Overhead-Aware Energy Optimization for Real-Time Streaming Applications on Multiprocessor System-on-Chip

YI WANG, Hong Kong Polytechnic University
HUI LIU, Xidian University
DUO LIU, ZHIWEI QIN, and ZILI SHAO, Hong Kong Polytechnic University
EDWIN H.-M. SHA, Hunan University and The University of Texas at Dallas

14

In this article, we focus on solving the energy optimization problem for real-time streaming applications on multiprocessor System-on-Chip by combining task-level coarse-grained software pipelining with DVS (Dynamic Voltage Scaling) and DPM (Dynamic Power Management) considering transition overhead, inter-core communication and discrete voltage levels. We propose a two-phase approach to solve the problem. In the first phase, we propose a coarse-grained task parallelization algorithm called RDAG to transform a periodic dependent task graph into a set of independent tasks by exploiting the periodic feature of streaming applications. In the second phase, we propose a scheduling algorithm, GeneS, to optimize energy consumption. GeneS is a genetic algorithm that can search and find the best schedule within the solution space generated by gene evolution. We conduct experiments with a set of benchmarks from E3S and TGFF. The experimental results show that our approach can achieve a 24.4% reduction in energy consumption on average compared with the previous work.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Interconnection architectures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Design, Performance, Algorithms

Additional Key Words and Phrases: Real-time, task scheduling, energy optimization, streaming applications, MPSoC, overhead-aware

ACM Reference Format:
Wang, Y., Liu, H., Liu, D., Qin, Z., Shao, Z., and Sha, E. H.-M. 2011. Overhead-aware energy optimization for real-time streaming applications on multiprocessor system-on-chip. ACM Trans. Des. Autom. Electron. Syst. 16, 2, Article 14 (March 2011), 32 pages.
DOI = 10.1145/1929943.1929946 http://doi.acm.org/10.1145/1929943.1929946

A preliminary version of this work appears in *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)* [Liu et al. 2008].
This work is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF PolyU 5260/07E and GRF PolyU 5269/08E), HK PolyU (A-PJ17), the National Science Foundation of China. (No. 608031520), the Key Research Project of the Ministry of Education, China (No. 2009-144), and the Fundamental Research Fund for the Central Universities, China.
Authors' addresses: Y. Wang, D. Liu, Z. Qin, and Z. Shao, Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong; H. Liu, Software Engineering Institute, Xidian University, XI'AN, China; E. H.-M. Sha, Hunan University, Changsha, China 410082 and the Department of Computer Science, the University of Texas at Dallas, Richardson, TX 75083. Z. Shao (corresponding author) email: cszlshao@comp.polyu.edu.hk.

DOI 10.1145/1929943.1929946 http://doi.acm.org/10.1145/1929943.1929946

1. INTRODUCTION

With increasing demand for high-performance multimedia in battery-driven mobile devices, multicore architecture such as MPSoC (Multiprocessor System-on-Chip) is becoming widely adopted in embedded systems. The examples include TI TMS320DM6467 DaVinci processors, Freescale MSC8122 and MSC8126 multicore DSP processors, ARM ARM11 MPCore and Intel Atom processors. Some multicore processors such as ARM ARM11 MPCore and Intel Atom processors provide multiple voltage levels for low power optimization. When real-time streaming applications such as Internet video conferences and surveillance digital video recorders are executed on such chip multiprocessors, both time performance and energy consumption need to be considered as energy consumption is one of the most important performance metrics in embedded systems. Therefore, it becomes an important research problem to optimize energy consumption for streaming applications on MPSoCs.

To solve this problem, several issues need to be taken into account. First, a streaming application can be modeled as periodic dependent tasks in real-time systems, in which a stream of data is treated as a sequence of requests that are serviced by the streaming application when arrived [Xu et al. 2007]. In this article, the periodic behavior of dependent tasks is explored with task-level software pipelining [Chao and Sha 1993; Chao and LaPaugh 1993; Passos and Sha 1996]. Second, both dynamic power management (DPM) and dynamic voltage scaling (DVS) should be applied for energy optimization. DPM exploits idle times of a processor and turns off power supply so as to reduce static energy caused by leakage power [Jha 2001]. DVS reduces energy consumption by adjusting supply voltages of processors [Jha 2001]. Here, we assume that DPM and DVS can be applied for each processor core independently. As a chip multiprocessor may contain many processor cores, we need to consider the trade-off between adjusting voltages by DVS and turning off by DPM in energy optimization. Third, various practical issues including transition overhead caused by mode/voltage changes in DPM/DVS, discrete voltage levels and intercore communication should be considered for a practical solution. Taking all these issues into consideration, in the article, we focus on energy consumption optimization for streaming applications by combining task-level coarse-grained software pipelining with DVS and DPM techniques.

DVS is one of the most effective techniques for energy optimization. Therefore, a lot of DVS scheduling techniques have been proposed in previous work. For periodic independent tasks, the DVS scheduling has been extensively studied for single and multiple processors. For single processor, Aydin et al. [2001] showed that for any periodic task, it is optimal for all of its task instances to run at the same processor speed on an ideal DVS processor. Jejurikar and Gupta [2004] considered periodic tasks on a processor with discrete speed levels. Several studies have been conducted in the DVS scheduling on single processor based on dynamic priority [Aydin et al. 2001; Chen et al. 2005; Mejia-Alvarez et al. 2004] or fixed priority [Bini et al. 2005; Shin et al. 2001; Saewong and Rajkumar 2003]. For multiple processors, several studies [AlEnawy and Aydin 2005; Aydin and Yang 2003; Chen and Kuo 2005] focused on DVS scheduling on homogeneous multiple processors while other work [Hung et al. 2006; Yu and Prasanna 2002; Luo and Jha 2007, 2000] focused on heterogeneous multiple processors. Recent work [Zhong and Xu 2007; Aydin et al. 2006] studied system-wide energy minimization for periodic and aperiodic tasks on a processor with continuous speed levels. They separate task execution into on-chip/off-chip cycles, which are applicable for both CPU and memory. Niu and Quan [2006] proposed an approach for system-wide dynamic power management for multimedia portable devices. In all of the above work, the task model is based on *periodic independent tasks* at process or thread level. In this article, we consider *periodic dependent tasks* which can better model stream-based applications such as MPEG-4 AVC decoder [Wiegand et al. 2003].

There have been a lot of studies of DVS scheduling for dependent tasks on multiprocessor systems with multiple voltage levels. Hua and Qu [2005] studied the voltage setup problem and proposed an approach to select optimal voltage levels. Gruian and Kuchcinski [2001] introduced a scheduling approach. In their approach, based on a given fixed task assignment, the delays of all tasks are scaled down by the ratio of the timing constraint over the critical path length. Luo and Jha [2000] proposed an approach to evenly distribute slacks based on a fixed task scheduling. Zhang et al. [2002] proposed a framework that integrates task scheduling and voltage selection together to minimize the energy consumption for dependent tasks on multiple processors. However, these works focus on *dependent* task model instead of *periodic dependent* task model. With the dependent task model, only intra-iteration data dependencies are considered. In our work, we further exploit inter-iteration data dependencies by utilizing the periodic characteristics of the periodic dependent task model.

Several recent studies have explored the periodic behavior of periodic tasks with pipelining and parallel processing [Kim et al. 2005; Shao et al. 2007; Li and Martínez 2005]. In Kim et al. [2005], a power reduction technique is proposed to optimize energy by exploring pipelining and parallel processing in uniprocessor systems. This uniprocessor-based technique cannot be directly applied to solve our multiprocessor-based problem. In Shao et al. [2007], a loop scheduling technique is proposed to minimize energy by exploring inter-iteration dependencies for applications with loops on multi-core systems. The given technique, however, is based on loop optimization with instruction-level parallelism; thus, it is not applicable to the periodic task model. In Li and Martínez [2005], an analytical model is developed to study the power-performance issues of running parallel applications on chip multiprocessors. The proposed technique shows that, parallel computing can bring significant power-performance benefits over uniprocessor systems.

Our work is closely related to the previous work [Xu et al. 2007; Zhang et al. 2002; Acharya and Mahapatra 2008; Kianzad et al. 2005; Bambha and Bhattacharyya 2000; Liu et al. 2009; Wang et al. 2010], in which periodic dependent tasks are modeled by task graphs (Directed Acyclic Graph (DAG)). In Xu et al. [2007], an energy-aware scheduling technique is proposed to minimize energy consumption while satisfying both throughput and response time with pipelining. In Zhang et al. [2002], an energy optimization framework is proposed to integrate task scheduling and voltage selection together. In Acharya and Mahapatra [2008], a technique is developed to utilize slacks based on service rate and change in intervals for static and dynamic scheduling schemes, and a fault-tolerant scheme is incorporated into the slack management technique to implement reliable systems. In Kianzad et al. [2005], an integrated framework combining task assignment, scheduling, and power management using genetic algorithm is proposed. In Bambha and Bhattacharyya [2000], a periodic graph model is explored to effectively select voltage levels of iterative applications on multiprocessor systems. In the above work, the intra-iteration precedence relations of a task graph are not changed, which limits the optimization for both performance and energy. In Liu et al. [2009], coarse-grained software pipelining is applied to solve real-time streaming applications on MPSoC, and a DVS scheduling technique is proposed to optimize energy based on it. However, the technique is built on the assumption that there are no inter-core communication overhead and transition overhead with mode/voltage changes in DVS. In Wang et al. [2010], a task scheduling technique that changes the data dependency relations across different periods is proposed to effectively remove intercore communication overhead. However, it does not consider several overheads (i.e., transition overhead, sleep overhead). In this work, we consider various practical issues and propose a genetic algorithm to solve the problem.

In this article, we propose a two-phase approach to solve the energy minimization problem for periodic dependent tasks on MPSoC architectures considering various practical issues. In our approach, we first completely remove the precedence relations of tasks based on task-level software pipelining, and then perform energy optimization. Our two-phase approach is summarized as follows.

—In the first phase, we propose a coarse-grained task-level software pipelining algorithm called RDAG to transform periodic dependent tasks into a set of independent tasks based on the retiming technique [Leiserson and Saxe 1991]. In RDAG, we regroup tasks and put tasks from different periods into one loop kernel so as to completely remove precedence relations. In this way, abundant idle slacks incurred by precedence relations among tasks can be utilized. In addition, after transforming a dependent task graph into a set of independent tasks, more opportunities are provided to do scheduling with energy optimization by simultaneously considering multiple factors such as dynamic/static power, sleep/voltage transition overheads, and inter-core communication.
—In the second phase, we propose a scheduling algorithm, GeneS, to optimize energy consumption based on the results obtained in the first phase. GeneS is a genetic algorithm, and it can search and find the best schedule within the solution space generated by gene evolution.

To the best of our knowledge, this is the first work to solve the energy optimization problem for periodic dependent tasks on MPSoCs by combining task-level software pipelining with DVS and DPM considering various practical issues.

We conduct experiments on a set of benchmarks from Embedded Systems Synthesis Benchmarks Suite (E3S) [Vallerio and Jha 2003] and TGFF [Dick et al. 1998]. The benchmarks from E3S consist of various multimedia applications such as JPEG compression/decompression, RGB to CYMK conversion, RGB to YIQ conversion, and FFT/IFFT. TGFF is used to generate several synthetic task graphs. We compare our technique with the approach in Zhang et al. [2002] that applied DVS and DPM but without software pipelining. The experimental results show that our technique can achieve better results compared with the previous work. On average, our GeneS algorithm can achieve a 24.4% reduction in energy consumption compared with the approach in [Zhang et al. 2002]. For systems with tight timing constraints, our approach can obtain a feasible solution while the approach in Zhang et al. [2002] cannot.

The remainder of this article is organized as follows. Section 2 describes models and defines the problem. Section 3 gives a motivational example. Section 4 analyzes the lower bound of the energy consumption. Our two-phase approach is presented in Section 5. Experimental results are provided in Section 6. The conclusion is presented in Section 7.

2. SYSTEM MODEL AND PROBLEM STATEMENT

In this section, we first introduce system architecture, application model and some basic concepts that will be used in the later sections and then define the problem.

2.1. System Architecture

In this article, we employ an MPSoC architecture shown in Figure 1. The architecture consists of M processor cores $\{PE_1, PE_2, \ldots, PE_M\}$, and each processor core has its own data and program memory. The programmable bus controller implements a predefined bus protocol and assigns bus access rights to individual cores. If a task needs to read data that are not available in its local memory, inter-core communication happens.

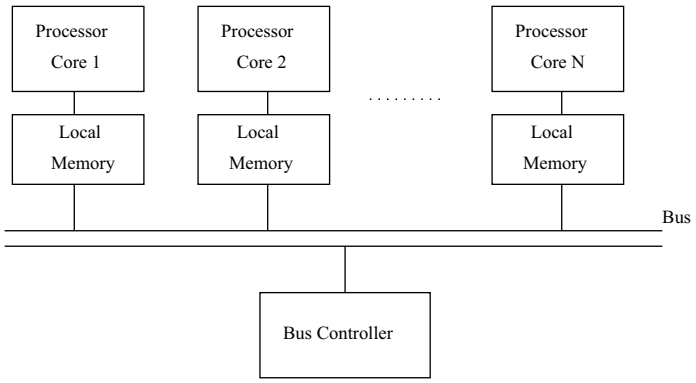


Fig. 1. A multiprocessor SoC architecture.

