge. A typical multimedia application consists of some initialization and clean-up code, each executed only once, and one or more long-running loops. These loops handle the processing of the input media stream, and account for the bulk of the execution time. The iterations of each of these loops are not serialized by control

dependencies; the CP of the MLCA executes tasks that update iteration counters in advance and out of order, making it safe to ignore control dependences. Furthermore, these loops often do not contain any control-flow instructions in their bodies, making each task read from and write to the same set of logical URF registers in each iteration. We refer to each of these loops as a *target loop*. These characteristics enable us to deduce properties of the run-time task graph using dependence analysis of the loop and collected profile information.

### 3.1 Algorithm Overview

The identification of non-critical tasks, computation of the available slack, and the optimal distribution of this slack are difficult problems. Furthermore, the available slack depends not only on the application program, but also on the employed task scheduling scheme. Therefore, we divide our approach into two sub-problems: *voltage selection* and *task scheduling*. The algorithms used to solve these two problems must be designed so as to complement each other. Furthermore, in practice, task graphs are complex and heuristic solutions to these problems often result in some increase in application execution time.

Given the body of target the loop that contains  $N$  task instructions  $S_1, \dots, S_N$ , each of which invokes a task function  $f_n$  (it is possible that  $f_i = f_j$  for  $i \neq j$ ), we find the voltage level  $L_n$  for each task instruction  $S_n$  in the loop body, the processor mapping for each task dispatched during the loop execution, and the ordering of tasks mapped onto each processor, such that the energy consumed by the processors during the loop execution is minimized. These computations are performed under the constraints that the task schedule does not violate the precedence constraints imposed by the data dependences, and the execution time of the application is not degraded in comparison to the execution time of the non-optimized application with the default MLCA task scheduling scheme. We analyze each target loop in order to:

1. Compute the total available slack in the task graph.
2. Identify the tasks in the target loop that can be executed at lower processor voltage level and find the optimal distribution of the slack across this set of tasks.
3. Schedule the task graph in a way that complements our method for voltage selection.

In the remainder of this section, we describe the dependence analysis of the loop, computation and distribution of the available slack, and task scheduling.

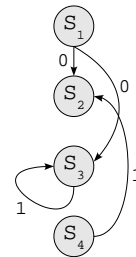
### 3.2 The Loop Dependence Graph

The *dependence graph* [12] of the target loop is a directed graph whose nodes represent task instructions from the target loop, and whose edges represent the data dependences between pairs of task instructions. We label each node by the execution time of its corresponding task function. We label each dependence edge with the *distance* of the dependence, which is defined as the distance in loop iterations between the sink and source of the dependence. All dependences in this graph are true data dependences, since the hardware eliminates the false dependences at run-time and the control dependences can be safely ignored.

We use the symbol  $T_{ni}$  to denote the task that executes the task instruction  $S_n$  in the loop iteration  $i$ . Assume there

```
S1: task t1, R1:w,R2:w
S2: task t2, R1:r,R4:r
S3: task t3, R2:r,R3:r,R3:w
S4: task t4, CR1, R4:w
S5: if (CR1 & 0x01) goto S1
```

(a) HyperAssembly code



(b) Dependence graph

Figure 4: Dependence graph of an example loop

exists a dependence  $T_{mi} \delta^t T_{nj}$  that arises due to the existence of one or more registers that are written by task  $T_{mi}$  and read by task  $T_{nj}$ . Obviously,  $i \leq j$  must hold, because the dependence source must precede its sink in the sequential execution order. However, all logical registers written by task  $T_{mi}$  are also written by task  $T_{m(i+1)}$ . Therefore, in the sequential execution order, task  $T_{mi}$  must precede task  $T_{nj}$ , but task  $T_{nj}$  must precede task  $T_{m(i+1)}$ . This is possible only if  $m \geq n$  and  $i = j - 1$ , or  $m < n$  and  $i = j$ . In the former case, the dependence is loop-carried and has the dependence distance of 1. In the latter case, the dependence distance is 0 and the dependence is loop-independent. A simple example of a loop and its associated dependence graph is shown in Figure 4.

### 3.3 Available Slack

In this section we show how to deduce the critical path in the task graph of a target loop and compute the available slack using its loop dependence graph.

The task graph of the loop execution contains two types of edges. Edges of the first type connect tasks executed within the same iteration and correspond to the edges from the loop dependence graph for which the dependence distance is zero. Edges of the second type connect tasks executed within two consecutive iterations and correspond to the edges from the loop dependence graph for which the dependence distance is one. A part of the task graph of the execution of the loop in Figure 4 is shown in Figure 5.

