A General Algorithm for Energy-Aware Dynamic Reconfiguration in Multitasking Systems

Weixun Wang Sanjay Ranka Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL
{wewang,ranka,prabhat}@cise.ufl.edu

***Abstract*—System optimization techniques based on dynamic reconfiguration are widely adopted for energy conservation. While dynamic voltage scaling (DVS) techniques have been extensively studied for processor energy conservation, dynamic cache reconfiguration (DCR) for reducing cache energy consumption in multitasking systems is still in its infancy. In this paper, we propose a general and flexible algorithm for energy optimization based on dynamic reconfiguration in multitasking systems. Our algorithm is flexibly parameterized and can be used to provide tradeoffs between running time and solution quality. Furthermore, it can easily incorporate variable reconfiguration overhead. Experimental results show that our technique can generate near-optimal solutions with significantly low running time and memory requirements.**

I. Introduction

Energy conservation is the key design consideration for embedded systems. Various techniques have been proposed over the years to reduce the energy consumption of processor as well as memory subsystem. Dynamic voltage scaling (DVS) can be effectively used to reduce the power requirement quadratically while only slowing the processor performance almost linearly. Recent studies show that memory hierarchy, especially the cache subsystem, has become comparable to the processor in terms of energy consumption [1]. Dynamic cache reconfiguration (DCR) provides the ability to change cache configuration at run time so that it can satisfy each application's unique requirement in cache size, line size and associativity. DCR is capable of improving cache energy efficiency as well as overall performance [2].

In real-time systems, multiple tasks execute in the system simultaneously by sharing common resources. In uniprocessor systems, only one task can execute at any point of time. Furthermore, each task normally has its arrival time and deadline constraints. Therefore, we have to decide when and how (under which voltage level/cache configuration) to execute each task. While the former decision is made by scheduling algorithms, the latter one is decided by DVS/DCR techniques.

In this paper, we develop a general algorithm that comprehensively solves energy-aware reconfiguration problems in uniprocessor multitasking systems. Our contribution can be summarized as:

1) The algorithm assumes that each task can be executed under one or multiple configurations and finds the optimal configuration assignment to minimize energy consumption while ensuring all the time constraints. Each configuration could correspond to one cache configuration, one voltage level or a combination of them. Therefore the algorithm can either separately or simultaneously accommodate DCR and DVS techniques.
2) It allows differential cost of switching from one configuration to another. Thus, it has advantages over existing techniques that it can effectively take variable runtime overhead into account.
3) The algorithm can be flexibly parameterized to tradeoff between algorithm running time and solution quality. Our experimental results show that the running time can be drastically reduced while only minor quality degradation is observed.

Furthermore, our algorithm is relatively independent of the scheduling policy and task properties. It can support tasks with/without time constraint, preemptive/non-preemptive scheduling or periodic/aperiodic tasks.

The rest of the paper is organized as follows. Related works are surveyed in Section II. We formulate and analyze the problem in Section III. Next, we describe our algorithm and various design considerations in Section IV. Section V presents our experimental results. Finally, Section VI concludes the paper.

II. Related Work

DCR has recently drawn considerable interests in both general-purpose [3] as well as real-time systems [4] [5]. DCR needs the support of reconfigurable cache architectures as proposed in [1] [2]. The major challenge for employing DCR in multitasking systems is to determine when and how to reconfigure the cache so that energy consumption is minimized while each task's timing constraints are satisfied. Wang et al. applied DCR in soft real-time systems by utilizing static profiling information at runtime for both single level cache [4] and multiple level cache hierarchy [5]. Recent efforts [6] tried to combine DCR and DVS together in hard real-time systems. However, these techniques are either designed for specific systems (e.g., soft real-time systems in which missing task deadlines are tolerable) or specific task characteristics (e.g., periodic tasks). Moreover, they are also based on certain assumptions, e.g., negligible or fixed reconfiguration overhead.