2.2. Task Graph and Static Schedule

Streaming applications are modeled as periodic dependent tasks. We use Directed Acyclic Graph (DAG) to represent periodic dependent tasks. DAG is a special case of Data Flow Graph (DFG). A DFG, $G = (V, E, \rho, C)$, is a node-weighted and edge-weighted directed graph. $V = \{T_1, T_2, \ldots T_n\}$ is the node set, and each node denotes a periodic task. $C(T_i)$ is the number of clock cycles to compute task T_i ($T_i \in V$). E is the edge set to represent data dependency among task nodes. An edge $(T_i, T_j) \in E$ represents that the data generated by T_i is needed in order to compute T_j, and $com(T_i, T_j)$ is used to represent the data volume associated with tasks T_i and T_j. $\rho(T_i, T_j)$ is a function to represent the number of delays for an edge $(T_i, T_j) \in E$. The edge without delay represents the intra-iteration data dependency, which means the dependencies inside one period, while the edge with delays represents the inter-iteration data dependency, which means the dependencies among different periods. The number of delays represents the number of periods involved. For an edge $(T_i, T_j) \in E$, initially $\rho(T_i, T_j) = 0$; later it may be changed as we group tasks from different periods into one period in our technique.

A *static* schedule of a given DFG is a repeated pattern of an execution of the corresponding periodic dependent tasks. In other words, a static schedule is used to represent the execution of *one period* of periodic dependent tasks. In this article, the execution of one period is called *one iteration* as well. A schedule implies both schedule step assignment and processor core allocation. A static schedule must obey the dependency relations of the DAG portion of the DFG. The DAG is obtained by removing all edges with delays in the DFG.

2.3. Power Model

In this article, a processor core in an MPSoC can support both DPM (Dynamic Power Management) and DVS (Dynamic Voltage Scaling). A processor core can operate at k different voltage/frequency levels, $\{(V_{dd_1}, f_1), (V_{dd_2}, f_2), \ldots, (V_{dd_k}, f_k)\}$, in which it consumes both dynamic power and static power. Without loss of generality, we assume that the voltage levels from V_{dd_1} to V_{dd_k} are in ascending order, in which V_{dd_1} is the lowest voltage level and V_{dd_k} is the highest voltage level. The voltage level of a processor core can be changed independently by voltage-level-setting instructions without influencing other cores. A processor core has one sleep mode as well in which it is deactivated and dissipates reduced power.

Given a static schedule S, its total energy consumption, $E_{total}(S)$, can be represented as follows:

$$E_{total}(S) = E_{t\_dynamic}(S) + E_{t\_static}(S) + E_{t\_sleep}(S) + E_{t\_sleepOH}(S) + E_{t\_tranOH}(S) + E_{t\_comm}(S). \quad (1)$$

Here, $E_{t\_dynamic}(S)$ is the total dynamic energy consumption, $E_{t\_static}(S)$ is the total static energy consumption, $E_{t\_sleep}(S)$ is the total energy consumption in the sleep mode, $E_{t\_sleepOH}(S)$ is the total transition energy overhead associated with the sleep mode, $E_{t\_tranOH}(S)$ is the total transition energy overhead caused by voltage changes, and $E_{t\_comm}(S)$ is the total energy overhead by inter-core communication. Next, we introduce how to calculate them one by one.

The dynamic power consumption of a processor core at a voltage level V_{dd} is calculated based on the power model in Rabaey et al. [2002]:

$$P_{dynamic}(V_{dd}) = C_{SW} \cdot f_{op} \cdot V_{dd}^2, \quad (2)$$

where C_{SW} is the capacitance, and f_{op} is the frequency of a processor core at voltage level V_{dd}. Then, the dynamic energy consumption of a task T_i running at voltage level V_{dd} is

$$E_{dynamic}(T_i, V_{dd}) = P_{dynamic}(V_{dd}) \cdot \frac{C(T_i)}{f_{op}} = C(T_i) \cdot C_{SW} \cdot V_{dd}^2, \quad (3)$$

where $C(T_i)$ is the number of cycles of task T_i.

Different leakage sources contribute to the static power consumption in a processor core. The major contributors are the subthreshold leakage current and the reverse bias junction current. Based on the model in Martin et al. [2002], the static power consumption, P_{static}, can be expressed as

$$P_{static}(V_{dd}) = I_{subn} \cdot V_{dd} + |V_{bs}| \cdot I_j, \quad (4)$$

where I_{subn} is the subthreshold current, V_{bs} is the body bias voltage, and I_j is the reverse bias junction current.

For the energy consumed in the sleep mode, let t_{sleep} be the time duration in which a processor core is in the sleep mode and let P_{sleep} be the corresponding power consumption. Then the energy consumed in the sleep mode, E_{sleep}, is calculated by

$$E_{sleep} = P_{sleep} \cdot t_{sleep}. \quad (5)$$

It takes both time and energy for a processor core to enter into and exit from the sleep mode. Let $t_{sleepOH}$ be the time transition overhead and $E_{sleepOH}$ be the energy transition overhead associated with one transition for entering into and exiting from the sleep mode. The total energy transition overhead can be obtained by the product of $P_{sleepOH}$ and the transition time.

Besides dynamic power and static power, we need to consider both time and energy overheads during voltage transitions. According to the power model in Burd [2001] and Mochocki et al. [2004], for a voltage change from V_{dd_i} to V_{dd_j}, the time transition t_{TRAN} can be calculated by

$$t_{TRAN} = \frac{2 \cdot C_{DD}}{I_{MAX}} \cdot |V_{dd_j} - V_{dd_i}|, \quad (6)$$

where C_{DD} is the capacitance of the voltage converter, and I_{MAX} is the maximum output current of the converter.

The energy transition overhead E_{tranOH} includes the energy consumed by the voltage converter, $E_{TRAN-DC}$, and the energy consumed by a processor core during the

transition, $E_{TRAN-CPU}$, so

$$E_{tranOH} = E_{TRAN-DC} + E_{TRAN-CPU}. \tag{7}$$

For a voltage change from V_{dd_i} to V_{dd_j}, $E_{TRAN-DC}$ and $E_{TRAN-CPU}$ are calculated by

$$E_{TRAN-DC} = \alpha \cdot C_{DD} \cdot |V_{dd_i}^2 - V_{dd_j}^2| \tag{8}$$

$$E_{TRAN-CPU} = P_{TRAN} \cdot t_{TRAN}. \tag{9}$$

Here, α is the efficiency factor of the voltage converter, P_{TRAN} is the power consumption at the voltage level entered in the transition.

The energy consumed by intercore communication between task T_i and task T_j is calculated by

$$E_{comm}(T_i, T_j) = P_{comm} \cdot \frac{com(T_i, T_j)}{B}, \tag{10}$$

where P_{comm} is the power consumption of the shared bus in one clock cycle, $com(T_i, T_j)$ is the data volume transferred between tasks T_i and T_j, and B is the bus bandwidth.

From the above, we can obtain the total energy consumption of a schedule by adding all components together.

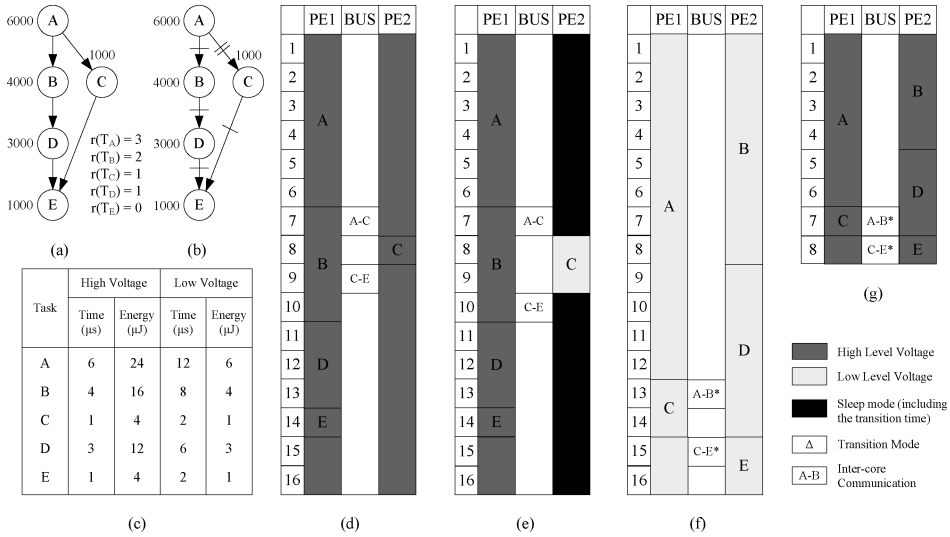
2.4. Retiming

Retiming is originally proposed to minimize the cycle period of a synchronous circuit by evenly distributing registers [Leiserson and Saxe 1991]. It has been extended to schedule data flow graphs on parallel systems [Chao and Sha 1993; Chao and LaPaugh 1993; Passos and Sha 1996]. In this article, we generate a new loop kernel by regrouping tasks from different periods so as to remove intra-iteration dependencies. We use retiming to model this regrouping.

Given a DFG $G = (V, E, \rho, C)$, *a retiming r of G* is a function that maps each node T_i in V to an integer $r(T_i)$. Basically, by retiming a task node in a DFG once, a delay is drawn from *each* of its incoming edges, and then pushed to *each* of its outgoing edges. Every retiming operation corresponds to a software pipelining operation, and as shown in Section 5.1, retiming a node once means one copy of this task is moved into the prologue. From the program point of view, the retiming technique regroups a loop body and attempts to remove intra-iteration dependencies among nodes. The transformed loop body after the retiming can be obtained based on the retiming values of nodes [Chao and LaPaugh 1993]. The delay count of an edge (T_i, T_j) $((T_i, T_j) \in E)$ after retiming, $\rho_r(T_i, T_j)$, is named the retimed delay count, and can be calculated by $\rho_r(T_i, T_j) = r(T_i) - r(T_j)$ [Leiserson and Saxe 1991]. An edge $(T_i, T_j) \in E$ with delay count $\rho(T_i, T_j) > 0$ means that the computation of node T_j at the ℓth iteration requires data produced by node T_i at the $\ell - \rho(T_i, T_j)$ th iteration. A retiming function r is *legal* if the retimed delay counts of all edges in the retimed graph G_r are nonnegative. An illegal retiming function occurs when the retimed delay count of one edge becomes negative, and this situation implies a reference to nonavailable data from a future period.

2.5. Genetic Algorithm

Genetic Algorithm (GA) is an iterative procedure to simulate evolution for a population of candidate solutions to the optimization problem. In this article, we adopt genetic algorithms to solve our energy optimization problem. In a genetic algorithm, each iteration step is called a *generation*, and each candidate solution is called a *chromosome* that consists of several pairs of *genes* [Mitchell 1996]. A genetic algorithm begins with an initial population of chromosomes. Two genetic reproduction operators, *mutation*



Task	High Voltage		Low Voltage	
	Time (μs)	Energy (μJ)	Time (μs)	Energy (μJ)
A	6	24	12	6
B	4	16	8	4
C	1	4	2	1
D	3	12	6	3
E	1	4	2	1

(c)

Fig. 2. (a) The original DFG. (b) The retimed DFG using our RDAG algorithm. (c) The task information. (d) The schedule generated by the list scheduling in Landskov et al. [1980] without power management (the energy is $137\mu J$). (e) The schedule generated by the DAG-based scheduling algorithm in Zhang et al. [2002] with DVS and DPM (the energy is $73.4\mu J$). (f) The schedule generated by our technique (the energy is $25\mu J$). (g) The schedule generated by our technique with a tight timing constraint ($8\mu s$).

and *crossover* (recombination), are designed to create new chromosomes for the next generation. The number of chromosomes in each generation is constant. Thus, a fitness function is used to evaluate each chromosome, and only those chromosomes with higher fitness value will be selected to form the new population of the next generation. This evolution process is repeated until a termination condition has been reached. The chromosome with the highest fitness value in the last generation is selected as the final solution [Alba and Troya 1999].

2.6. Problem Statement

For a DFG used to model given dependent periodic tasks, the overhead-aware energy optimization problem is defined as follows:

Given a DFG $G = (V, E, \rho, C)$, a timing constraint TC, an MPSoC with M processor cores, $\{PE_1, PE_2, \ldots, PE_M\}$, and each processor core with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \ldots, V_{dd_k}\}$, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy consumption within the timing constraint TC, that is, for each task T_i ($T_i \in V$), find its assignment, its release time and its voltage level such that for the obtained static schedule S, the schedule length of S is less than or equal to TC and the total energy consumption of S, $E_{total}(S)$, is minimized.

3. MOTIVATIONAL EXAMPLE

In this section, we motivate the energy optimization problem by showing how to schedule a DFG. We compare energy consumption of the schedules generated by the list scheduling in Landskov et al. [1980], the algorithm in Zhang et al. [2002], and our technique.

Figure 2(a) shows the DFG that is used to model periodic dependent tasks. In the DFG, each node represents a task, and the number beside each node represents the number of clock cycles needed to execute the node. The edge between two nodes represents data dependency, and if two nodes of an edge are assigned to different

processor cores in a schedule, intercore communication will occur. For example, In the schedule shown in Figure 2(d), there are two intercore communications: $A - C$ and $C - E$.

We assume that there are two processor cores in this MPSoC, and each core has two voltage/frequency levels, the high level and the low level. Based on the dynamic power model in Equation (2), $P_{dynamic} = C_{SW} \cdot f_{op} \cdot V_{dd}^2$, without loss of generality, we assume that $C_{SW} = 1nF$; the voltage/frequency pair is $(2V, 1GHz)$ at the high level and $(1V, 0.5GHz)$ at the low level. Therefore, we get $P_H = 4W$, $P_L = 0.5W$, $CP_H = 1ns$, and $CP_L = 2ns$, where P_H and P_L are used to represent the high-level and low-level power consumptions, respectively, and CP_H and CP_L are the high-level and low-level clock periods, respectively. The number of clock cycles of a node is not changed with DVS. Thus, we get the execution time and energy consumption of each node in Figure 2(c), where the time unit is μs and the energy unit is μJ. For simplicity, we assume that the transition time overhead is $1\mu s$ for each voltage change, the transition energy overhead from the high to low voltage levels is approximately $4\mu J$, and one from the low to high voltage levels $0.5\mu J$. The time overhead and energy overhead to enter into and exit from the sleep mode are $5\mu s$ and $2\mu J$, respectively, and the sleep state power is $0.1W$. The read/write communication power through the communication bus is $0.5W$, and the communication time between two tasks is $1\mu s$. The static power is $0.25W$. These assumptions are only for demonstration purpose. Our technique is general enough to deal with general cases, as discussed in later sections.