Since for each edge in the loop dependence graph  $(S_i, S_j)$  whose dependence distance is zero,  $i < j$  must hold, each cycle in the loop dependence graph must contain at least one edge with dependence distance of one. We define a *1-cycle* as a cycle in the loop dependence graph that contains exactly one edge with the dependence distance of one. Each 1-cycle in the loop dependence graph translates into a path in the task graph that stretches across all loop iterations. For example, the cycle consisting of node  $S_3$  in the loop dependence graph from Figure 4 translates into the path across all tasks  $T_{3i}$  in the task graph from Figure 5, which



and  $S'_k$  when run at the voltage level  $j$  using the symbols  $t_{ij}$  and  $t'_{kj}$ . These times are constants in the model. Therefore, the execution times of the task instructions  $S_i$  and  $S'_k$  can be represented by the linear expressions  $\alpha_{i1}t_{i1} + \dots + \alpha_{iL}t_{iL}$  and  $\beta_{k1}t'_{k1} + \dots + \beta_{kL}t'_{kL}$ .

We introduce  $m + 1$  binary variables  $c_0, \dots, c_m$  that encode the voltage transitions in the first group of scaled instructions. The value of  $c_0$  is 1 if a transition takes place immediately prior to the execution of the instruction  $S_1$ , and 0 otherwise. For  $1 \leq i \leq m$ , the value of  $c_i$  is 1 if a transition takes place immediately after the execution of  $S_i$ , and 0 otherwise. We define  $n + 1$  binary variables  $d_0, \dots, d_n$  that encode the transitions in the second group of scaled instructions in the same manner. The definitions of variables  $c_i$  and  $d_k$  can be expressed by the sets of linear constraints  $c_i \geq \alpha_{ij} - \alpha_{(i+1)j}$  and  $d_k \geq \beta_{kj} - \beta_{(k+1)j}$ , for all  $i = 1, \dots, m - 1$ ,  $k = 1, \dots, n - 1$ , and  $j = 1, \dots, L$ . For the borderline variables, there are special constraints  $c_0 = 1 - \alpha_{1L}$ ,  $c_m = 1 - \alpha_{mL}$ ,  $d_0 = 1 - \beta_{1L}$ , and  $d_n = 1 - \beta_{nL}$ .

We formulate the following constraint on the total execution time of the scaled instructions:

$$\sum_{i=1}^m \sum_{j=1}^L \alpha_{ij} t_{ij} + \sum_{k=1}^n \sum_{j=1}^L \beta_{kj} t'_{kj} + \sum_{i=0}^m c_i t_{TR} + \sum_{k=0}^n d_k t_{TR} \leq t_{min} + r \cdot t_{slack},$$

where  $t_{TR}$  is the transition time,  $t_{min}$  is the total execution time of the scaled instructions at the highest processor voltage level,  $t_{slack}$  is the slack time computed according to the formula 2, and  $r$  is the tunable parameter of the algorithm.

Increases in the execution times of scaled instructions must not result in certain 1-cycles in the loop graph becoming longer than the longest 1-cycle, which determines the critical path. Therefore, for each 1-cycle  $c$  that contains one or more scaled instructions, a constraint must be formulated:

$$\sum_{S_i \in c} \left( c_i t_{TR} + \sum_{j=1}^L \alpha_{ij} t_{ij} \right) + \sum_{S'_k \in c} \left( d_k t_{TR} + \sum_{j=1}^L \beta_{kj} t'_{kj} \right) < \tau_C - \tau_n,$$

where  $\tau_C$  is the length of the longest 1-cycle,  $\tau_n$  is the total execution time of all non-scaled task instructions in cycle  $c$  (which is a constant in the model), and the sums are over all scaled task instructions that belong to  $c$ .

The objective function to be minimized is the sum of the energy consumed by each task instruction, including the energy overhead of the transitions, and the total energy consumed by the processors in the idle mode. We use the symbol  $E_{ij}$  to denote the energy consumed by the task instruction  $S_i$  executed at the voltage level  $j$ . We define  $E'_{kj}$  for each task instruction  $S'_k$  similarly. We use  $E_{TR}$  to denote the energy consumed by a single transition. Since these quantities are constants in the model, we can account for the active and transition power in the objective function by multiplying them with the corresponding variables  $\alpha_{ij}$ ,  $\beta_{kj}$ ,  $c_i$ , and  $d_k$ . We can account for the idle power by noting that each increase in task execution time and each transition overhead reduces the total processor time spend in idle mode. Therefore, if we define the constants  $\eta_{ij} = E_{ij} - P_{idle} t_{ij}$ ,  $\eta'_{kj} = E'_{kj} - P_{idle} t'_{kj}$ , and  $\epsilon = E_{TR} - P_{idle} t_{TR}$ , we can for-

mulate the following linear objective function:

$$\sum_{i=1}^n \sum_{j=1}^L \alpha_{ij} \eta_{ij} + \sum_{k=1}^m \sum_{j=1}^L \beta_{kj} \eta'_{kj} + \sum_{i=0}^m c_i \epsilon + \sum_{k=0}^n d_k \epsilon. \quad (3)$$

Solving the ILP model with this objective function and the previously listed constraints yields the optimal distribution of the available slack among the scaled tasks. In our implementation of the voltage selection algorithm, we use the freely available LP\_SOLVE optimization library [3].