DVS is widely supported in general as well as specific-purpose processors [7]. DVS techniques have been developed

*This work was partially supported by NSF grant CCF-0903430 and SRC grant 2009-HJ-1979.

for periodic task sets [8], aperiodic tasks [9], preemptive and non-preemptive tasks [10], [11]. Inter-task DVS, in which each task is solely assigned one voltage level, is exploited in most existing works [12]. Its counter-part, intra-task DVS, which exploits dynamic slack created by early finished jobs, is studied in [13]. PreDVS [14] can lead to more energy savings than optimal inter-task DVS without introducing any extra overhead. Temperature constraint is also considered in recent DVS approaches [15]. Chen et al. [16] presents a survey on DVS techniques in real-time systems. Swaminathan et al. [17] modeled the uniprocessor voltage scaling for real-time system as a generalized network flow problem and solved it using network flow algorithms. However, their method does not support cache reconfiguration and only considered voltage switching at task boundaries. Moreover, their method cannot incorporate variable runtime overhead nor make tradeoff between running time and design quality. We address these limitations in the methods proposed in this paper.

III. Problem Formulation

A. Energy Model

***Cache Energy Model*:** Our cache energy model is adopted from [2]. Let $E_{dynamic}$ and E_{static} denote the dynamic energy and static energy of the cache subsystem, respectively. The total cache energy consumption hence is $E_{cache} = E_{dynamic} + E_{static}$. Specifically, we have:

$$E_{dynamic} = num\_accesses \cdot E_{access} + num\_misses \cdot E_{miss} \quad (1)$$

$$E_{miss} = E_{offchip\_access} + E_{\mu P\_stall} + E_{block\_fill} \quad (2)$$

$$E_{static} = P_{static} \cdot CC \cdot t_{cycle} \quad (3)$$

where E_{access}, E_{miss} and P_{static} are the energy required per cache access, per cache miss and static power consumption, respectively, which are all collected from CACTI [18] for all cache configurations. Here, CC denotes the number of clock cycles that is required to execute the task, and t_{cycle} is the length of each clock cycle. Following [2], we represent energy consumption for fetching data from off-chip memory, processor stall due to cache miss and cache block refilling after a miss by $E_{offchip\_access}$, $E_{\mu P\_stall}$ and E_{block_fill}, respectively.

Processor Energy Model: The dynamic power dissipation of the processor can be characterized as:

$$P_{dynamic} = K \cdot C_{eff} \cdot V_{dd}^2 \cdot f \quad (4)$$

where K is an application-specific constant which represents the average number of switches in one cycle, C_{eff} is the total switching capacitance of the processor, V_{dd} is the supply voltage and f denotes the operation frequency. Note that different applications may have various processor energy profile decided by how much effective switches they actually use during execution ($K \cdot C_{eff}$). Leakage power is given by [19]:

$$P_{static} = V_{dd} \cdot I_{subth} + |V_{bs}| \cdot I_j \quad (5)$$

where I_{subth} and I_j are the subthreshold current and reverse bias junction current, respectively. V_{bs} denotes the body bias voltage. Note that I_{subth} is in direct proportion to V_{dd} and V_{bs}. Processor energy consumption is calculated as: $E_{processor} = (P_{dynamic} + P_{static}) \cdot \frac{CC}{f}$. This energy model is also used in [6].

B. Task Model

We are given:

- A highly configurable cache architecture which supports h different configurations C$\{c_1, c_2, \ldots, c_h\}$ and/or,
- A voltage scalable processor which supports l different voltage levels V$\{v_1, v_2, \ldots, v_l\}$

Task set can be characterized as the following:

- A set of m independent tasks T$\{\tau_1, \tau_2, \ldots, \tau_m\}$.
- Each task $\tau_i \in$ T has known attributes including arrival time, deadline or period (if it is periodic).
- Each task τ_i has known worst-case workload.

We use worst-case workload of each task since we focus on static slack allocation. In practice, the bound can be found by any existing worst-case execution time analysis techniques. Our goal is to find a voltage/cache configuration assignment for each task that minimizes the total energy consumption while ensuring that all the time constraints are met.

IV. Energy-Aware Reconfiguration Algorithm

Our proposed approach accepts a trace of *execution blocks* as the input. Given a task set and a scheduling policy, we first execute all the tasks under the base case (under the base cache configuration[1] in DCR or the highest voltage level for DVS) assuming each of them requires its worst-case workload. The scheduler generates the execution blocks in temporal order. Note that for non-preemptive scheduling, execution blocks are essentially a sequence of task instances (jobs) with each of them having an absolute deadline and earliest start time (arrival time). In preemptive systems, however, execution blocks can be segments of tasks produced by preemptions. Figure 1 illustrates the relation between execution blocks and the tasks which they belong to. Suppose there are three periodic tasks τ_1, τ_2 and τ_3 with the characteristics of $(1,3,3)$[2], $(2,5,5)$ and $(4,12,12)$. Under EDF schedule, there are 10 execution blocks (b_1, b_2, ... , b_{10}) before time unit 12. Our algorithm makes reconfiguration decisions on the granularity of each execution block. Thus, it is optimal in non-preemptive systems with inter-task manner DVS/DCR. It can also generate more energy savings in preemptive systems without introducing additional runtime overhead since a context switching has to be carried out during task preemption.