Assume that the timing constraint is $16\mu s$. The first schedule shown in Figure 2(d) is obtained by the traditional list scheduling algorithm in Landskov et al. [1980] that focuses on optimizing time performance without power management. In Figure 2(d), we can see that both processor cores operate at the high voltage level for the best time performance. There are some idle slacks in the schedule; however, they cannot be utilized because of the data dependencies. Based on Equation (1), we can obtain the total energy of the schedule: $E_{total}(S) = E_{t\_dynamic}(S) + E_{t\_static}(S) + E_{t\_comm}(S) = P_H \times 32 + P_{static} \times 16 \times 2 + P_{comm} \times 2 \times 1 = 4 \times 32 + 0.25 \times 32 + 0.5 \times 2 = 137\mu J$.

The second schedule shown in Figure 2(e) is obtained by the DAG-based scheduling algorithm in Zhang et al. [2002] that applies DPM and DVS to minimize the energy consumption. From the schedule, we can see that on the first core, tasks T_A, T_B, T_D, and T_E are assigned the high voltage level due to the timing constraint. On the second core, task T_C is assigned the low voltage level with DVS, and the idle slacks are turned into the sleep mode with DPM. Based on Equation (1), we can obtain the total energy of the schedule: $E_{total}(S) = E_{t\_dynamic}(S) + E_{t\_static}(S) + E_{t\_comm}(S) + E_{t\_sleep}(S) + E_{t\_sleepOH}(S) = P_H \times 16 + P_L \times 2 + P_{static} \times (16+2) + P_{comm} \times 2 \times 1 + P_{sleep} \times (16-2-5) + E_{sleepOH} \times 1 = 4 \times 16 + 0.5 \times 2 + 0.25 \times 18 + 0.5 \times 2 + 0.1 \times 9 + 2 = 64 + 1 + 4.5 + 1 + 0.9 + 2 = 73.4\mu J$.

The schedule generated by our approach is shown in Figure 2(f). In our approach, we first use our RDAG algorithm (shown in Section 5.1) to transform all the intra-iteration data dependencies in Figure 2(a) into the inter-iteration data dependencies as shown in Figure 2(b). This step makes all tasks in one iteration be independent of each other. Next, we use our GeneS scheduling algorithm (shown in Section 5.2 to generate a task schedule shown in Figure 2(f). Because we adopt coarse-grained software pipelining, the slacks caused by the intra-iteration data dependencies are reclaimed. At the same time, because we can overlap communication and computation, the slacks caused by intercore communication can be reused as well (for the inter-core communication in Figure 2(f)–(g), the symbol $*$ represents the data dependence to the next period). In the schedule in Figure 2(f), as all idle slacks can be fully utilized, all the tasks can be executed at the low voltage level. Based on Equation (1), we can obtain the total energy of the schedule: $E_{total}(S) = E_{t\_dynamic}(S) + E_{t\_static}(S) + E_{t\_comm}(S) = P_L \times 32 + P_{static} \times 16 \times 2 + P_{comm} \times 2 \times 1 = 0.5 \times 32 + 0.25 \times 32 + 0.5 \times 2 = 16 + 8 + 1 = 25\mu J$.

If given a tight timing constraint smaller than $14\mu s$, the two DAG-based scheduling algorithms cannot obtain feasible solutions while our approach can. Figure 2(g) shows the schedule obtained by our technique with the timing constraint $8\mu s$.

From these results, we can see that our technique can effectively reduce energy consumption. Based on the schedule obtained by our approach, the tasks can be scheduled with the insertion of voltage-setting instructions. The technique can be integrated into compilers or real-time O.S. to generate energy-efficient code. Next, we will present the details of our approach.

4. LOWER-BOUND ANALYSIS

In this section, we conduct lower-bound analysis for the energy optimization problem defined in Section 2.6. In our two-phase approach as shown in Section 5, in the first phase, we transform intra-iteration data dependencies into inter-iteration dependencies so as to make all tasks independent of each other inside a period (Section 2.6). With this transformation, the problem defined is changed to: Given a set of independent tasks, a timing constraint TC, an MPSoC with M cores, $\{PE_1, PE_2, \ldots, PE_M\}$, and each core with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \ldots, V_{dd_k}\}$, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy consumption within the timing constraint TC. Next, we study the lower bound of the above problem.

The problem of finding an optimal task schedule with the minimum energy consumption for a single-core or multicore system is known to be NP-complete [El-Rewini et al. 1995; Hu and Marculescu 2004; Quan and Hu 2002; Liu et al. 2009]. For our problem with independent tasks on multicore systems, efficient algorithms to obtain lower bounds do not exist from the previous work. Therefore, we simplify our problem to be a single-core scheduling problem, and use its solution as the approximate lower bound of our problem. The simplified problem is defined as follows: Given a set of independent tasks, a single-core system with k discrete voltage levels, $\{V_{dd_1}, V_{dd_2}, \ldots, V_{dd_k}\}$, a timing constraint $M \cdot TC$, assuming that there is no transition overhead for voltage changes, find voltage assignment for each task and a static schedule such that the schedule has the minimum energy consumption within the timing constraint $M \cdot TC$.

In the simplified problem, we put all available time slots from all cores to one core ($M \cdot TC$) and assume an ideal case without transition overhead for voltage changes. In our problem, a task cannot be divided into subtasks and assigned to different cores, and transition overhead associated with voltage changes is inevitable in practice; therefore, the result obtained by an optimal solution for this simplified problem must not be worse (should be better in most cases) than that by an optimal solution of our problem. On the other hand, in our approach, the dependencies among tasks inside one period are removed so tasks have more freedom to be moved around. For tasks assigned on the same processor core, transition overhead can be minimized by grouping tasks with the same voltage level together and scheduling groups following ascending order in terms of voltage levels. So the results obtained by our approach are close to those obtained by optimal solutions for this simplified problem. Therefore, optimal solutions of the simplified problem can serve as the theoretical lower bound of our problem.

In Liu et al. [2009], this simplified problem is proved to be NP-complete, and a pseudo-polynomial algorithm based on dynamic programming is proposed to obtain optimal solutions. Although the proposed algorithm is pseudo-polynomial as its complexity is related to the timing constraint, the algorithm is efficient in practice as the execution time of each task is upper bounded by a constant. However, the proposed algorithm in Liu et al. [2009] focuses on optimizing energy consumption with DVS only. Next, based on it, by applying both DVS and DPM, we propose a new algorithm called OLB to obtain the lower bound.

ALGORITHM 4.1: Algorithm OLB (Obtain the Lower Bound of the energy consumption)

Input: A task set V with n independent tasks, $V = \{T_1, T_2, \ldots T_n\}$, timing constraint $M \cdot TC$, a processor core with k different voltage/frequency levels $\{(V_{dd_1}, f_1), (V_{dd_2}, f_2), \ldots, (V_{dd_k}, f_k)\}(V_{dd_1} < V_{dd_2} < \ldots < V_{dd_k})$.

Output: E_{LB}, the lower bound of the energy consumption.

1: $E_{LB} \leftarrow \infty$; $Min\_Time \leftarrow \sum_{T_i \in V} \frac{C(T_i)}{f_k}$.
2: **if** $M \cdot TC < Min\_Time$ **then**
3: No feasible solution and exit.
4: **end if**
5: $Time\_DVS \leftarrow M \cdot TC$; $Time\_DPM \leftarrow 0$.
6: **while** $Time\_DVS \geq Min\_Time$ **do**
7: Using $Time\_DVS$ as the timing constraint, call Algorithm DPVS in Liu et al. [2009] to obtain an optimal voltage assignment for all tasks in V, and let $E_{t\_dynamic}$ be the total dynamic energy with the voltage assignment.
8: For the obtained voltage assignment, let $V_{dd}(T_i)/f(T_i)$ be the corresponding voltage level/frequency of T_i. $Total\_Time\_DVS = \sum_{T_i \in V} C(T_i)/f(T_i)$, and $min\_voltage$=min$\{V_{dd}(T_i)\}$, $T_i \in V$.
9: **if** $(Time\_DVS - Total\_Time\_DVS) > 0$ **then**
10: $E_{idle} \leftarrow (Time\_DVS - Total\_Time\_DVS) \cdot P_{dynamic}(min\_voltage)$.
11: **else**
12: $E_{idle} \leftarrow 0$.
13: **end if**
14: $E_{DVS} \leftarrow E_{t\_dynamic} + P_{static} \cdot Time\_DVS + E_{idle}$.
15: $NumSleepCore \leftarrow Time\_DPM/TC$; $E_{sleepCore} \leftarrow NumSleepCore \cdot TC \cdot P_{sleep}$.
16: $Time\_DPM \leftarrow Time\_DPM\%TC$; $E_{DPM} \leftarrow P_{static} \cdot Time\_DPM$.
17: **if** $Time\_DPM > t_{sleepOH}$ **then**
18: **if** $(P_{sleep} \cdot (Time\_DPM - t_{sleepOH}) + E_{sleepOH}) < E_{DPM}$ **then**
19: Put the core into the sleep mode for the time period $Time\_DPM$.
20: $E_{DPM} \leftarrow P_{sleep} \cdot (Time\_DPM - t_{sleepOH}) + E_{sleepOH}$.
21: **end if**
22: **end if**
23: **if** $E_{LB} > (E_{DVS} + E_{DPM} + E_{sleepCore})$ **then**
24: $E_{LB} \leftarrow E_{DVS} + E_{DPM} + E_{sleepCore}$.
25: **end if**
26: $Time\_DVS - -$; $Time\_DPM + +$.
27: **end while**

In Algorithm 4.1, we separate the total available time into two parts, *Time_DVS* and *Time_DPM*. *Time_DVS* is the time period managed by DVS to schedule all tasks; *Time_DPM* is the idle time managed by DPM. Initially, *Time_DVS* is set as $M \cdot TC$ to represent that all the available time is used for task execution and managed by DVS; *Time_DPM* is set as zero to represent that there is no idle time to be managed by DPM. For *Time_DVS*, the time period for DVS, Algorithm DPVS in Liu et al. [2009] is called to obtain an optimal voltage assignment for all tasks in V with *Time_DVS* as the timing constraint. Based on the voltage assignment, we obtain the total execution time of all tasks and compare it with *Time_DVS*. If there is idle slack, we then apply the lowest voltage on it to compute the idle energy as we can always move the idle slack next to the task that are executed with the lowest voltage level.

For *Time_DPM*, the time period for DPM, we attempt to apply DPM to save energy. If *Time_DPM* is greater than TC, considering that a schedule will be repeatedly executed, it means that one processor core is completely idle; thus, we can turned it off outside the loop so its energy is $TC \cdot P_{sleep}$. We calculate how many idle cores by *Time_DPM* / *TC* and put these cores into the sleep mode. Then we attempt to apply *DPM* on the remaindering

ALGORITHM 5.1: The RDAG Algorithm

Input: A DFG $G = (V, E, \rho, C)$.
Output: The retiming value $r(T_i)$ of each task T_i.
1: **for** each task $T_i \in V$ **do**
2: $r(T_i) \leftarrow 0$
3: **end for**
4: **for** each $T_i \in V$ **do**
5: **if** T_i is a leaf node **then**
6: $ENQUEUE(Q, T_i)$
7: $tail \leftarrow T_i$
8: **end if**
9: **end for**
10: **while** $Q \neq \emptyset$ **do**
11: $T_i \leftarrow DEQUEUE(Q)$
12: **for** each parent node T_j of T_i **do**
13: $r(T_j) \leftarrow \max\{r(T_j), r(T_i) + 1\}$
14: **if** $tail \neq T_j$ **then**
15: $ENQUEUE(Q, T_j)$
16: $tail \leftarrow T_j$
17: **end if**
18: **end for**
19: **end while**

time (*Time_DPM%TC*) in *Time_DPM*, and the time period will be put into the sleep mode if we can save more energy by doing that. Finally, we calculate the total energy based on the power model in Section 2.3, and record the minimum energy accordingly. At the end of each iteration, *Time_DVS* is decreased by one, *Time_DPM* is increased by one, and the above procedure is repeated so all combinations with DVS and DPM can be obtained. The algorithm stops when *Time_DVS* becomes $\sum_{T_i \in V} \frac{C(T_i)}{f_k}$ that is the minimum time we need to execute all tasks (the processor core is operating at the highest voltage at that time).

As shown in Liu et al. [2009], the complexity of Algorithm DPVS is $O(TC \cdot n)$ where TC is the timing constraint, and n is the number of tasks. So the complexity of Algorithm 4.1 is $O(TC^2 \cdot n)$. Usually, the execution time of each task is upper bounded by a constant. So TC is equal to $O(n^c)$ (c is a constant). In this case, Algorithm 4.1 is polynomial.

5. TASK PARALLELIZATION AND SCHEDULING

In this section, we propose our two-phase approach for task parallelization and energy optimization. Because intra-iteration data dependencies of a DFG not only impede parallelism but also cause abundant idle slacks on processor cores, it plays a negative role on energy minimization. Hence, in the first phase, we propose an algorithm called RDAG to remove intra-iteration data dependencies in Section 5.1. Our RDAG algorithm transforms a dependent task graph into a set of independent tasks. We will also analyze the prologue latency and the extra memory overhead caused by the RDAG algorithm. Then in the second phase, we propose a scheduling algorithm called GeneS that adopts a genetic approach to perform energy optimization considering DVS, DPM and various transition overheads in Section 5.2.