### 3.5 Task Scheduling

The tasks are divided into three groups by decreasing priority: the critical tasks, the scaled non-critical tasks, and all remaining tasks. Within each group, the priority is determined according to the sequential execution order. The mapping of the tasks onto the processors is performed according to the following rules. A single processor is reserved for the execution of the critical tasks exclusively. All critical tasks are executed as an uninterrupted sequence on this processor. Non-critical tasks are scheduled on the remaining processors according to the round-robin scheme, with a single exception: once the first task in a set of scaled tasks from an iteration has been scheduled onto a processor, all remaining tasks from the same set of scaled tasks are scheduled onto the same processor.

The stated rules for task ordering and processor mapping imply that each set of scaled tasks is executed as an uninterrupted sequence, thus ensuring that the voltage selection by the algorithm described in Section 3.4 is indeed optimal for the given amount of slack per loop iteration. The necessary run-time computations in the control unit are simple enough to be realized in an MLCA system with a small overhead in hardware complexity.

## 4. EVALUATION

In this section, we evaluate our technique using three realistic MLCA multimedia applications: a GSM voice encoder, a JPEG image encoder, and an MPEG sound decoder. We use a simulator of the MLCA [11] to collect the necessary profiling information. For each application, we measure the energy savings and the application slowdown compared to the default MLCA task scheduling scheme. The parallel speedup<sup>1</sup> of these three applications is shown in Figure 6. We expect each application to execute on the number of processors that represents the scaling limit. For JPEG, GSM, and MPEG, these numbers are 6, 4, and 11 respectively.

These three applications spend the majority of their execution time in long-running loops whose bodies lack control-flow instructions and whose iterations are not serialized by control dependences, which makes them suitable for the application of our technique. However, for JPEG and MPEG, the execution times of most task functions vary significantly between loop iterations and depend on the input content. For each of these two applications, we perform the profiling run over a set of several different inputs, which we refer to as the *training set*. For the parameters of our algorithms, we use the average of the profiling results measured over

<sup>1</sup>Since the MLCA applications are inherently parallel and there is no equivalent of the sequential program version for them, we define the parallel speedup as the ratio of the application execution time on a 1-processor MLCA and an  $N$ -processor MLCA.

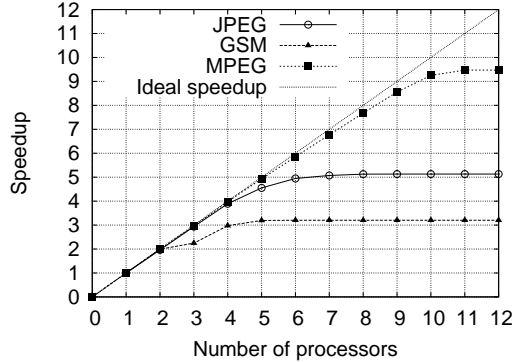


Figure 6: Speedup of the MLCA applications

individual inputs from the training set. Using these parameters, we apply our technique to the original training set, as well as another set of different inputs, expecting to achieve similar positive results for both sets.

#### 4.1 Experimental Platform and Processor Properties

Our implementation of the algorithms for task scheduling and voltage selection reads the task graph of the application execution and the profiling information from the output of the MLCA simulator. For the given task graph of the application execution, the program first schedules the task graph using the default MLCA task scheduling scheme, without applying the DVS, and determines the application execution time and power consumption for the default MLCA system configuration. Subsequently, it computes the voltage selection, modifies the task graph according to the effects of DVS, including the transition overheads, schedules the modified task graph using the task scheduling scheme described in Section 3.5, and computes the changes in the application execution time and power consumption compared to the default MLCA system configuration.

We assume that the processor cores in the MLCA system support eight discrete voltage levels, with the relation between supply voltage, operating frequency, and power consumption characteristic of the Intel XScale processor core [5]. The relative frequency and power consumption for each level is shown in Table 1. When a processor is not performing any useful work, we assume that it enters the idle mode in which the power consumption is 20% of the power consumption at the highest voltage level. We assume that the overhead of transitions between voltage levels is 1000 cycles, during which the power is 80% of the power at the highest voltage level. The task execution times of JPEG, GSM, and MPEG are mostly on the order of several tens of thousands of cycles, so that the transition overheads assumed in our model are not negligible. We expect that our results will generalize to the future MLCA systems executing more computationally-intensive applications, with tasks of coarser granularity, in the presence of higher transition overheads.