For DVS, if tasks' energy profiles are identical, the energy consumption and execution time of each execution block can be calculated according to the processor energy model. For DCR or DVS with variable task energy profile, these values need to be collected using static profiling [4]. Only Pareto-optimal configurations are considered for each block. Specifically, for DVS, since leakage power is considered,

[1]The base cache is defined as the configuration used in the system without DCR capability that meets all task deadlines.

[2]Here the three numbers stand for execution time, period and relative deadline, respectively.

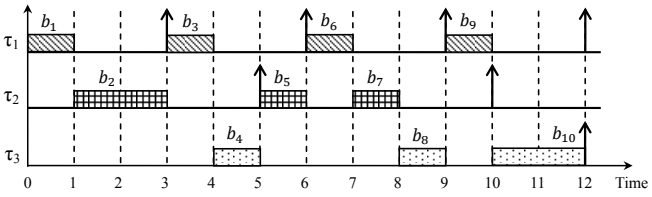

Fig. 1. Tasks and execution blocks.

the minimum voltage level is lower bounded (as a further decrease will lead to increasing in overall energy consumption [6]). For DCR, each block's Pareto-optimal points are those cache configurations which are not dominated by any other configuration in terms of both energy consumption and time requirement. Note that Pareto-optimal configuration set is application-specific. In this section, we define a general term *configuration* which could be a cache configuration, a voltage level, or any other form of system configuration. Let h and h_i denote the total number of available configurations and the number of Pareto-optimal configurations for the i^{th} execution block, respectively.

We model the runtime reconfiguration overhead as variables depending on the transition from one configuration to another. For example, the overhead for reconfiguring a 4KB cache to a 8KB cache is generally larger than just changing the line size from 16 bytes to 32 bytes since the former requires waking up cache banks but the later is done by line concatenation. The input to our algorithm can be formally represented as:

- A set of n execution blocks B$\{b_1, b_2, ..., b_n\}$.
- Execution block $b_i \in$ B has an arrival time a_i if it is the first block in the task instance and an absolute deadline d_i if it is the last block.
- Execution block b_i has execution time t_i^k and energy consumption e_i^k under configuration k (c_k).
- Reconfiguration energy overhead $\rho(i,j)$ and time overhead $\sigma(i,j)$ for converting from configuration c_i to configuration c_j.

Note that a_i and d_i correspond to the task to which the execution block belongs. a_i and d_i are set to -1 when they are not applicable to block b_i. If we denote t_i as the start time and k_i as the index of the configuration assigned to block b_i given in the solution, the general dynamic reconfiguration problem $\wp$ can be formulated as[3]:

$$minimize\ E = \sum_{i=1}^{n} (e_i^{k_i} + \rho(c_{k_{i-1}}, c_{k_i})) \tag{6}$$

subject to,

$$t_i \geqslant a_i, \forall a_i \geqslant 0 \tag{7}$$

$$t_i + \sigma(c_{k_{i-1}}, c_{k_i}) + t_i^{k_i} \leqslant d_i, \forall d_i \geqslant 0 \tag{8}$$

$$t_{i+1} \geqslant t_i + \sigma(c_{k_{i-1}}, c_{k_i}) + t_i^{k_i}, \forall i \in [1,n) \tag{9}$$

Equation (7) represents the timing constraint that all the execution blocks must start executing after the task instance's arrival time. Equation (8) ensures deadline is not violated for any task. Note that time overhead is accounted at the beginning of task execution. Since we stick to the original schedule, Equation (9) guarantees the execution order of all the blocks in the final solution. The goal is to find k_i for all blocks in B so that Equation (6) can be achieved. The described modelling method makes our approach generally applicable – it does not depend on any task set characteristic or scheduling algorithm.