5.1. The RDAG Algorithm for Task Parallelization

5.1.1. The RDAG Algorithm. Intra-iteration data dependency in task graphs may impede parallelism and cause idle slacks on processor cores. For example, due to the

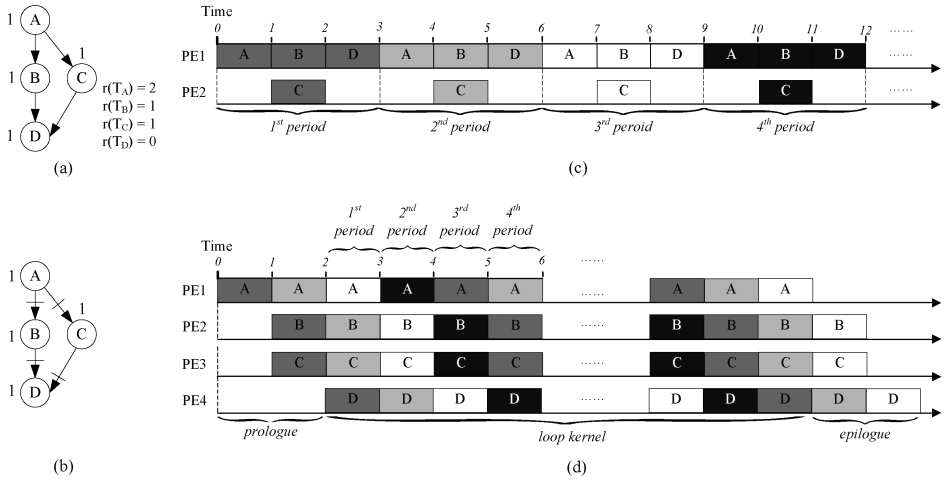


Fig. 3. (a) The original DFG G. (b) The retimed DFG G_r. (c) The static schedule generated from G. (d) The pipelined schedule generated from G_r.

intra-iteration dependencies of the DFG, in Figure 2(d) and Figure 2(e), the schedules can utilize at most two processor cores, in other words, there is no more gain with three or more processor cores. The idle slacks in Figure 2(d) and Figure 2(e) play a negative role in energy consumption. Hence, if we get rid of intra-iteration data dependencies, we can obtain more design space to reduce idle slacks or achieve better parallelism. In this way, more opportunities are provided for energy optimization. Motivated by this, we propose the RDAG algorithm for task parallelization by transforming a DFG into a new DFG with only inter-iteration data dependencies.

According to the definition of retiming, in order to transform periodic dependent tasks into a set of periodic independent tasks, we need to add at least one delay onto each edge of the original DFG. At the same time, we need to find out the minimum retiming value for each node because a big retiming value may cause a big prologue and epilogue. To achieve this, we use Equation (11) to calculate the retiming value of each node as follows:

$$r(T_i) = \begin{cases} \max\{r(T_i), r(T_j) + 1\}, & \text{if } T_i \text{ is } T_j\text{'s parent} \\ 0, & \text{if } T_i \text{ is a leaf node.} \end{cases} \tag{11}$$

In Equation (11), basically, for each leaf node, we set its retiming value as 0; the retiming value of each non-leaf node is calculated from bottom to top. Based on Equation (11), we design the RDAG algorithm that is shown in Algorithm 5.1.

In the RDAG algorithm, two procedures, *ENQUEUE* and *DEQUEUE*, are used for the INSERT and DELETE operations on a queue, respectively. In Lines 1–3, we assign the initial retiming value of each node to 0. In Lines 4–9, we find out all the leaf nodes and put them into a queue named Q, and store the current tail element of Q into a variable named *tail*. Next, in Lines 10–19, we calculate the retiming value of each node based on Equation (11) in a breadth-first manner. Especially, in Lines 14–17, we judge if the parent node T_j of node T_i is the tail element of the current queue. If T_j happens to be the tail element, then we do not need to put node T_j into the queue again. In such a way, we can avoid putting extra adjacent nodes into the queue which will cause unnecessary redundant calculations.

An example is given in Figure 3 to show the potential impact that the RDAG algorithm can provide for scheduling. Figure 3(a) is used to model a streaming application

with periodic dependent tasks and the execution time of each task is listed beside each node. Based on our RDAG algorithm, we obtain the retiming value of each node, and the corresponding retiming values are listed as $r(T_i)$ for each task T_i. Figure 3(b) is the retimed DFG obtained by the RDAG algorithm, in which the count of delays of an edge is represented by the number of bars. In Figure 3(b), there is at least one delay on each edge; therefore, all tasks are independent of each other in one iteration. Suppose that we have a four-core MPSoC, according to Figure 3(a) and Figure 3(b), we generate different schedules as shown in Figure 3(c) and Figure 3(d), respectively. In Figure 3(c), we can see that, due to the inherent data dependency of the application, the schedule can only use 2 cores and the schedule length in each period is 3 time units. After adopting the RDAG algorithm, in Figure 3(d), the scheduler can effectively take advantage of 4 processor cores. In Figure 3(d), each period of the loop kernel consists of 4 tasks coming from 3 different iterations, and the schedule length in each period in the loop kernel is reduced to 1 time unit.

With the RDAG algorithm, by transforming intra-iteration dependencies into inter-iteration dependencies, we can utilize more processor cores to increase parallelism. However, as shown in Figure 3(d), an extra prologue is added in order to make all tasks independent of each other inside one period. Although the prologue is only executed once, it causes the extra latency in the beginning. Also, we may need more memory to hold data caused by regrouping different periods of tasks into one iteration. Next, we analyze the prologue latency of the RDAG algorithm, and discuss the memory overhead caused by the RDAG algorithm.

5.1.2. Prologue Latency. The prologue latency of the RDAG algorithm is caused by rescheduling tasks to previous periods with earlier release time. In the RDAG algorithm, the prologue latency is equal to the time duration of the prologue, and it is determined by the maximum retiming values among all tasks. Given a DFG, let *Prologue_Latency* represent the prologue latency, and it can be calculated by

$$Prologue\_Latency = r_{max} \cdot I = max\{r(T_i)\} \cdot I, (T_i \in V), \tag{12}$$

where r_{max} is the maximum retiming value among all tasks in V, and I is the period.

The maximum retiming value obtained by RDAG is determined by the dependency relations of a DFG. Note that the prologue is only executed once so the overhead it introduces is one-time delay. Typical streaming applications such as high-performance multimedia belong to soft real-time applications in which deadline misses are not desirable but allowed. Therefore, as long as the prologue latency is not too large, our technique can be applied to optimize streaming applications on MPSoC. On the other hand, a streaming application is repeatedly executed for many times. After waiting for the execution of the prologue, tasks can be periodically executed in the new loop kernel. As our approach can greatly reduce the schedule length of each period, we can either apply a shorter period or apply DVS and DPM for energy optimization. So we can benefit from each period in the loop kernel after waiting for the execution of the prologue that only executes once.

5.1.3. Memory Overhead. As the RDAG algorithm regroups tasks from different periods into one period, extra memory space is needed to hold data across different periods. In this section, we analyze the memory space required by our method. To make comprehensive analysis, the memory space needed by both intracore communication and inter-core communication is included. Given a DFG that models a streaming application, a retiming function obtained by RDAG in Section 5.1.1, and a static schedule obtained in Section 5.2, our objective is to obtain the maximum memory space needed by our approach. Based on this, we can analyze the extra energy caused by the extra

ALGORITHM 5.2: Algorithm Obtain_Lifetime()

Input: A DFG $G = (V, E, \rho, C)$, a retiming function r, period I, a static schedule S in which for task T_i, R_{T_i} is its release time and ET_{T_i} is its execution time in S.
Output: A lifetime segment set, $DB\_Lifetime\_Set$, that contains all lifetime segments of all data buffers in the prologue and the first period.

1: Sort all nodes in V in topological ordering.
2: $r_{max} \leftarrow max\{r(T_i)\}, T_i \in V$.
3: **for** each $T_i \in V$ following the topological order **do**
4: **for** each of T_i's adjacent node T_j in G **do**
5: $\rho_r(T_i, T_j) = r(T_i) - r(T_j)$.
6: **for** rt $= 0$; rt $\leq r(T_i)$; rt++ **do**
7: Add li, the lifetime segment of the data buffer that holds the data transferred from T_i to T_j, into $DB\_Lifetime\_Set$, in which
8: $li.start = (r_{max} - \text{rt}) \cdot I + R_{T_i} + ET_{T_i}$;
9: $li.end = (r_{max} - \text{rt} + \rho_r(T_i, T_j)) \cdot I + R_{T_j} + ET_{T_j}$;
10: $li.volume = com(T_i, T_j)$.
11: **if** $li.end > (r_{max} + 1) \cdot I$ **then**
12: $li.end = (r_{max} + 1) \cdot I$.
13: **end if**
14: **end for**
15: **end for**
16: **end for**

memory space introduced by our approach and perform comparison with the previous work.

In order to achieve this, we need to analyze all data transfer in a static schedule across the prologue and all periods. However, as a static schedule will be repeatedly executed starting from the first period as shown in Figure 3, all the memory space can be obtained by analyzing the data transfer in the prologue and the first period. In a schedule, data transfer is associated with two tasks of an edge of a DFG. For an edge (T_i, T_j) in a DFG, we need a data buffer to hold the data from the time when they are generated by T_i to the time when T_j is finished, and this time period is called *the lifetime segment* of a data buffer. The lifetime segment of a data buffer is represented as $< start, end, volume >$ in which "*start*" represents the start time of the segment, "*end*" represents the end time of the segment, and "*volume*" represents the data volume that needs to be transferred through the data buffer. Given a lifetime segment, li, we use $li.start$, $li.end$ and $li.volume$ to represent its start time, end time, and data volume, respectively. To analyze all data transfer in the prologue and the first period, next, we first obtain the lifetime segments of all data buffers associated with all intra-core communication and inter-core communication, and we then conduct lifetime analysis so as to obtain the total data volume of each time unit by putting all lifetime segments together.

Algorithm Obtain_Lifetime() in Algorithm 5.2 is used to collect the lifetime segments of all data buffers needed for data transfer in the prologue and the first period. In Algorithm 5.2, we first obtain the maximum retiming value, r_{max}, among all nodes. As the prologue with $r_{max} \cdot I$ (I is the period) is added in the schedule in our method, this should be counted for calculating the abstract release and end times of a task. Given a static schedule S, for a task $T_i \in V$, let R_{T_i} be its release time in S, then its abstract release time in the first period is $r_{max} \cdot I + R_{T_i}$. In the algorithm, we add the lifetime segment of each data buffer associated with each edge into a set following the topological order. For an edge $(T_i, T_j) \in E$, after retiming, its delay count is $\rho_r(T_i, T_j)$. So in the first period, the lifetime segment of the data buffer associated with (T_i, T_j)

ALGORITHM 5.3: Algorithm Lifetime_Analysis()

Input: *DB_Lifetime_Set* obtained from Algorithm Obtain_Lifetime().
Output: A lifetime segment set, *Total_Lifetime_Set*, by which we can obtain the total data volume of each time unit in the prologue and the first period.

```
1: while DB_Lifetime_Set is not empty do
2:   if There is only one segment in DB_Lifetime_Set then
3:     Remove the segment from DB_Lifetime_Set, and add it into Total_Lifetime_Set.
4:     Exit.
5:   end if
6:   Sort all lifetime segments in DB_Lifetime_Set in ascending order in terms of start times of
     all lifetime segments.
7:   Remove the first two lifetime segments, lt1 and lt2, from DB_Lifetime_Set.
8:   if lt1.start == lt2.start then
9:     if lt1.end == lt2.end then
10:      Add < lt1.start, lt1.end, lt1.volume + lt2.volume > into DB_Lifetime_Set.
11:    else
12:      if lt1.end > lt2.end then
13:        Add two lifetime segments: < lt1.start, lt2.end, lt1.volume + lt2.volume > and
           < lt2.end, lt1.end, lt1.volume >, into DB_Lifetime_Set.
14:      else
15:        Add two lifetime segments: < lt1.start, lt1.end, lt1.volume + lt2.volume > and
           < lt1.end, lt2.end, lt2.volume >, into DB_Lifetime_Set.
16:      end if
17:    end if
18:  else
19:    if lt1.end ≤ lt2.start then
20:      Add < lt1.start, lt1.end, lt1.volume > into Total_Lifetime_Set.
21:      Add < lt2.start, lt2.end, lt2.volume > into DB_Lifetime_Set.
22:    else
23:      Add < lt1.start, lt2.start, lt1.volume > into Total_Lifetime_Set.
24:      if lt1.end == lt2.end then
25:        Add < lt2.start, lt1.end, lt1.volume + lt2.volume > into DB_Lifetime_Set.
26:      else
27:        if lt1.end > lt2.end then
28:          Add two lifetime segments: < lt2.start, lt2.end, lt1.volume + lt2.volume > and
             < lt2.end, lt1.end, lt1.volume >, into DB_Lifetime_Set.
29:        else
30:          Add two lifetime segments: < lt2.start, lt1.end, lt1.volume + lt2.volume > and
             < lt1.end, lt2.end, lt2.volume >, into DB_Lifetime_Set.
31:        end if
32:      end if
33:    end if
34:  end if
35: end while
```

should begin with $r_{max} \cdot I + R_{T_i} + ET_{T_i}$ and end with $(r_{max} + \rho_r(T_i, T_j)) \cdot I + R_{T_j} + ET_{T_j}$, and its data volume is $com(T_i, T_j)$. As defined in Section 2.4, by retiming a node once, one of its copy is moved into the prologue. So for (T_i, T_j), after obtaining its lifetime segment in the first period, correspondingly, we add $r(T_i)$ (the retiming value of T_i) lifetime segments with one period time difference into the set for these data buffers in the prologue. For each segment, if its end time is over the first period whose the abstract time is $(r_{max} + 1) \cdot I$, we change it to be $(r_{max} + 1) \cdot I$ as we only need to calculate up to the first period.