#### 4.2 JPEG Encoder

We evaluate the performance of our technique on the JPEG encoder using two sets of 12 photographic images. The two sets were taken on different occasions. The first set is used as the training set, and our technique is then applied to both

Table 1: Properties of the processor voltage levels

Level	1	2	3	4
Power	0.167	0.272	0.371	0.470
Slowdown	0.417	0.500	0.583	0.667
Level	5	6	7	8
Power	0.557	0.654	0.768	1.000
Slowdown	0.750	0.833	0.917	1.000

sets with parameters computed from the profiling data for the training set. The results are shown in Figure 7, which shows the reduction in the processor energy consumption and the increase in the application execution time resulting from the application of our technique, as a function of  $r$ , the fraction of computed slack used during voltage selection (see Section 3.3). All results are computed as weighted averages over each set of images, where the weights assigned to individual images are proportional to the overall computational work necessary for encoding each image. Weighted averages are used because the computational work varies significantly between images. The negative slowdown at certain points means that despite the increase in the execution time of the scaled tasks, the overall application execution time is decreased by that percentage by the application of our task scheduling scheme.

The results for the second set of images are slightly better than for the training set, but the difference is too small to be significant. For  $r = 0.5$ , our technique achieves power savings of over 9.2%, slowing down the application execution by less than 0.9%. Figure 8 shows the results for each individual image from the second input set with  $r = 0.5$ . For certain images, the slowdown is relatively high, but the average slowdown is small. The power savings are achieved consistently for all images.

#### 4.3 GSM Encoder

Since the GSM encoder application is characterized by very small variations in the execution time of the task functions, we use the profiling information for a single input and demonstrate that the results achieved by our technique using the same profiling information are almost identical for different inputs. This is shown in Figure 9. The figure shows the results only for the values of  $r$  of less than 0.3, because outside of this interval, the execution slowdown becomes excessive. Because of the very small variability of the execution time of task functions, the difference between results for different inputs is also small. The best result is achieved for the parameter value  $r = 0.28$ , which results in power savings of over 5.5% and a small negative slowdown (i.e. a small speedup) of the application execution.

#### 4.4 MPEG Decoder

We profile the application using a training set of seven MP3 files encoding different kinds of sound content. Using this profiling input, we apply our DVS technique to the training set and another set of seven input files. The results achieved by our technique, averaged over each input set, are shown in Figure 10. The power savings are almost identical for both sets, while the execution slowdown is slightly greater for the second set. As in the case of JPEG, the results achieved for the training input are reproducible with a different input set. The optimal value of the parameter  $r$  is