[3] c_{k_0} denotes the initial configuration.

A. Extended Complete Bipartite Graph

We formulate the dynamic reconfiguration problem $\wp$ as a minimum-cost path finding problem in an extended complete bipartite graph (ECBG) as shown in Figure 2. Unlike traditional complete bipartite graph, an ECBG has multiple (specifically, n) disjoint sets $\{V_1, V_2, ..., V_n\}$ and a single source node as well as a single destination node. Every node in one set is connected to every node in its neighboring sets. Formally, an ECBG can be defined as ECBG$\{V_1 + V_2 + ... + V_n, E\}$ such that for any two nodes $v_i^k \in V_i$ and $v_{i+1}^j \in V_{i+1}$, there is an edge (v_i^k, v_{i+1}^j) in E.

Semantically, each disjoint set V_i represents an execution block b_i in B. Each node in the disjoint set stands for one configuration for that block. Hence, the number of nodes in set V_i is h_i. Each edge (v_i^k, v_{i+1}^j) in E is associated with two values: e_i^k and t_i^k. It means that, by moving from set V_i to V_{i+1} through this edge (choosing c_k), it requires t_i^k time units and e_i^k units of energy to execute block b_i. The runtime overhead is also taken into account on each edge. Specifically, edge (v_i^k, v_{i+1}^j) carries a pair of values: $(e_i^k + \rho(c_k, c_j), t_i^k + \sigma(c_k, c_j))$. Therefore, the objective shown in Equation (6) is algorithmically equal to finding a path from the source node to the destination node in the ECBG which has the minimum accumulative energy E. This path contains one and only one node from each disjoint set (choosing one edge between neighboring sets), which corresponds to selecting one configuration for each block. Moreover, all the constraints shown in Equation (7), (8) and (9) have to be satisfied in the path. For those nodes with arrival time constraint, say b_i, it is possible that the finish time of its previous node b_{i-1} is earlier than a_i. To ensure $t_i \geqslant a_i$, there is an idle node before every block node to represent the possible idle intervals. Note that edge (v_1^1, v_2^1) does not involve any overhead since no reconfiguration is carried out (i.e. $k_1 = k_2$). However, edge (v_2^1, v_3^2) includes reconfiguration overhead $\rho(c_1, c_2)$ and $\sigma(c_1, c_2)$.

B. Minimum-Cost Path Algorithm

In this paper, we employ a dynamic programming based algorithm to find the minimum-cost path. Let E_i and T_i denote the total energy consumption (cost) and execution time up to node b_i. Starting from the first node, for each node b_i, we find the lowest cost E_i under each possible value of T_i and possible configuration choice for b_i (i.e. c_{k_i}), in a node by node manner until the destination node is reached. If there is no such partial path which has an accumulative execution time no larger than a specific value of T_i and ends up with a specific configuration for b_i, the corresponding E_i is set to infinity. The calculation of all E_i values for each node is based on the lowest cost values

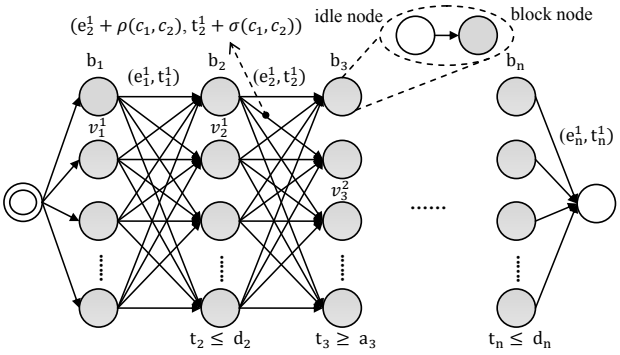

Fig. 2. ECBG model of $\wp$.

of its previous node calculated in last step. At each step, say b_i, we know the lowest total energy of last $i-1$ nodes under each possible value of T_{i-1} and configuration for b_{i-1}. Based this information and various overhead, we can easily find the minimum E_i under all possible T_i and c_{k_i}.