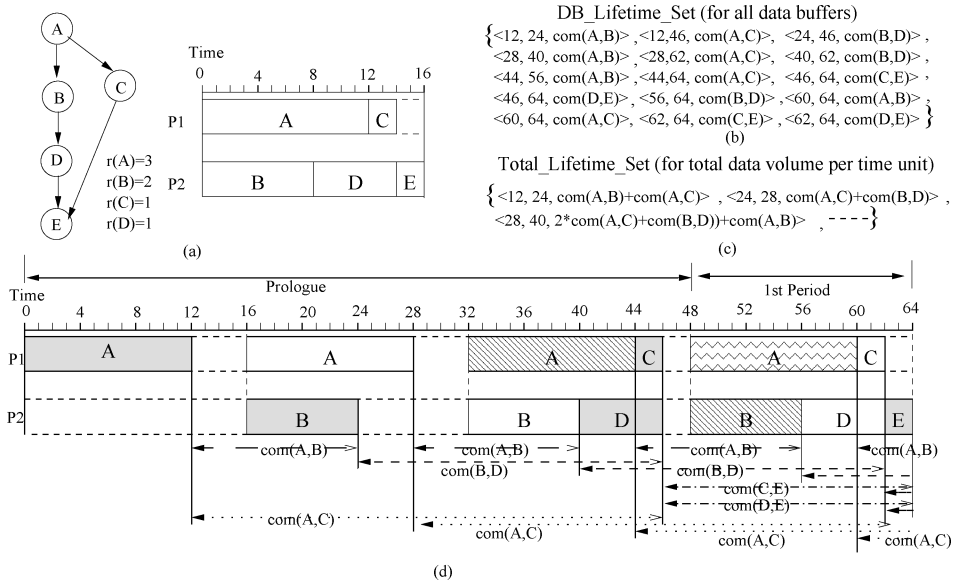


Fig. 4. Latency and memory overhead of the RDAG algorithm.

Algorithm Lifetime_Analysis() in Algorithm 5.3 is to perform lifetime analysis so we can obtain the total data volume of each time unit. The input of Algorithm 5.3 is the lifetime segment set, *DB_Lifetime_Set*, that contains all lifetime segments of all data buffers in the prologue and the first period obtained in Algorithm 5.2. In Algorithm 5.3, in each iteration, we first sort all lifetime segments in *DB_Lifetime_Set* in ascending order in terms of their start times as some new segments may be added into the set. Then following the order, we remove the first two segments from *DB_Lifetime_Set* and compare their start and end times. Basically, when their start times are equal, we combine them together and put the new segments back into *DB_Lifetime_Set*. Otherwise, we output a new lifetime segment that is generated by the start times of the two segments as the data volume of this time period is fixed; then we combine other parts of the two segments and put them back into *DB_Lifetime_Set*. The above procedure is repeated until *DB_Lifetime_Set* is empty or there is only one element is in *DB_Lifetime_Set* when we can directly output that element. The output of Algorithm 5.3 is a set that contains disjoint lifetime segments and each segment represents the total data volume we need to store in the time period from its start time to its end time. Based on it, therefore, we can find the maximum memory space needed by our method.

Using the DFG and schedule in Figure 2 as an example, Figure 4(a) shows the given DFG, the retiming values obtained by our RDAG algorithm, and the schedule obtained by our GeneS algorithm in Section 5.2. Based on the DFG, retiming function and schedule in Figure 4(a), by applying Algorithm 5.2, we can obtain *DB_Lifetime_Set*, which is the set that contains all lifetime segments of all data buffers in the prologue and the first period, shown in Figure 4(b). Figure 4(c) shows *Total_Lifetime_Set*, the output of Algorithm 5.3 based on *DB_Lifetime_Set* in Figure 4(b). Because of limited space, we only list the first three items of *Total_Lifetime_Set* in Figure 4(c). The corresponding schedule with the prologue and the first period is shown in Figure 4(d), in which the lifetime segments of all data transfer are provided. From it, we can see that it is not easy to obtain the maximum space needed directly from a schedule.

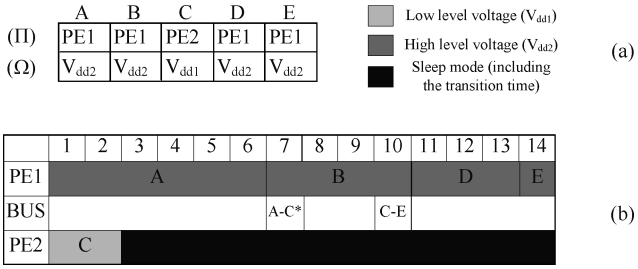


Fig. 5. Chromosome representation and its corresponding task schedule.

The complexity of Algorithm 5.2 is $O(|E|)$ in which E is the total number of edges in a DFG, as we need to traverse each edge in order to obtain the data transfer associated with it. So for *DB_Lifetime_Set*, the output of Algorithm 5.2, its total segment number is bounded by $O(|E|)$. In Algorithm 5.3, we try to find all disjoint sets of all the lifetime segments in *DB_Lifetime_Set* in terms of their start and end times, and the maximum number of all the disjoint sets is $2 \cdot |DB\_Lifetime\_Set|$. Therefore, the complexity of Algorithm 5.3 is $O(|E|)$.

5.2. The GeneS Algorithm for Energy Optimization

In this section, we propose our genetic scheduling algorithm, GeneS, to perform energy optimization with DVS and DPM. GeneS can search and find the best schedule within the solution space generated by gene evolution. Next, we first introduce three key components of our GeneS algorithm, chromosome representation, crossover and mutation (the two basic genetic operators), and the fitness function, in Sections (5.2.1), (5.2.2), and (5.2.3), respectively. We then present our GeneS algorithm in Section 5.2.4.

5.2.1. Chromosome and Schedule. In our approach, each chromosome, ξ_i, consists of two lists of genes, Π and Ω. The content in list Π represents the task assignment of each task, and the content in list Ω represents the voltage level of each task. For each task, its task assignment and voltage selection form a pair of genes. Given a chromosome, we generate a schedule with energy optimization as follows.

—Step 1. *Construct task groups.* Following the genes of tasks, put the tasks that are assigned into the same processor core and have the same voltage level into the same group.
—Step 2. *Generate a schedule with DPM.* For each processor core, if there are some task groups assigned on it, sort the groups in ascending order in terms of their voltage levels and schedule them following the order. If there are idle slacks after all the task groups have been scheduled on a core, move all the idle slacks to the location that is immediately next to the first task group (with the lowest voltage level in the core). Put the idle slacks into the sleep mode if we can save energy by doing it. For a processor core, if there is no any task scheduled on it, put it into the sleep mode.

As just shown, when generating a schedule based on a chromosome, we first group all tasks based on their voltage levels on the same core and then schedule the groups following ascending order in terms of the voltage levels. In this way, we can minimize the total energy/time transition overhead on the core based on Equations (6), (7), (8), and (9). We also move all the idle slacks next to the group with the lowest voltage level and attempt to put them into the sleep mode if we can save energy. For a processor core that is completely idle, as we can turned it off outside the loop so its power is P_{sleep}.

Figure 5 shows a chromosome and its corresponding task schedule. In Figure 5(a), from each pair of genes of a task, we can obtain its task assignment and voltage

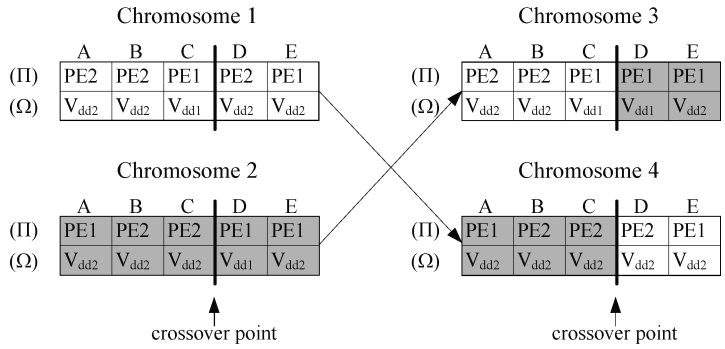


Fig. 6. The crossover operator generates new chromosomes, chromosomes 3 and chromosome 4.

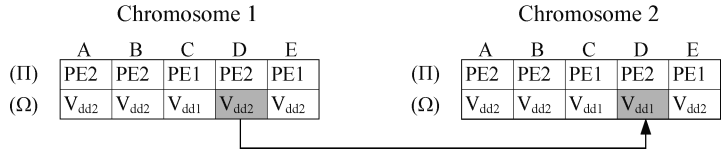


Fig. 7. The mutation operator. Task T_D is selected to perform mutation, and its voltage level is changed from V_{dd_2} to V_{dd_1}.

selection. For example, task T_B is assigned to processor PE_1 with the voltage level V_{dd_2}. Therefore, PE_1 and V_{dd_2} forms a pair of genes for task T_B. Given the chromosome in Figure 5(a), the task schedule can be generated as shown in Figure 5(b). For the tasks that map to processor core PE_1, their voltage levels are the same (V_{dd_2}) so there is only one task group. Similarly we can obtain the schedule on PE_2 in which we put the idle slacks into the sleep mode with DPM.

5.2.2. Crossover and Mutation. We use two genetic operators, *crossover* and *mutation*, to create new generations of chromosomes. The crossover operator selects genes from parent chromosomes and creates new pairs of offspring. For each pair of chromosomes, we randomly select the crossover point of two chromosomes to swap their genes, and create a new pair of chromosomes. Figure 6 shows an example of the crossover operator.

In this example, we perform crossover to create chromosome 3 and chromosome 4 from chromosome 1 and chromosome 2. We use a vertical bar to represent the crossover point. To generate a new pair of offspring, the genes before crossover point (the genes of tasks T_A and T_B) of chromosome 1 and the genes after the crossover point (the genes of tasks T_C, T_D and T_E) of chromosome 2 form chromosome 3. Similarly, the genes of tasks T_A and T_B of chromosome 2 and the genes of tasks T_C, T_D and T_E of chromosome 1 form the chromosome 4.

The mutation operator is used to maintain the genetic diversity from one generation to another. In our approach, for each chromosome, we perform mutation by randomly selecting one task and decreasing its voltage for one voltage level. Figure 7 shows an example of the mutation operator. In this example, after mutation, the voltage level of task T_D changes from V_{dd_2} to V_{dd_1}.

5.2.3. Fitness Function. The fitness function is designed to evaluate each chromosome in order to find a schedule with the minimum energy within the solution space generated by gene evolution. Given the timing constraint TC, a chromosome ξ_i whose schedule length is $L(\xi_i)$ and whose energy consumption is $E_{total}(\xi_i)$ that is calculated based on

ALGORITHM 5.4: The GeneS Algorithm

Input: A set of independent tasks, the timing constraint TC, M processor cores, and each processor core with k different voltage levels $\{V_{dd_1}, V_{dd_2}, \ldots, V_{dd_k}\}$.
Output: An objective task schedule with the minimum energy consumption.
1: Generate the initial generation with N_C chromosomes. In each chromosome, the voltage level of each task is set as the highest voltage V_{dd_k}, and the task assignment is randomly selected.
2: Calculate the fitness value of each chromosome based on Equation 13.
3: Sort chromosomes in the ascending order of the fitness value.
4: Remove $\frac{1}{2}N_C$ chromosomes whose fitness values are smaller.
5: Perform crossover on the preserved $\frac{1}{2}N_C$ chromosomes to create another $\frac{1}{2}N_C$ chromosomes.
6: Calculate the fitness value of each newly generated chromosome based on Equation 13.
7: Sort the preserved $\frac{1}{2}N_C$ chromosomes and the newly generated $\frac{1}{2}N_C$ chromosomes in ascending order in terms of the fitness values, and remove $\frac{1}{4}N_C$ chromosomes whose fitness values are smaller.
8: Randomly select $\frac{1}{4}N_C$ chromosomes from the preserved $\frac{3}{4}N_C$ chromosomes to perform mutation and generate $\frac{1}{4}N_C$ chromosomes.
9: Let the current N_C chromosomes be chromosomes in the new generation.
10: **if** the termination condition is satisfied **then**
11: Let the chromosome with the biggest fitness value be the best solution.
12: **else**
13: Go to Step 4.
14: **end if**

the schedule associated with it, the fitness value, $fitness(\xi_i)$, is defined by

$$fitness(\xi_i) = \begin{cases} \frac{1}{E_{total}(\xi_i)}, & TC \leq L(\xi_i) \\ 0, & TC > L(\xi_i). \end{cases} \tag{13}$$

In the fitness function, both the timing constraint TC and the energy consumption $E_{total}(\xi_i)$ are taken into account. We compare the timing constraint TC with the schedule length $L(\xi_i)$ of the chromosome ξ_i. If the schedule length $L(\xi_i)$ is greater than the timing constraint TC, the fitness value of the chromosome ξ_i, $fitness(\xi_i)$, is equal to zero since the schedule is not feasible; otherwise, its fitness value increases when its total energy consumption $E_{total}(\xi_i)$ decreases. Using this fitness function, we can select the chromosome with the highest fitness value to be the solution of our energy minimization problem.

5.2.4. The GeneS Algorithm. In this section, we present our genetic algorithm, GeneS, to generate the objective task schedule with the minimum energy consumption. Our GeneS algorithm is shown in Algorithm 5.4.

In GeneS, we start from an initial population that is randomly generated by a number of chromosomes. The number of chromosomes in each generation is denoted by N_C. For each chromosome ξ_i in the initial generation, the task assignments are randomly selected, and the voltage level of each task is assigned with the highest voltage level, V_{dd_k}.

Starting from the initial population, we iteratively perform crossover and mutation operators over the chromosomes to create new generations. Based on the fitness function, we calculate the fitness value of each chromosome. The average fitness value of each generation will be increased as we only keep the chromosomes with higher fitness value through evolution. The algorithm terminates when the predefined maximum number of generations is reached.