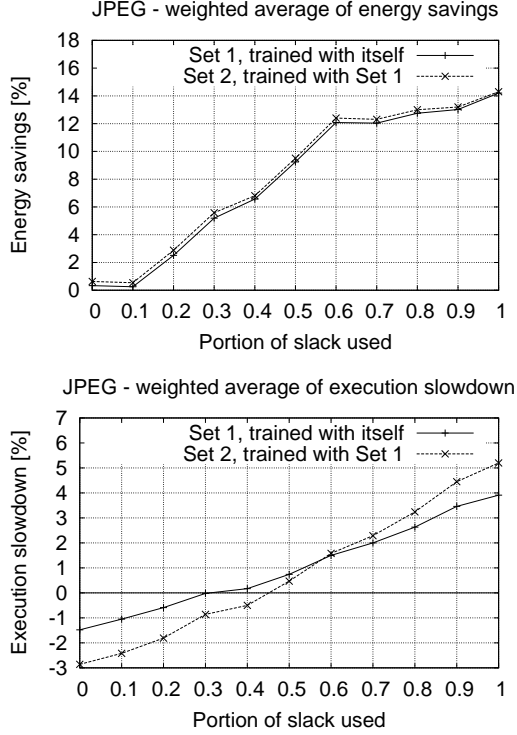


Figure 7: Evaluation results for the JPEG encoder

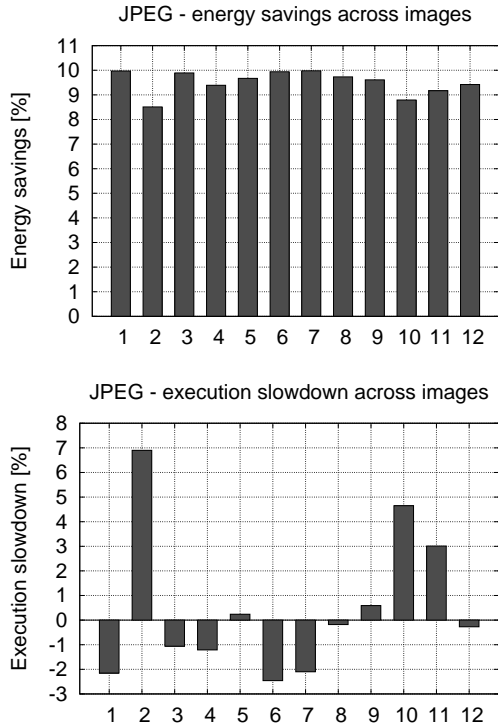


Figure 8: Breakdown of JPEG evaluation results across images for  $r = 0.5$

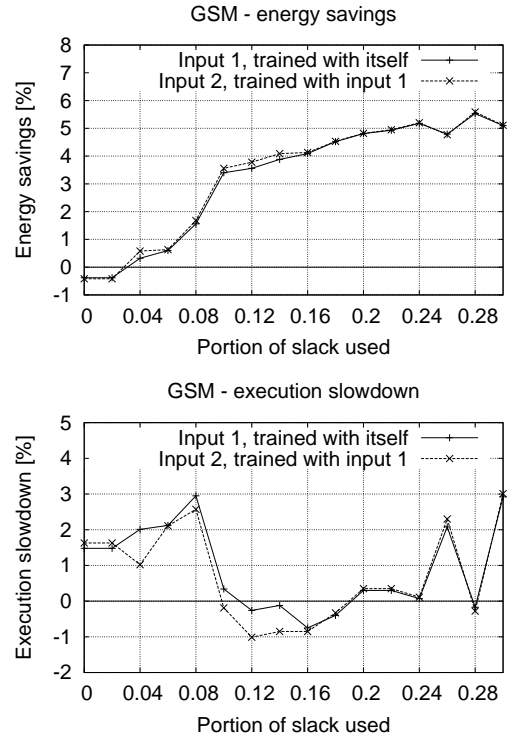


Figure 9: Evaluation results for the GSM encoder

0.6, which yields the power savings of 8.4% with the execution slowdown of approximately 1.5%. Figure 11 shows the breakdown of the results achieved with  $r = 0.6$  across the inputs from the second set. The power savings are consistent, with some variation in the execution slowdown.

#### 4.5 Summary of Results

The evaluation results indicate that our technique consistently achieves processor power savings across a set of realistic MLCA applications, without significant negative performance impact and thus achieves the goal set forward at the beginning of Section 3. Positive results are achieved despite the restrictive assumptions on which the theoretical derivation of our algorithms is based. In particular, variable execution times of task instructions is sufficiently well-approximated by averaging their execution time across a training set of inputs.

The impact on the application execution time shows more variability than the achieved power savings, but with a proper choice of the distributed fraction of the slack  $r$ , the average slowdown is small. With  $r = 0$ , no DVS takes place and we observe only the effect of our task scheduling algorithm. Its effect on the application execution time without voltage selection varies between applications, but it is not drastic for any of the applications.

Our technique has not shown to be computationally-intensive. Solving the ILP model for slack distribution is the only performance-critical operation, since it presents an NP-complete problem. For each benchmark application, Table 2 shows the number of task instructions in the target loop  $N_T$ , the number of critical and scaled task instructions, and the number of variables in the ILP model for slack distribution,