Since the execution time is continuous but the design space is actually discrete (consists of finite number of choices), it is neither possible nor necessary to consider all possible values of T_i. Hence, we discretize T_i into a finite set of values. The interval between two adjacent discretized values is regarded as one time unit, which could be as small as one clock cycle or as large as one millisecond in practice. To reduce running time, we can limit T_i within the rage of $[T_{min}, T_{max}]$. We set $T_{min} = \sum_{i=1}^{n} t_i^h$ where t_i^h is the execution time under the most performance efficient configuration. T_{max} can be set to the deadline constraint of last task instance or the common deadline for all tasks. In other words, all blocks need to be completed before T_{max}. A three-dimensional array D is created for dynamic programming in which each element $D[i][\tau][j]$ stores the lowest total cost for nodes $b_1, b_2, ..., b_i$ while total execution time T_i is equal or less than τ ($T_i \leqslant \tau$) and configuration choice for b_i is c_j. As a result, there are n rows in D with each row consisting of $(T_{max} - T_{min})$ vectors and each vector has h elements. Therefore, the recursive relation for our dynamic programming scheme can be represented as:

$$D[i][\tau][j] = \min_{k \in [1, h_{i-1}]} \{D[i-1][\tau - t_i^j - \sigma(c_k, c_j)][k] + e_i^j + \rho(c_k, c_j)\} \quad (10)$$

D is filed up in a row by row manner and in an order so that all the previous $i-1$ rows are filled when the i^{th} row is being calculated. Note that only those elements corresponding to the Pareto-optimal configuration of b_i is calculated in each vector of $D[i][\tau][\,]$. Finally, the solution quality is decided by $min\{D[n][\tau][j]\}$, for $\tau \in [T_{min}, T_{max}]$ and $j \in [1, h_n]$, which is the lowest value in last row of D.

Complexity Analysis: Our algorithm iterates over all the nodes (1 to n). In other words, the input size of our algorithm is actually the number of execution blocks. In each iteration, all discretized T_i values $(T_{max} - T_{min})$ as well as all Pareto-optimal configuration points (1 to h_i) for current and previous nodes are examined. Hence the time complexity is $O(n \cdot (max\{h_i\})^2 \cdot (T_{max} - T_{min}))$. The memory requirement of our algorithm is determined by the size of D, which stores $n \cdot (T_{max} - T_{min}) \cdot h \cdot sizeof(element)$ bytes. To reduce the memory complexity, in each entry of D, we can simply use minimum number of bits to remember the configuration choice instead of real E_i values. For calculation purposes, two two-dimensional arrays are used for temporarily storing E_i values for current and previous nodes.

Deadline Constraint: To ensure that the solution we find does not violate any task's deadline, during each step of the dynamic programming process, if b_i has deadline constraint, all the entries with T_i value larger than d_i are set to infinity. As a result, in the next step, those entries will be regarded as invalid.

Arrival Time Constraint: In the final solution, we have to guarantee that none of the initial blocks of each task instance starts execution before the task's arrival time as shown in Equation (7). However, since it is possible that one execution block finishes earlier than its very next block (thus creating an idle interval), the entries (each of which is a vector) with $T_i \leqslant a_{i+1}$ in the i^{th} row of D are valid. One important observation is that, for block b_{i+1}, it does not really matter when exactly b_i ends if b_i finishes before b_{i+1}'s arrival time. In other words, the T_i values of these entries have no impact on the decision making in b_{i+1}. Hence, in the final solution, if b_i actually ends before a_{i+1}, the choice we make for b_i must be the one that results in the lowest E_i value.

We partition the i^{th} row into three ranges by the next block's arrive time a_{i+1} and the current block's deadline d_i as shown in Figure 3. The first range, named range A, in which entries with finish time earlier than a_{i+1}, are all valid but not all are needed during decision making. The ones with minimum E_i, for each configuration choice of b_i, are selected and stored in the vector $D[i][a_{i+1}][\,]$. All entries in range A are then set to infinity. By doing this, without losing any precision, we force b_{i+1} to start no earlier than its arrival time. The second range (range B) in which entries with T_i values between a_{i+1} and d_i are all valid for the calculation of next iteration since they make b_{i+1} start after a_{i+1}. The last range are all discarded due to deadline constraint of b_i.

For periodic task set, if each task's deadline is equal to its period, a_{i+1} is always earlier than d_i. It can be proved by contradiction. If a_{i+1} is larger than d_i, it implies that the next job of the task corresponding to b_i arrives before b_{i+1} does. Therefore, there exists a ready-to-execute task between b_i and b_{i+1}, which contradicts the fact that b_{i+1} is the very next execution block of b_i. In cases where a_{i+1} may be after d_i (e.g. for aperiodic task set), range B vanishes and, as a result, the problem essentially becomes two independent subproblems (one consists of blocks before b_i while the other consists of blocks after b_{i+1}, inclusively).