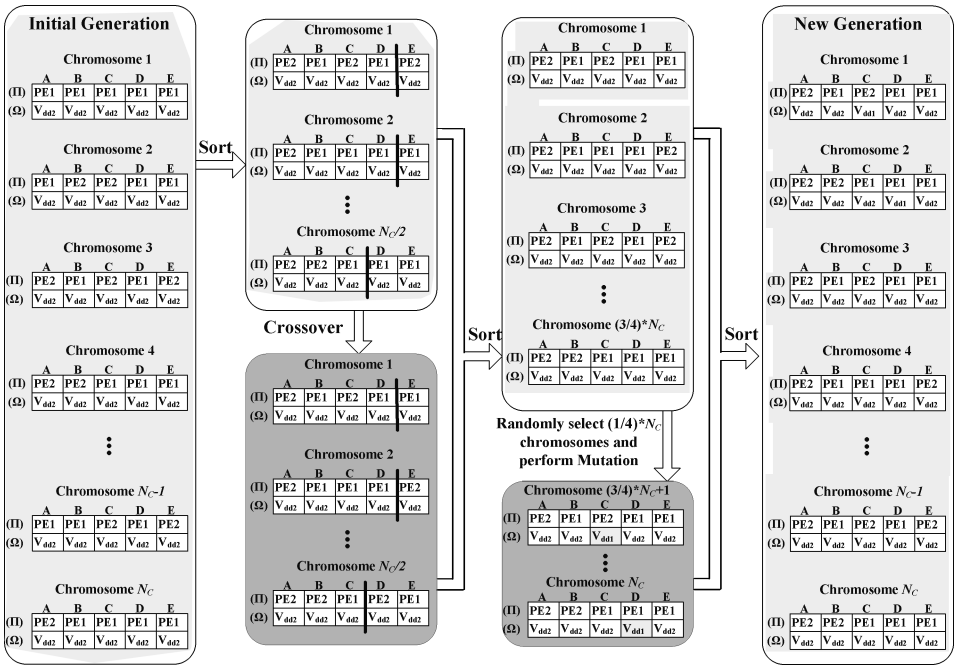


Fig. 8. An example of the GeneS algorithm.

We give an example in Figure 8 to illustrate our GeneS algorithm. Given the DFG in Figure 2(a), the power model in Figure 2(c), two processor core $\{PE_1, PE_2\}$, and the timing constraint $TC = 16$, after applying our RDAG algorithm on the DFG, we can obtain a set of independent tasks. As they are independent of each other, they can be scheduled in any orders in one period. Following GeneS, we first generate the initial generation with N_C chromosomes, in which all tasks in each chromosome are assigned with the highest voltage level V_{dd_2}. We can calculate the fitness value of each chromosome, sort them and remove $\frac{1}{2}N_C$ chromosomes that have the smaller fitness values. The preserved $\frac{1}{2}N_C$ chromosomes form $\frac{1}{4}N_C$ pairs of chromosomes. For each pair of chromosomes, we perform crossover and generate two new chromosomes. For the preserved $\frac{1}{2}N_C$ chromosomes and the newly generated $\frac{1}{2}N_C$ chromosomes, we sort them by the fitness values and remove the $\frac{1}{4}N_C$ chromosomes with the smaller fitness values. From the current $\frac{3}{4}N_C$ chromosomes, we randomly select $\frac{1}{4}N_C$ chromosomes and perform mutation operation. Then we sort the $\frac{3}{4}N_C$ chromosomes and the $\frac{1}{4}N_C$ chromosomes newly generated by the fitness values. These N_C chromosomes form the new generation for gene evolution.

The complexity of our GeneS algorithm is governed by sorting chromosomes and constructing schedules from chromosomes. Let N_G be the number of generations in the GeneS algorithm, let N_C be the number of chromosomes in each generation, and let n be the number of tasks. To generate a new generation, the GeneS algorithm will sort the N_C chromosomes two times, so the corresponding complexity is $O(N_C \; log \; N_C)$. To construct a schedule from a chromosome, in which we can obtain its energy consumption at the same time, we need to group tasks based on their voltage level, sort task groups based on their voltage levels and then schedule task groups. It takes $O(n)$ to group each task, takes at most $O(n \; log \; n)$ to sort all groups in one core, and takes $O(n)$ to perform

Table I. The Voltage Levels, Frequencies and Power Consumption Based on the Power Model of the Mobile Athlon4 Processor [AMD 2001]

Voltage $V_{dd}(V)$	Freq. $f_{op}(MH_z)$	Power $P(W)$	Voltage $V_{dd}(V)$	Freq. $f_{op}(MHz)$	Power $P(W)$
1.2	500	9.2	1.25	600	12.0
1.3	700	15.1	1.35	800	18.6
1.4	1000	25.0			

scheduling. Therefore, the complexity is $O(n\ log\ n)$ for constructing a schedule from one chromosome. So for each generation, it takes $O(N_C \cdot n\ log\ n)$ to construct schedules from chromosomes as we totally need to generate $1\frac{3}{4}N_C$ schedules (the times to calculate the fitness values). Thus, the complexity of our GeneS algorithm is $O(N_G \cdot N_C\ log\ N_C + N_G \cdot N_C \cdot n\ log\ n)$.

6. EXPERIMENTS

In this section, we evaluate and compare our approach with the PEDF algorithm [Zhang et al. 2002] and the SpringS algorithm [Liu et al. 2008] in terms of three performance metrics: (1) the energy consumption, (2) the minimum valid timing constraint, and (3) the extent of parallelism. We will also present the results of prologue latency and memory overhead of our approach.

6.1. Experimental Setup

Power model. The experiments are conducted based on the power model of the AMD Mobile Athlon4 DVS processor [AMD 2001]. The AMD Mobile Athlon4 processor can operate at various voltage levels in the range of $1.2 - 1.4V$ with $50mV$ steps, and the corresponding frequencies vary from $500MHz$ to $1GHz$ with $100MHz$ steps [Mochocki et al. 2004]. The power is calculated by $P_{dynamic} = C_{SW} \cdot f_{op} \cdot V_{dd}^2$ [Rabaey et al. 2002], where C_{SW} is $12.75nF$ from the data sheet of the AMD Mobile Athlon4 processor [AMD 2001]. The five voltage levels, and their corresponding frequencies and power consumption are shown in Table I.

The time overhead during a voltage transition among five voltage levels is calculated based on Equation (6), $t_{TRAN} = \frac{2 \cdot C_{DD}}{I_{MAX}} \cdot |V_{dd_j} - V_{dd_i}|$, in which C_{DD} and I_{MAX} are set as $12pF$ and $16mA$ [AMD 2001]. The energy transition overhead is calculated based on Equation 8 and Equation 9. For the energy consumed by the converter, $E_{TRAN-DC} = \alpha \cdot C_{DD} \cdot |V_{dd_i}^2 - V_{dd_j}^2|$, α is set as 0.9 [Burd 2001].

AMD Mobile Athlon4 DVS processors have low power sleep states that can be utilized when systems are idle. The power consumed in the sleep state is $2.4W$ [Mochocki et al. 2004]. For the transition involving sleep states, considering the synchronization delay with off-chip components such as memory, the transition time is quite large. It takes at least $5ms$ to synchronize with the main memory entering in or exiting from the sleep state [AMD 2001]. The power consumption for transition overhead associated with one transition for entering into and exiting from the sleep mode is assumed to be the power consumption at the voltage level entering from the sleep mode. The power of the bus is assumed to be $147mW$. For static energy consumption, $P_{static} = I_{subn} \cdot V_{dd} + |V_{bs}| \cdot I_j$, the subthreshold current I_{subn} is set as $250\mu A$ [AMD 2001], the body bias voltage V_{bs} is set as $0.4V$, and the reverse bias junction current I_j is set as $4.8 \times 10^{-10}A$ [Martin et al. 2002].

Benchmarks. We conduct experiments on 12 benchmarks as shown in Table II. Among them, the first 9 benchmarks are obtained from Embedded Systems Synthesis Benchmarks (E3S) [Vallerio and Jha 2003]. E3S is largely based on data from the Embedded Microprocessor Benchmark Consortium (EEMBC). Consumer-1 and

Table II. Benchmark Descriptions and Characteristics

Benchmarks	No. of tasks	No. of cycles	Execution time of critical path (μs)
consumer-1	7	11050	8
consumer-2	5	16520	10
auto-industry-1	6	13607	9
auto-industry-2	4	351120	348
auto-industry-3	9	1397567	1392
telecom-1	4	53900	51
telecom-2	6	438900	395
office-1	5	3311000	2584
network-1	4	264127	263
TGFF-1	6	90000	83
TGFF-2	8	170000	58
TGFF-3	24	924000	563

consumer-2 are embedded consumer electronic applications including tasks like JPEG compression, JPEG decompression, high pass gray-scale filter, RGB to CYMK conversion and RGB to YIQ conversion, etc. Auto-industry-1, auto-industry-2 and auto-industry-3 come from embedded auto-industry applications including major tasks like FFT (Fast Fourier Transform), finite/infinite impulse response filter, IDCT (Inverse Discrete Cosine Transform), IFFT (Inverse Fast Fourier Transform), matrix arithmetic, road speed calculation, table lookup and interpolation, etc. telecom-1 and telecom-2 represent embedded telecom applications consisting of tasks like autocorrelation-data (pulse, sine, and speech), convolution encoder-data (xk5r2dt, xk4r2dt, and xk3r2dt), viterbi GSM decoder-data, etc. Office-1 describes an embedded office application which consists of tasks like dithering, image rotation and text processing. Network-1 is an embedded network application including tasks like OSPF/Dijkstra, route lookup/patricia, and packet flow, etc.

Besides the 9 benchmarks, we use TGFF [Dick et al. 1998] to generate 3 periodic task graphs, TGFF-1, TGFF-2, and TGFF-3. Among them, TGFF-1 is a *slim* graph while TGFF-2 is a *fat* graph. Basically, in TGFF-1, it has very long critical path length and there are not many independent nodes; in TGFF-2, its critical path is relatively short and there are many independent nodes. Table II illustrates the detailed information of each benchmark. In Table II, "No. of tasks" represents the number of tasks in each benchmark; "No. of cycles" represents the total number of clock cycles in each benchmark; "Execution time of critical path" represents the total execution time of tasks along the critical path, and each task is with the highest voltage/frequency level (V_{dd_k}, f_k).

6.2. Results and Discussion

In this section, we evaluate the energy consumption of *one iteration* of each benchmark under different timing constraints and different number of processor cores using three algorithms, GeneS, SpringS [Liu et al. 2008], and PEDF [Zhang et al. 2002].

GeneS is our scheduling algorithm that uses genetic approach to solve energy minimization problem for *periodic dependent tasks* on multicore architecture. In the experiments, the maximum number of generations is set as 5000, and the number of chromosomes in each generation is set as 64. SpringS is a scheduling algorithm proposed in Liu et al. [2008]. PEDF is a DVS scheduling algorithm used to solve the energy minimization problem for dependent tasks on multiple variable voltage processors. Instead of fixing the task assignment like Gruian and Kuchcinski [2001] or task scheduling like Luo and Jha [2000], a two-phase framework has been proposed in the PEDF algorithm that integrates task assignment, ordering and voltage selection together to iteratively generate a schedule. So the PEDF algorithm performs better in

Table III. The Energy of Each Benchmark under Various Timing Constraints on 2, 3, 4 Processor Cores

Benchmark	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	PEDF over GeneS (%)
2-core									
consumer-1	8–13	–	249	197	14–19	362	264	264	27.1
consumer-2	10–15	–	391	330	16–22	471	376	311	34.0
auto-1	7–13	–	296	235	14–26	367	315	315	14.2
auto-2	335–348	–	11739	8596	349–705	7828	6476	6449	17.6
auto-3	830–1392	–	31622	25326	1393–2792	30716	30437	24136	21.4
telecomm-1	41–51	–	1600	1176	52–110	1473	1457	1261	14.4
telecomm-2	330–395	–	12445	9107	396–900	13590	12910	11087	18.4
office-1	1925–2584	–	80913	79695	2585–5725	79710	74893	74893	6.0
network-1	253–263	–	8849	6484	264–606	9366	9211	8422	10.1
TGFF-1	48–65	–	2142	2113	66–110	2627	1875	1833	30.2
TGFF-2	87–97	–	–	4183	98–180	3671	3607	3288	10.4
TGFF-3	164–591	–	22263	22263	592–1000	21513	19216	19216	10.7
average									*17.9*
3-core									
consumer-1	5–13	–	271	269	14–19	491	413	413	15.9
consumer-2	6–11	–	421	375	12–22	708	452	452	36.2
auto-1	5–13	–	340	316	14–26	542	469	469	13.5
auto-2	335–348	–	14880	8758	349–705	15229	15205	13695	10.1
auto-3	550–1392	–	32907	30489	1393–2792	30850	30502	30502	1.1
telecomm-1	41–51	–	2022	1176	52–110	2040	1934	1643	19.5
telecomm-2	330–395	–	15779	9108	396–900	18302	17768	14124	22.8
office-1	1925–2584	–	89819	71274	2585–5725	121537	101309	77556	36.2
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2017	1992	66–110	4005	2429	2429	39.4
TGFF-2	57–61	–	4236	4236	62–180	4124	3923	2778	32.6
TGFF-3	309–504	–	21085	21085	505–700	23329	18046	18046	22.6
average									*23.0*
4-core									
consumer-1	5–13	–	344	319	14–19	666	565	565	15.2
consumer-2	5–10	–	441	375	11–22	998	590	590	40.9
auto-1	5–13	–	422	376	14–26	542	515	515	5.0
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	32591	29799	1393–2792	30970	26014	26014	16.0
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	19114	9108	396–900	24374	22625	17160	29.6
office-1	1925–2584	–	110561	78493	2585–5725	167057	130911	95268	43.0
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	233–465	–	20362	14862	466–700	23899	17655	17655	26.1
average									*27.4*

energy consumption compared with the list scheduling algorithm and the algorithms in Gruian and Kuchcinski [2001] and Luo and Jha [2000]. Therefore, PEDF is selected for comparison in the article.