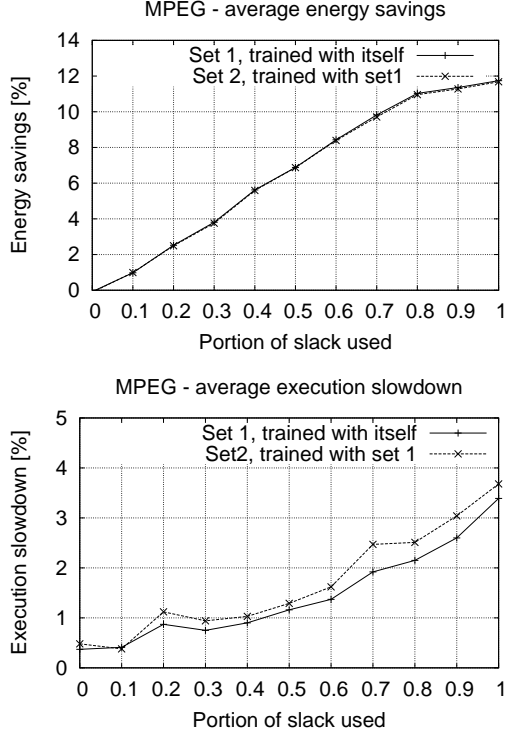


Figure 10: Evaluation results for the MPEG decoder

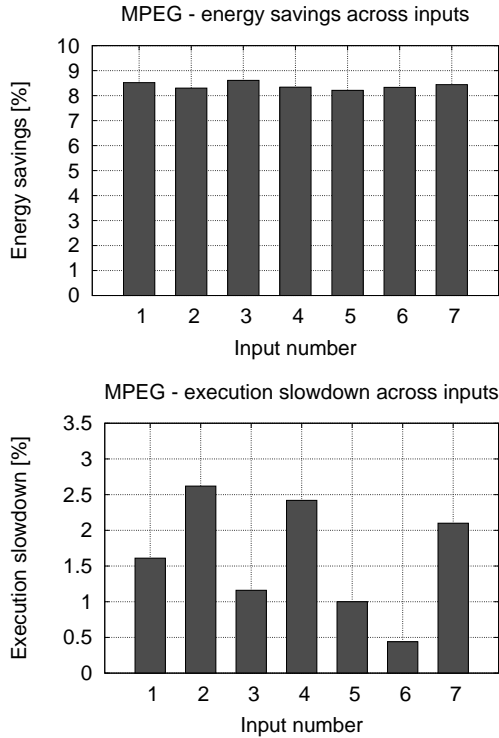


Figure 11: Breakdown of MPEG evaluation results across inputs for  $r = 0.6$

Table 2: Numbers of task instructions and variables in the ILP model

Application	$N_T$	$N_{critical}$	$N_{scaled}$	$N_{vars}$
JPEG	8	1	6	56
GSM	26	11	6	56
MPEG	27	1	23	209

which is proportional to the number of scaled task instructions. The number of tasks in the run-time task graph of the target loop is equal to  $N_T \cdot I$ , where  $I$  is the number of iterations of the target loop. For all three applications, this number is on the order of hundreds or thousands even for very small inputs. Clearly, the size of the run-time task graph is beyond capabilities of previously proposed DVS algorithms that attempt to analyze the task graph in its entirety. In contrast, our technique focuses on the loop dependence graph, which is smaller by several orders of magnitude. The data in Table 2 indicate that the number of variables in the ILP model for slack distribution is well within the capabilities of standard ILP solvers. In practice, the average solving time on a PC workstation using LP\_SOLVE [3] is less than 0.2 seconds for GSM and JPEG and 47 seconds for MPEG. Because of the short execution time, it is possible to determine a good value of  $r$  by repeated execution with different values of  $r$ .

In order to further evaluate the achieved results, we conducted experiments whose purpose was to determine a practical upper bound on the achievable power savings for the MLCA applications. We implemented several task scheduling algorithms and attempted to find the optimal voltage selections for the resulting task schedules using an integer linear programming model similar to those proposed in [1, 24]. The results, which we omit for brevity, indicate that the technique proposed in this paper succeeds in capturing most of the potential for power optimizations in these applications. A discussion of these results can be found in [15].

Furthermore, we conducted experiments in order to compare our technique with methods based on partitioning the application task graph into a periodic series of subgraphs small enough to be analyzed using computationally-intensive means such as ILP-based voltage selection algorithms. In these experiments, we construct an idealized run-time task graph under the assumption of constant execution time of task functions. We schedule this idealized task graph and select an interval in the task schedule that encloses a single loop iteration  $i$  (plus tasks from other iterations executed within the same interval). For this interval, we compute the optimal voltage selection using an ILP model similar to those proposed in [1, 24] and assign the voltage levels computed for the tasks from iteration  $i$  to the corresponding task functions. The results, which we omit for brevity, indicate that our method is superior both in result quality and computational intensity. The method based on interval selection produces ILP problems with numbers of variables greater by an order of magnitude, in some cases too large to be solved in practice. Voltage selection solutions were inferior to those produced by our technique, resulting in smaller energy savings. Furthermore, without a task scheduling algorithm based on prioritizing the critical tasks, excessive slowdown is incurred. A discussion of these results can be found in [15].