Tradeoff by Time Discretization: As discussed above, the time complexity of our algorithm is dominated by the term $(T_{max} - T_{min})$. A tradeoff can be made between solution quality and algorithm performance by further discretizing the execution time T_i. During the dynamic programming, instead of calculating for every time unit, we can compute in interval

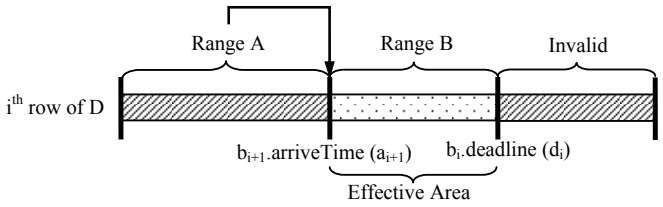

Fig. 3. Ensuring the time constraints.

of multiple units. We define this number of time units as a parameter δ. For example, if $\delta = 2$, every row of D will contain $\lceil \frac{T_{max}-T_{min}}{\delta} \rceil$ vectors which are $\{T_{min}, T_{min}+2, T_{min}+4, ..., T_{max}\}$. The time complexity is reduced to $O(n \cdot (max\{h_i\})^2 \cdot \frac{T_{max}-T_{min}}{\delta})$. By doing this, we actually examine every possible path at a coarser granularity. Our experimental results demonstrate that time discretization only brings minor design quality degradation in terms of energy consumption while the algorithm efficiency can be greatly improved.

V. Experiments

A. Experimental Setup

DCR: To demonstrate the effectiveness of our algorithm on DCR, we use real applications which are selected benchmarks from MediaBench [20], MiBench [21] and EEMBC [22] to form four task sets, each consisting 4 to 7 tasks, as shown in Table I. In order to avoid some task dominating the others in terms of energy consumption, we select the benchmarks such that tasks in the same set have comparable sizes. For each task set, we consider both cases of periodic and aperiodic/sporadic tasks. In the former scenario (periodic tasks), we assign the period and task's worst-case workload so that the system utilization varies in the range of 0.3 to 0.9[4] in incremental step of 0.1. In the later scenario (aperiodic/sporadic tasks), for each task, all the jobs are randomly generated with total accumulative system utilization at any moment under the schedulability constraint (e.g., 1). The job inter-arrival time is generated based on an exponential distribution. The reconfigurable cache architecture we utilized [2] is a four-bank cache with tunable cache sizes of 4KB, 8KB and 16KB, line sizes of 16 bytes, 32 bytes and 64 bytes and associativity of 1-way, 2-way and 4-way. Empirically, there are around 3 to 5 Pareto-optimal cache configurations for conventional applications [5]. We use SimpleScalar [23] to collect the static profiling information including the number of cache accesses, cache misses and clock cycles.

DVS: To evaluate our algorithm for DVS, we consider Marvell's StrongARM [7] as the underlying DVS-enabled processor, which supports four voltage/frequency levels (1.5V/206MHz, 1.4V/192Mhz, 1.2V/162MHz and 1.1V/133MHz). We randomly generate four synthetic task sets, with similar characteristics in DCR.

[4]This is a practical and reasonable range since below 0.3 the solution can be trivially found by selecting most energy-efficient configurations for all tasks.

TABLE I
Task sets consisting of real benchmarks.

Sets	Tasks
Set 1	ospf, susan, pegwit, pktflow
Set 2	cjpeg, epic, dijkstra, FFT, qsort
Set 3	CANRDR01, PUWMOD01, AIFIRF01, BITMNP01, CACHEB01, AIFFTR01
Set 4	stringsearch, ospf, CRC32, pegwit, untoast, qsort, toast