6.2.1. Energy Consumption. Table III and Table IV show the experimental results for all the 12 benchmarks running on 2, 3, 4, 6, and 8 processor cores. In Table III and Table IV, column "TC Range (μs)" represents timing constraints we used that start from the minimum execution time and increase by $2\mu s$ each step. Columns "PEDF (μJ)," "SpringS (μJ)," and "GeneS (μJ)" represent the energy consumption obtained by corresponding algorithms. Column "PEDF over GeneS(%)" represents the percentage of how much extra energy is saved by GeneS compared to PEDF. "-" means for a timing constraint, PEDF or SpringS cannot find a solution. Note that for each benchmark, experimental results are separated into two parts based on different ranges

Table IV. The Energy of Each Benchmark under Various Timing Constraints on 6 and 8 Processor Cores

Benchmark	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	TC Range (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	PEDF over GeneS (%)
6-core									
consumer-1	5–13	–	489	419	14–19	1063	867	867	18.4
consumer-2	5–10	–	509	375	11–22	1186	742	742	37.4
auto-1	5–13	–	587	496	14–26	827	771	771	6.8
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	49354	47501	1393–2792	42449	40632	40632	4.3
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	25783	9108	396–900	36517	32339	23232	36.4
office-1	1925–2584	–	131302	78493	2585–5725	225593	160513	112979	49.9
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	161–321	–	20429	13256	322–600	26214	22168	22168	15.4
average									*26.8*
8-core									
consumer-1	5–13	–	561	469	14–19	1201	1018	1018	15.2
consumer-2	5–10	–	509	375	11–22	1186	742	742	37.4
auto-1	5–13	–	587	496	14–26	827	771	771	6.8
auto-2	335–348	–	18021	8783	349–705	22722	21596	16781	26.1
auto-3	696–1392	–	60890	57622	1393–2792	77374	67009	67009	13.4
telecomm-1	41–51	–	2444	1176	52–110	3083	2742	2024	34.3
telecomm-2	330–395	–	25783	9108	396–900	36517	32339	23232	36.4
office-1	1925–2584	–	131302	78493	2585–5725	225593	160513	112979	49.9
network-1	253–263	–	11223	6484	264–606	14020	12752	10327	26.3
TGFF-1	33–65	–	2173	1966	66–110	5198	3238	3238	37.7
TGFF-2	44–59	–	4199	4172	60–180	5181	4625	3702	28.5
TGFF-3	122–243	–	20597	12785	244–550	29413	26373	26373	10.3
average									*26.9*

of timing constraints. We call the timing constraint at the partition point *the critical timing constraint*. On the left part, the timing constraint is smaller than the critical timing constraint, and the PEDF algorithm has no solution under these small timing constraints. On the contrary, on the right part, the timing constraints are bigger than the critical timing constraint, and PEDF can find feasible schedules. In the experiments, we test each benchmark with different timing constraints. Starting from the minimum timing constraint to perform task scheduling, we gradually increase the timing constraint by $2\mu s$ each step. The experiment results list the average energy consumption of each benchmark for different ranges of timing constraints.

In Table III and Table IV, the results show that with tight timing constraints, the PEDF algorithm cannot achieve a feasible solution while ours can. By extending the timing constraint, PEDF may obtain feasible solutions. On average, GeneS achieves a 24.4% reduction in energy consumption compared with PEDF. For two groups of timing constraints, our algorithm can save extra energy consumption compared with SpringS. From these experimental results, we can see that (1) our scheduling algorithm does not have a strict requirement on timing constraint and can be applied for those embedded systems with tight timing constraint, and (2) our scheduling algorithm can perform better compared with PEDF and SpringS for different number of processor cores.

In order to evaluate our proposed approach with larger number of processor cores, we also present the experiment results on 6 and 8 processor cores. In Table IV, we can see that our approach can also achieve significant energy reduction compared to the PEDF algorithm and the SpringS algorithm. For the MPSoC with 6 and 8 processor cores, our GeneS algorithm can save average 26.8% and 26.9% of energy consumption, respectively, compared to the PEDF algorithm.

Table V. The Comparison of Energy Consumption by the GeneS Algorithm and the Lower Bound Energy Consumption E_{LB}

Benchmark	TC Range (μs)	2-core GeneS (μJ)	2-core E_{LB} (μJ)	4-core GeneS (μJ)	4-core E_{LB} (μJ)	6-core GeneS (μJ)	6-core E_{LB} (μJ)	8-core GeneS (μJ)	8-core E_{LB} (μJ)
consumer-1	5–19	232	216	414	381	592	523	680	666
consumer-2	5–22	319	309	459	440	627	590	627	590
auto-1	5–26	290	257	429	401	643	610	643	610
auto-2	335–705	7049	6499	14660	14660	14660	14660	14660	14660
auto-3	696–2792	24659	20017	27374	24518	42450	39448	62374	58963
telecomm-1	41–110	1189	1125	1640	1122	1640	1122	1640	1122
telecomm-2	330–900	10267	9842	15625	13416	17241	14890	17241	14890
office-1	1925–5725	77290	67835	85901	72794	100241	79493	100241	79493
network-1	253–606	7705	6546	8850	7308	8850	7308	8850	7308
TGFF-1	33–110	1909	1717	2710	1935	2710	1935	2710	1935
TGFF-2	44–180	3381	3135	3756	3534	3756	3534	3756	3534
TGFF-3	122–1000	20220	18147	16502	16191	17293	16962	18147	17075

At this point, we have analyzed the data in one table horizontally. Now we fix the benchmark and analyze the data in the table vertically. This analysis helps us identify whether or not the scheduling algorithm can effectively take advantage of the potential computation power of multiple processor cores. For simplicity, the last benchmark, TGFF-3, is used as an example. The experimental results show that when the number of processor cores is increased from 2 to 8, the minimum valid timing constraint for GeneS (by which we can obtain a feasible solution) is reduced from $464\mu s$ to $122\mu s$ while it is not changed for PEDF. This shows that our GeneS algorithm can effectively exploit the potential of multicore architectures to minimize the energy consumption.

In Section 4, we have analyzed the lower bound energy consumption E_{LB}. As we mentioned in Section 4, E_{LB} is the lower bound energy consumption that is close to the optimal solution, and an optimal solution for multicore processors may not achieve it. Here we compare the energy consumption obtained by the GeneS algorithm and the lower bound energy consumption E_{LB}. Table V shows the experimental results. From the results, we can see that, for all 12 benchmarks, the GeneS algorithm can generate the task schedule with energy consumption close to the lower bound energy consumption E_{LB}. For benchmark auto-2 running on 4, 6, or 8 processor cores, the energy consumption of the GeneS algorithm is the same as the lower bound energy consumption E_{LB}.

6.2.2. Optimization Trade-off on Energy, Number of Processor Cores and Timing Constraint. In the tables above, we compared energy consumption of all benchmarks with different algorithms. In this section, we will list detailed data of two benchmarks and analyze other gains and trade-offs in terms of energy, number of processor cores and timing constraint. We use TGFF-1 and TGFF-2 as examples. TGFF-1 is a *slim* DFG while TGFF-2 is a *fat* DFG. Basically, in the slim DFG, it has very long critical path length and there are not a lot of independent nodes; in the fat DFG, its critical path is relatively short and there are many independent nodes. We attempt to use the two extreme cases to compare GeneS with SpringS and PEDF.

Figure 9 shows the trend of TGFF-1 with three algorithms in terms of the minimum valid timing constraint and energy consumption on 2, 3, and 4 processor cores, respectively. In this figure, the notation GeneS(x), SpringS(x) and PEDF(x) represent the execution trace of corresponding algorithm on x processor cores.

From this figure, we can see that: (1) When the timing constraints are in the scope $[33\mu s, 65\mu s]$, PEDF cannot find a feasible solution while SpringS and GeneS can, which means that GeneS does not put a lot of limitation on timing constraints. This is

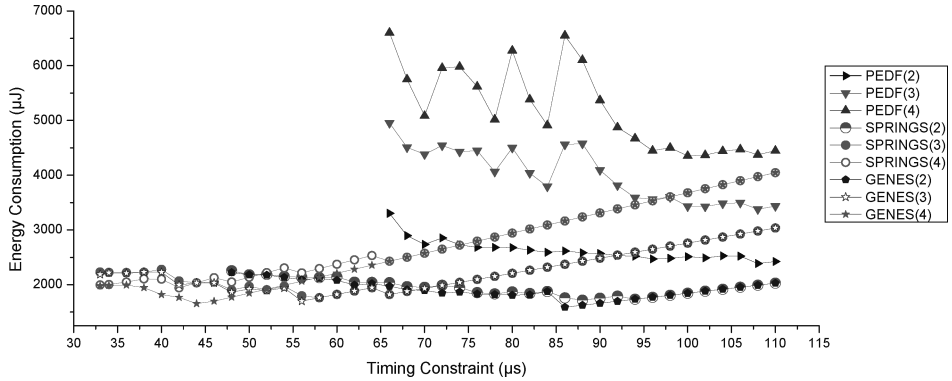


Fig. 9. The changing tendency of energy and timing constraint with three algorithms under different number of processor cores on benchmark TGFF-1.

Table VI. The Comparison for the Schedules Generated by PEDF, SpringS, and GeneS on TGFF-2, a *Fat* Task Graph

TC	2-core			3-core			4-core		
(μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μs)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)	PEDF (μJ)	SpringS (μJ)	GeneS (μJ)
44	–	–	–	–	–	–	–	4398	4398
58	–	–	–	–	4283	4283	–	3861	3861
60	–	–	–	–	4188	4188	6001	3952	3912
62	–	–	–	4651	4120	4118	5923	3845	3833
88	–	–	4233	4150	3684	3658	4587	3237	2123
98	4191	4172	4074	3897	3413	3399	4345	3604	2493
100	4129	4065	4065	3784	3467	3413	4197	3678	2567
120	3874	3779	3750	3605	3315	1411	4422	4414	3307
140	3687	3638	3464	3870	3867	1951	5158	5150	4047
160	3310	3300	3300	4423	4419	2491	5894	5886	4787
180	3326	3324	1955	4975	4971	3031	6630	6622	5527

important for embedded systems, especially for those with tight timing constraints. (2) When the timing constraints are more than $65\mu s$, all of three algorithms can find feasible solutions. However, PEDF and SpringS consumes more energy than GeneS. (3) When the number of processor cores is increased, for GeneS, the minimum feasible timing constraint is reduced from $48\mu s$ to $33\mu s$. However, for PEDF, it is always $65\mu s$ no matter on 2, 3 or 4 processor cores. This comparison shows that GeneS can take advantage of the benefit of multiple processor cores to generate a feasible schedule with more tight timing constraint. (4) Although SpringS can also get feasible solutions in timing constraints $[33\mu s, 65\mu s]$, GeneS can save more energy compared to SpringS.

From this analysis, we can see that GeneS can achieve better energy consumption compared with the PEDF algorithm [Zhang et al. 2002] and the SpringS algorithm [Liu et al. 2008] for the *slim* DFG.

Table VI shows the result of TGFF-2, the *fat* task graph. For simplicity, we list the part of the data derived from the selected timing constraints to show the trend. From the results, similar conclusion can be obtained as for the *slim* DFG. Some little differences are that, for the *fat* DFG, PEDF behaves better than it does for the *slim* DFG. PEDF can use more processor cores compared to the case of the *slim* DFG, while its parallelism is still not as good as that of GeneS. The energy gap between GeneS and PEDF is reduced. However, GeneS still performs better than PEDF. On average, GeneS

Table VII. The Prologue Latency of Our RDAG Algorithm with Different Timing Constraints

Benchmark	TC Range (μs)	*Prologue-Latency* (μs)	TC Range (μs)	*Prologue-Latency* (μs)
consumer-1	5–12	38	13–19	65
consumer-2	5–15	34	16–22	64
auto-1	5–13	56	14–26	103
auto-2	335–348	1021	349–705	1319
auto-3	696–1392	6936	1393–2792	14640
telecomm-1	41–51	137	52–110	228
telecomm-2	330–395	1468	396–900	2568
office-1	1925–2584	6751	2585–5725	10782
network-1	253–263	772	264–606	1242
TGFF-1	33–65	260	66–100	389
TGFF-2	44–59	135	60–180	221
TGFF-3	122–500	3970	501–700	7565

can achieve 23.8% improvement over PEDF in energy consumption for the *fat* DFG. And also, GeneS still performs better than SpringS regarding energy consumption.

6.2.3. Prologue Latency. In Section 5.1.2, we have analyzed the prologue latency of the RDAG algorithm. The prologue latency $Prologue\_Latency$ is the time duration in the prologue. Table VII shows the prologue latency of each benchmark with different timing constraints. From the results, we can see that, our approach causes several periods of prologue latency. As discussed before, however, the prologue is only executed once, so the overhead it introduces is one-time delay. After waiting for the execution of the prologue, tasks can be periodically executed in the new loop kernel as a streaming application is usually repeatedly executed for many times. As our approach can greatly reduce the schedule length of each period as shown above, we can either apply a shorter period or apply DVS and DPM for energy optimization. As we can benefit from each period in the loop kernel, it is usually worth waiting for the execution of the prologue.

6.2.4. Memory Overhead. In this section, we give a case study to illustrate the energy consumption caused by the memory subsystem with different methods. We have analyzed the memory overhead caused by the RDAG algorithm in Section 5.1.3. Using our approach, extra memory space is needed to hold data across different periods as the RDAG algorithm regroups tasks from different periods into one period. Therefore, we should conduct analysis based on the energy overhead caused by the extra memory space from our method. However, we cannot find a consistent SRAM energy model that can clearly show the relation between memory size and energy consumption. For most SRAM chips, only one power consumption parameter is provided, so we cannot effectively evaluate the energy consumption caused by the extra memory space. On the other hand, based on the memory lifetime analysis in Section 5.1.3, we can apply DPM to turn off memory subsystem when it is idle. So in this section, we use Rambus DRAM (RDRAM) as an exemplary memory subsystem to compare the energy consumption between our approach and PEDF by applying DPM.