## 5. RELATED WORK

Jha [10] presents a survey of the early research work in the area of DVS techniques for real-time systems, which was mostly aimed at single-processor systems executing sets of independent tasks. Subsequent research has addressed the more general problems of DVS techniques for a multi-processor real-time system that periodically executes a set of dependent tasks. Two basic approaches to this problem have emerged. The first approach [25] is aimed at exploiting the slack that arises at run-time when the execution time of certain tasks happens to be shorter than the worst case. The second approach [1, 7, 20, 24] assumes that the execution time of each task is known at design-time and tries to exploit the slack in the task graph using static algorithms for task scheduling and voltage selection. Some authors [18] have attempted to combine these two approaches. Most of the heuristic algorithms for the static task scheduling and voltage selection either use guided random search techniques [20] or combine ILP formulations of the voltage selection problem with traditional task scheduling algorithms [24]. Novel heuristics have also been proposed [7, 18]. Some authors [1] ignore the issue of task scheduling and focus on finding the optimal voltage selection for the given task schedule. The cited works are mostly focused on deriving the algorithm for voltage selection, treating the task scheduling scheme as given. Our approach takes a different path, focusing on the voltage selection first, and then deriving an appropriate task scheduling scheme based on the notions defined in the context of our voltage selection algorithm.

The DVS algorithms for real-time systems executing periodic task graphs are somewhat similar to our technique, since the repeated execution of a set of dependent tasks with a fixed deadline can be compared with the repeated execution of the set of task functions from the loop in an MLCA application. However, the crucial difference is that the parallelism in the former case is limited to a single instance of the periodic task graph, which is assumed to be small enough to be analyzed using computationally-intensive means such as ILP models or genetic algorithms. Loops in MLCA programs are characterized by cross-iteration dependences and pipelining the execution of loop iterations at run-time, which effectively precludes reducing the power optimization problem to a task graph of a single loop iteration. Partitioning of the task graph is possible only by computing the run-time task schedule of an approximation of the task graph and computing the voltage selection for a selected interval in the manner described in Section 4.5. However, this approach is significantly more computationally intensive than our technique and produces inferior solutions.

Several authors have proposed more general approaches to the power optimizations for real-time systems. Varatkar and Marculescu [22] propose a DVS technique that accounts for the energy overheads of inter-task communication. Wu et al. [23] introduce a DVS algorithm for real-time systems executing conditional task graphs, which capture both data and control dependences in a set of tasks.

In the area of multitasking operating systems, dynamic voltage scaling can be used to reduce the processor power consumption during the time intervals when the average workload is low. Lorch and Smith [14] present an overview of research in this area. In the context of optimizing compilers for general-purpose applications, there have been attempts at compile-time power DVS optimizations targeting

the program regions characterized by excessive numbers of processor stall cycles. Several authors have also studied the influence of traditional compiler optimizations on the power characteristics of the program. A comprehensive overview of the research in the area of power optimization in the context of compilers and operating systems is presented in [8]. Unlike our work, research in this area is primarily aimed at single-processor, general-purpose computer systems.

The models used to determine the relations between the processor voltage and frequency, power consumption, and program performance vary among cited works. Most authors use analytical models to determine the relations between processor voltage, frequency, and power. These analytical power models range from simple formulas for the dynamic power consumption [18, 24] to more sophisticated models that take into account effects such as the leakage power [1]. Instead of modeling the processor characteristics analytically, we have used the figures characteristic of the Intel XScale processor reported in [5].

Most of the authors in the area of DVS power optimizations ignore the performance impact of the transitions between voltage levels [7, 18, 20, 22, 24, 25]. Notable exceptions are [1, 16, 19]. Our integer linear programming formulation for the slack distribution is closely modeled on the ILP formulations for voltage selection from [1, 19, 24].

The majority of the cited authors evaluate the proposed algorithms using randomly generated artificial task graphs [7, 20, 24], or a combination of artificial task graphs and only one task graph of a realistic application [1, 16, 18, 23]. In contrast, we evaluate our technique using exclusively task graphs pertaining to realistic multimedia applications.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel DVS technique for power optimizations of multimedia applications running on the MLCA. Our technique consists of profile-based heuristic compiler algorithms for voltage selection and task scheduling. The algorithms use loop dependence analysis, and take advantage of control-flow regularities that emerge across a wide range of MLCA multimedia applications. Previous work in the area of DVS optimizations for parallel systems has focused on applications in form of task graphs small enough to be analyzed in their entirety or periodic task graphs whose analysis can be reduced to the analysis of a single period. In contrast, our technique handles arbitrarily large task graphs, generated by the execution of loops that feature cross-iteration dependences and parallelism achieved by pipelining loop iterations.

We evaluated the proposed technique on three realistic applications, using profiling data from an MLCA simulator. We demonstrated that power savings in the range of 5-10% can be achieved with very small performance penalties. Although the behavior of applications is not consistent with the assumption that the execution times of task instructions are invariable, using the execution times averaged across a training set of inputs shows to be a sufficiently good approximation.

Although our technique was developed in the context of the MLCA architecture, we believe that it is also applicable in the more general context of task-level parallelism in multimedia applications. The principal source of task-level parallelism in multimedia applications is the pipelining of the computations performed by the tasks in the main loop.