B. Results

Energy Reduction: We compare our algorithm with two heuristics which are applicable to both DVS and DCR, namely Uniform Slowdown and Greedy Repairing, since the techniques proposed in [4] and [5] are only for soft real-time systems and thus not applicable to our case. These two heuristics are adapted from DVS techniques [24] [25]. Generally, in uniform slowdown, we choose the configuration for task τ_i which consumes minimum energy while has equal or less execution time compared to t_i^{base}/η, where t_i^{base} is the execution time under base case and η is the system utilization. In greedy repairing, we first assign the most energy efficient configuration to every task. If the task set becomes unschedulable, we run a greedy repairing phase, during which the next more performance efficient configuration for one of the tasks is selected which leads to minimum ratio of energy increase to system utilization decrease. The process repeats until the task set becomes schedulable. This heuristic is also used in [26]. Note that these two heuristics assign only one configuration per task and are not able to consider variable overhead. Figure 4 (a) and (b) shows the comparison results for both DVS and DCR, respectively. The time discretization parameter δ is set to 1, 2, 4 and 8 milliseconds[5] and leads to 17% of energy savings for DCR and 9% for DVS on average.

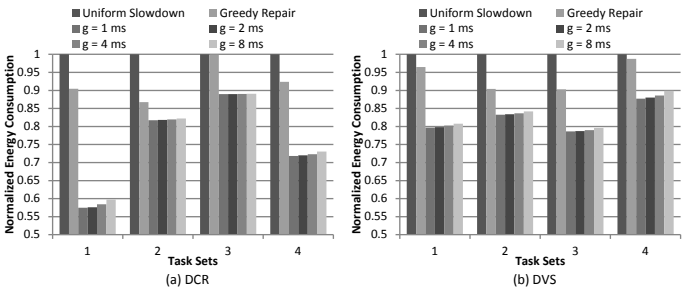

Fig. 4. Energy consumption compared with two heuristics.

Time Discretization Effect: Figure 5 illustrates the flexibility of our algorithm by varying the time discretization. Results are the average over all task sets. δ is increased exponentially from 1 millisecond to 128 milliseconds. The important observation is that, although our algorithm running time is drastically reduced, the design quality (total energy consumption) is only slightly sacrificed and still very close to the case where $\delta = 1ms$. For example, for task set 4 in DCR which has 679 execution blocks in the hyper-period, our

[5]In DCR, since tasks in set 3 has smaller sizes in terms of energy consumption and execution time than other sets, the unit of δ is microsecond.

algorithm gives the solution in 1.5 seconds with $\delta = 128ms$. The energy consumption of this solution is only 7% worse than the one generated with $\delta = 1ms$, which requires 19 seconds of execution time.

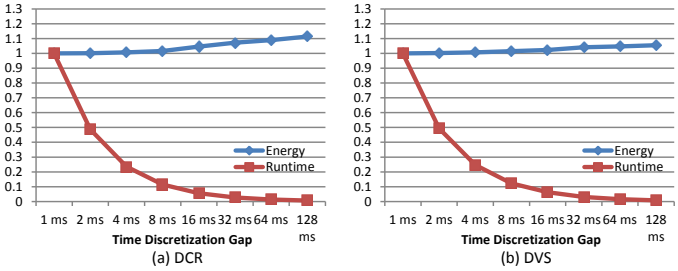

Fig. 5. Time discretization effect (Normalized to $\delta = 1ms$).

Variable Overhead Aware Effect: We compare two different versions of our algorithm: one is aware of variable reconfiguration overhead and the other assumes constant overhead (which is the average of all variable overhead values). For DVS, the variable overhead matrix is generated so that each value depends on and is in proportion to how much voltage/frequency is increased or decreased. For DCR, the matrix is similarly generated except that the overhead for tuning the cache capacity from one level to another is 10 times larger than tuning the line size and associativity. Therefore, the actual overhead is the sum of all three cache parameters.

Figure 6 demonstrates that effectively utilize the variable overhead can lead to substantial energy saving improvements for all task sets in DCR. Same observation can be made for DVS scenario. However, variable overhead awareness in DCR can lead to averagely 10% more energy savings than in DVS, which is because the size and variability of DCR's design space is much larger than DVS. Note that our approach is beneficial for any kind of optimization problem based on reconfiguration – where the runtime overhead is substantial.