The power model of RDRAM is listed in Table VIII. RDRAM offers four power modes: active, standby, nap, and powerdown. The energy consumption in each mode and transition times between different modes are listed in Table VIII. Rambus RDRAM chip runs at the frequency of $1600MHz$ and provides a peak transfer rate of $3.2GB/s$ [Pandey et al. 2006].

Table IX shows the energy consumption of memory subsystem by the PEDF algorithm and the RDAG algorithm for processing one iteration of each benchmark. As our approach changes intra-iteration data dependencies into inter-iteration data dependencies, the memory chip has to be in active mode during the whole period. For the PEDF algorithm, memory chip can be put in the standby or power down mode to save

Table VIII. Power Consumption and Transition Time for Memory

Power State/Transition	Power	Time
Active	300mW	-
Standby	180mW	-
Nap	30mW	-
Powerdown	3mW	-
Active → Standby	240mW	1 memory cycle
Active → Nap	160mW	8 memory cycle
Active → powerdown	15mW	8 memory cycle
Standby → Active	240mW	+6ns
Nap → Active	160mW	+60ns
Powerdown → Active	15mW	+6000ns

Table IX. The Memory Energy Consumption of the PEDF Algorithm and the RDAG Algorithm

Benchmark	TC Range (μs)	GeneS (μJ)	PEDF (μJ)			
			2-core	4-core	6-core	8-core
consumer-1	14–19	5.0	3.3	3.1	3.1	3.1
consumer-2	16–32	7.2	5.7	5.6	5.6	5.6
auto-1	14–26	6.0	4.2	4.1	4.1	4.1
auto-2	349–705	158.1	135.7	114.9	114.9	114.9
auto-3	1393–2792	627.8	525.2	517.1	517.1	517.1
telecomm-1	52–100	24.3	10.0	9.3	9.3	9.3
telecomm-2	396–900	194.4	156.6	143.1	137.7	137.7
office-1	2585–5725	1246.5	1027.8	963.0	938.7	938.7
network-1	264–606	130.5	100.8	95.4	95.4	95.4
TGFF-1	85–110	29.3	24.7	23.9	22.0	22.0
TGFF-2	121–180	45.2	40.8	39.5	39.5	39.5
TGFF-3	564–1000	234.6	224.1	216.3	213.1	210.1

energy consumption. Compared to the energy consumption by the PEDF algorithm, our approach causes extra memory consumption overhead. Compared to the energy consumption caused by processor cores, however, the extra energy consumption of memory subsystem caused by our approach is relatively small. If taking the energy consumption caused by memory as one of the components of the overall energy consumption E_{total}, our approach can still significantly reduce the overall energy consumption compared to the PEDF algorithm.

7. CONCLUSION AND FUTURE WORK

In this article, we proposed a two-phase approach to solve the energy optimization problem for periodic dependent tasks on MPSoCs considering various overheads. In the first phase, we proposed a coarse-grained task-level software pipelining algorithm called RDAG to transform periodic dependent tasks into a set of independent tasks based on the retiming technique [Leiserson and Saxe 1991]. In the second phase, we proposed a genetic algorithm called GeneS for energy optimization. We conducted experiments with a set of benchmarks from E3S [Vallerio and Jha 2003] and TGFF [Dick et al. 1998]. The experimental results show that through the combination of software pipelining with DVS and DPM, our approach can fully exploit the potential of MPSoC architectures and the periodic characteristic of streaming applications to reduce energy consumption.

There are several directions for future work. First, we mention that the intercore communication can be overlapped with the computation, but we do not show how to achieve this. To fully take advantage of computation power of an MPSoC, various techniques have been proposed to explore and increase parallelism of streaming applications, and a large amount of intercore communications with significant communication overhead

may be introduced with the increase of parallelism [Varatkar and Marculescu 2003]. So how to reduce intercore communication overhead becomes an important problem. Currently, we are working on solving this problem. Second, task splitting and task migration are not allowed in this work. How to combine our approach with the techniques applying task splitting and task migration is another direction we can explore. Third, we will study how to integrate our technique into a compiler and real-time operating systems to leverage system-wide energy consumption.

REFERENCES

ACHARYA, S. AND MAHAPATRA, R. 2008. A dynamic slack management technique for real-time distributed embedded systems. *IEEE Trans. Comput. 57,* 2, 215–230.

ALBA, E. AND TROYA, J. M. 1999. A survey of parallel distributed genetic algorithms. *Complex. 4,* 4, 31–52.

ALENAWY, T. A. AND AYDIN, H. 2005. Energy-aware task allocation for rate monotonic scheduling. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS'05)*. IEEE Computer Society Press, Los Alamitos, CA, 213–223.

AMD. 2001. Mobile AMD Athlon 4 processor model 6 CPGA data sheet. Advanced Micro Devices, Tech, rep. 24319.

AYDIN, H., DEVADAS, V., AND ZHU, D. 2006. System-level energy management for periodic real-time tasks. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE Computer Society Press, Los Alamitos, CA, 313–322.

AYDIN, H., MELHEM, R., MOSSÉ, D., AND MEJÍA-ALVAREZ, P. 2001. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS'01)*. IEEE Computer Society Press, Los Alamitos, CA, 225–232.

AYDIN, H. AND YANG, Q. 2003. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS'03)*. IEEE Computer Society Press, Los Alamitos, CA, 113–121.

BAMBHA, N. K. AND BHATTACHARYYA, S. S. 2000. A joint power/performance optimization algorithm for multiprocessor systems using a period graph construct. In *Proceedings of the 13th International Symposium on System Synthesis (ISSS'00)*. IEEE Computer Society Press, Los Alamitos, CA, 91–97.

BINI, E., BUTTAZZO, G., AND LIPARI, G. 2005. Speed modulation in energy-aware real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*. IEEE Computer Society Press, Los Alamitos, CA, 3–10.

BURD, T. 2001. Energy-efficient processor system design. Ph.D. thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

CHAO, L.-F. AND LAPAUGH, A. 1993. Rotation scheduling: A loop pipelining algorithm. In *Proceedings of the 30th International Design Automation Conference (DAC'93)*. ACM, New York, NY, 566–572.

CHAO, L.-F. AND SHA, E. H.-M. 1993. Static scheduling of uniform nested loops. In *Proceedings of 7th International Parallel Processing Symposium*. IEEE Computer Society Press, Los Alamitos, CA, 254–258.

CHEN, J.-J. AND KUO, T.-W. 2005. Energy-efficient scheduling of periodic real-time tasks over homogeneous multiprocessors. In *Proceedings of the 2nd International Workshop on Power-Aware Real-Time Computing (PARC'05)*. IEEE Computer Society Press, Los Alamitos, CA, 30–35.

CHEN, J.-J., KUO, T.-W., AND SHIH, C.-S. 2005. $1 + \epsilon$ approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05)*. ACM, New York, NY, 247–250.

DICK, R., RHODES, D., AND WOLF, W. 1998. TGFF: Task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign (CODES'98)*. ACM, New York, NY, 97–101.

EL-REWINI, H., ALI, H. H., AND LEWIS, T. 1995. Task scheduling in multiprocessing systems. *Computer 28,* 12, 27–37.

GRUIAN, F. AND KUCHCINSKI, K. 2001. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'01)*. ACM, New York, NY, 449–455.

HU, J. AND MARCULESCU, R. 2004. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the Conference on Design Automation and Test in Europe (DATE'04)*. IEEE Computer Society Press, Los Alamitos, CA, 234–239.

HUA, S. AND QU, G. 2005. Voltage setup problem for embedded systems with multiple voltages. *IEEE Trans. VLSI Syst. 13,* 7, 869–872.

HUNG, C.-M., CHEN, J.-J., AND KUO, T.-W. 2006. Energy-efficient real-time task scheduling for a dvs system with a non-dvs processing element. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE Computer Society Press, Los Alamitos, CA, 303–312.

JEJURIKAR, R. AND GUPTA, R. 2004. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'04)*. ACM, New York, NY, 78–81.

JHA, N. K. 2001. Low power system scheduling and synthesis. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*. IEEE Press, Los Alamitos, CA, 259–263.

RABAEY, J. M., CHANDRAKASAN, A., AND NIKOLIC, B. 2002. *Digital Integrated Circuits 2nd Ed*. Prentice Hall, Englewood Cliffs, NJ.

KIANZAD, V., BHATTACHARYYA, S. S., AND QU, G. 2005. Casper: An integrated energy-driven approach for task graph scheduling on distributed embedded systems. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*. IEEE Computer Society Press, Los Alamitos, CA, 191–197.

KIM, N. S., KGIL, T., BOWMAN, K., DE, V., AND MUDGE, T. 2005. Total power-optimal pipelining and parallel processing under process variations in nanometer technology. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'05)*. IEEE Computer Society Press, Los Alamitos, CA, 535–540.

LANDSKOV, D., DAVIDSON, S., SHRIVER, B., AND MALLETT, P. W. 1980. Local microcode compaction techniques. *ACM Comput. Surv. 12,* 3, 261–294.

LEISERSON, C. E. AND SAXE, J. B. 1991. Retiming synchronous circuitry. *Algorithmica 6*, 5–35.

LI, J. AND MARTÍNEZ, J. F. 2005. Power-performance considerations of parallel computing on chip multiprocessors. *ACM Trans. Archit. Code Optim. 2,* 4, 397–422.

LIU, H., SHAO, Z., WANG, M., AND CHEN, P. 2008. Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'08)*. IEEE Computer Society Press, Los Alamitos, CA, 92–101.

LIU, H., SHAO, Z., WANG, M., DU, J., XUE, C. J., AND JIA, Z. 2009. Combining coarse-grained software pipelining with dvs for scheduling real-time periodic dependent tasks on multi-core embedded systems. *J. Signal Process. Syst. 57,* 2, 249–262.

LUO, J. AND JHA, N. K. 2000. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'00)*. IEEE Press, Los Alamitos, CA, 357–364.

LUO, J. AND JHA, N. K. 2007. Power-efficient scheduling for heterogeneous distributed real-time embedded systems. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst. 26,* 6, 1161–1170.

MARTIN, S. M., FLAUTNER, K., MUDGE, T., AND BLAAUW, D. 2002. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'02)*. IEEE Computer Society Press, Los Alamitos, CA, 721–725.

MEJIA-ALVAREZ, P., LEVNER, E., AND MOSSÉ, D. 2004. Adaptive scheduling server for power-aware real-time tasks. *ACM Trans. Embed. Comput. Syst. 3,* 2, 284–306.

MITCHELL, M. 1996. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA.

MOCHOCKI, B., HU, X., AND QUAN, G. 2004. A unified approach to variable voltage scheduling for nonideal DVS processors. *IEEE Trans. Comput. Aid. Des. Integr. Circ. Syst. 23,* 9, 1370–1377.

NIU, L. AND QUAN, G. 2006. System-wide dynamic power management for portable multimedia devices. In *Proceedings of the 8th IEEE International Symposium on Multimedia (ISM'06)*. IEEE Computer Society Press, Los Alamitos, CA, 97–104.

PANDEY, V., JIANG, W., ZHOU, Y., AND BIANCHINI, R. 2006. Dma-aware memory energy management. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA'06)*. IEEE Computer Society Press, Los Alamitos, CA, 133–144.

PASSOS, N. L. AND SHA, E. H.-M. 1996. Achieving full parallelism using multidimensional retiming. *IEEE Trans. Parall. Distrib. Syst. 7,* 11, 1150–1163.

QUAN, G. AND HU, X. 2002. Minimum energy fixed-priority scheduling for variable voltage processor. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'02)*. IEEE Computer Society Press, Los Alamitos, CA, 782–787.

SAEWONG, S. AND RAJKUMAR, R. R. 2003. Practical voltage-scaling for fixed-priority RT-systems. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*. IEEE Computer Society Press, Los Alamitos, CA, 106–114.

SHAO, Z., WANG, M., CHEN, Y., XUE, C., QIU, M., YANG, L. T., AND SHA, E. H. M. 2007. Real-time dynamic voltage loop scheduling for multi-core embedded systems. *IEEE Trans. Circ. Syst. II 54,* 5, 445–449.

SHIN, D., KIM, J., AND LEE, S. 2001. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. ACM, New York, NY, 438–443.

VALLERIO, K. S. AND JHA, N. K. 2003. Task graph extraction for embedded system synthesis. In *Proceedings of the 16th International Conference on VLSI Design (VLSID'03)*. IEEE Computer Society Press, Los Alamitos, CA, 480–486.

VARATKAR, G. AND MARCULESCU, R. 2003. Communication-aware task scheduling and voltage selection for total systems energy minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'03)*. IEEE Computer Society Press, Los Alamitos, CA, 510–517.

WANG, Y., LIU, D., WANG, M., QIN, Z., AND SHAO, Z. 2010. Optimal task scheduling by removing inter-core communication overhead for streaming applications on MPSoC. In *Proceedings of the 16th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'10)*. IEEE Computer Society Press, Los Alamitos, CA, 195–204.

WIEGAND, T., SULLIVAN, G. J., BJONTEGAARD, G., AND LUTHRA, A. 2003. Overview of the H.264/AVC video coding standard. *IEEE Trans. Circ. Syst. Video Technol. 13,* 7, 560–576.

XU, R., MELHEM, R., AND MOSSE, D. 2007. Energy-aware scheduling for streaming applications on chip multiprocessors. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. IEEE Computer Society Press, Los Alamitos, CA, 25–38.

YU, Y. AND PRASANNA, V. 2002. Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In *Proceedings of the 9th International Conference on Parallel and Distributed Systems (ICPADS'02)*. IEEE Computer Society Press, Los Alamitos, CA, 341–348.

ZHANG, Y., HU, X. S., AND CHEN, D. Z. 2002. Task scheduling and voltage selection for energy minimization. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, New York, NY, 183–188.

ZHONG, X. AND XU, C.-Z. 2007. Frequency-aware energy optimization for real-time periodic and aperiodic tasks. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*. ACM, New York, NY, 21–30.

Received July 2009; January 2010; accepted September 2010