Regardless of the parallel execution model and the underlying architecture, our analysis of the task graph could be applied to a broad variety of such applications.

In the future, we hope to evaluate our technique on an extended set of benchmark MLCA applications. We also hope to generalize the technique to more complex control-flow structures in the control programs. Furthermore, we hope to enhance our technique to exploit further opportunities for power savings that arise in applications with variable run-time characteristics, such as bursty behavior. Finally, we intend to test our technique on real hardware once the physical implementation of a DVS-enabled MLCA system is available.

## 7. ACKNOWLEDGMENTS

This work is supported by research grants from STMicroelectronics and Communications Information Technology Ontario (CITO).

## 8. REFERENCES

- [1] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. M. Al-Hashimi. Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 518–523, 2004.
- [2] U. Aydonat. Compiler support for a system-on-chip multimedia architecture. Master’s thesis, University of Toronto, 2005.
- [3] M. Berkelaar, K. Eikland, and P. Notebaert. *Open Source Mixed Integer Linear Programming System*, 2004. [http://groups.yahoo.com/group/lp\\_solve](http://groups.yahoo.com/group/lp_solve).
- [4] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Jour. of Solid-State Circuits*, 35(11):1571–1580, 2000.
- [5] L. T. Clark, E. J. Hoffman, J. Miller, M. Biyani, Y. Liao, S. Strazdus, M. Morrow, K. E. Velarde, and M. A. Yarch. An embedded 32-b microprocessor core for low-power and high-performance applications. *IEEE Jour. of Solid-State Circuits*, 36(11):1599–1608, 2001.
- [6] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman & Co., 1979.
- [7] F. Gruian and K. Kuchcinski. LEneS: Task scheduling for low-energy systems using variable supply voltage processors. In *Proc. of the Conf. on Asia South Pacific Design Automation*, pages 449–455, 2001.
- [8] C.-H. Hsu. *Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction*. PhD thesis, Rutgers University, 2003.
- [9] Intel Corporation. *Intel 80200 Processor Based on Intel XScale Microarchitecture*, 2003. <http://www.intel.com/design/iio/manuals/273411.htm>.
- [10] N. K. Jha. Low power system scheduling and synthesis. In *Proc. of the IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 259–263, 2001.
- [11] F. Karim, A. Mellan, A. Nguyen, U. Aydonat, and T. S. Abdelrahman. A multi-level computing architecture for embedded multimedia applications. *IEEE Micro*, 24(3):55–56, 2004.
- [12] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
- [13] K. Lahiri, S. Dey, D. Panigrahi, and A. Raghunathan. Battery-driven system design: A new frontier in low power design. In *Asia South Pacific Design Automation/VLSI Design*, page 261, 2002.
- [14] J. R. Lorch and A. J. Smith. Operating system modifications for task-based speed and voltage scheduling. In *Proc. of the Intl. Conf. on Mobile Systems, Applications, and Services*, pages 259–263, 2003.
- [15] I. Matosevic. Power optimizations for the MLCA using dynamic voltage scaling. Master’s thesis, University of Toronto. In preparation.
- [16] B. Mochocki, X. S. Hu, and G. Quan. A realistic variable voltage scheduling model for real-time applications. In *Proc. of the IEEE/ACM Intl. Conf. on Computer-Aided Design*, pages 726–731, 2002.
- [17] K. J. Nowka, G. D. Carpenter, and B. C. Brock. The design and application of the PowerPC 405LP energy-efficient system-on-a-chip. *IBM Jour. of Research and Development*, 47(5–6):631–639, 2003.
- [18] D. Roychowdhury, I. Koren, and C. M. Krishna. A voltage scheduling heuristic for real-time task graphs. In *Proc. of the Intl. Conf. on Dependable Systems and Networks*, pages 741–750, 2003.
- [19] H. Saputra, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C.-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Proc. of the Joint Conf. on Languages, Compilers and Tools for Embedded Systems*, pages 2–11, 2002.
- [20] M. T. Schmitz, B. Al-Hashimi, and P. Eles. Energy-efficient mapping and scheduling for DVS enabled distributed embedded systems. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 514–521, 2002.
- [21] Transmeta Corporation. *Crusoe processor technology*. <http://www.transmeta.com/crusoe>.
- [22] G. Varatkar and R. Marculescu. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 510–517, 2003.
- [23] D. Wu, B. M. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graphs for the synthesis of low power embedded systems. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 90–95, 2003.
- [24] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. In *Proc. of the Conf. on Design Automation*, pages 183–188, 2002.
- [25] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):686–700, 2003.