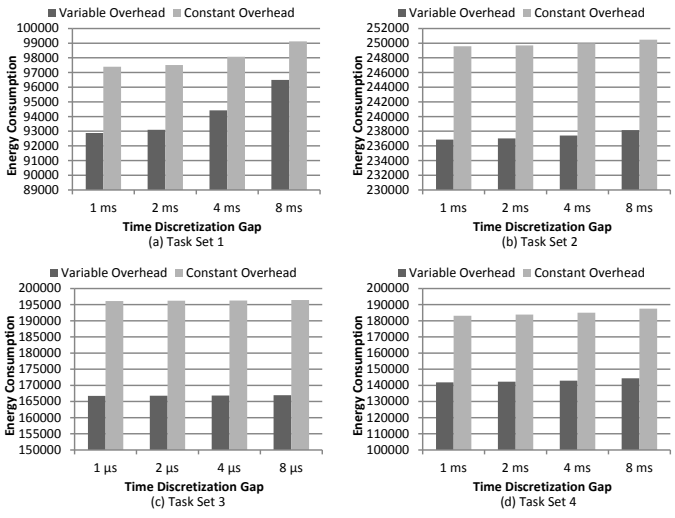

Fig. 6. Variable overhead aware effect in DCR.

VI. Conclusion

Dynamic reconfiguration is widely used for improving energy efficiency in microprocessor systems. We proposed a general and flexible algorithm for both cache reconfiguration and voltage scaling in multitasking systems with timing constraints. Our approach has the following advantages. First, it can lead to more energy savings than inter-task manner DVS/DCR techniques. Secondly, it can effectively take variable reconfiguration overhead into consideration. Finally, our algorithm can be flexibly parameterized so that only slight solution quality degradation can be traded for drastically reduced running time requirement. It is also independent of task characteristics and scheduling policy. Extensive experiments demonstrates the effectiveness of our approach.

References

[1] A. Malik et al., "A low power unified cache architecture providing power and performance flexibility," *ISLPED*, 2000.
[2] C. Zhang et al., "A highly configurable cache for low energy embedded systems," *ACM TECS*, vol. 6, pp. 362–387, 2005.
[3] A. Gordon-Ross et al., "A self-tuning configurable cache," *DAC*, 2007.
[4] W. Wang et al., "SACR: Scheduling-aware cache reconfiguration for real-time embedded systems," *VLSI Design*, 2009.
[5] W. Wang et al., "Dynamic reconfiguration of two-level caches in soft real-time embedded systems," *ISVLSI*, 2009.
[6] W. Wang et al., "Leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in real-time systems," *VLSI Design*, 2010.
[7] Marvell, *Marvell StrongARM 1100 processor*, www.marvell.com.
[8] H. Aydin et al., "Power-aware scheduling for periodic real-time tasks," *IEEE TC*, vol. 53, no. 5, pp. 584–600, May 2004.
[9] D. Shin et al., "Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems," *ASP-DAC*, 2004.
[10] X. Zhong et al., "System-wide energy minimization for real-time tasks: Lower bound and approximation," *ICCAD*, 2006.
[11] R. Jejurikar et al., "Energy aware non-preemptive scheduling for hard real-time systems," *ECRTS*, 2005.
[12] J. Chen et al., "1 + ε approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor," *EMSOFT*, 2005.
[13] D. Shin et al., "Optimizing intratask voltage scheduling using profile and data-flow information," *IEEE TCAD*, vol. 26, pp. 369–385, 2007.
[14] W. Wang et al., "PreDVS: Preemptive dynamic voltage scaling for real-time systems using approximation scheme," *DAC*, 2010.
[15] W. Wang et al., "Temperature- and energy-constrained scheduling in multitasking systems: A model checking approach," *ISLPED*, 2010.
[16] J.-J. Chen et al., "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms," *RTCSA*, 2007.
[17] V. Swaminathan et al., "Network flow techniques for dynamic voltage scaling in hard real-time systems," *IEEE TCAD*, vol. 23, no. 10, pp. 1385–1398, Oct. 2004.
[18] HP, *CACTI, HP Labs, CACTI 5.3*, http://www.hpl.hp.com/.
[19] S. M. Martin et al., "Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads," *ICCAD*, 2002.
[20] C. Lee et al., "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," *Micro*, 1997.
[21] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite," *WWC*, 2001.
[22] EEMBC, *EEMBC, The Embedded Microprocessor Benchmark Consortium*, http://www.eembc.org/.
[23] D. Burger et al., "Evaluating future microprocessors: The simplescalar tool set," University of Wisconsin-Madison, Tech. Rep., 1996.
[24] H. Aydin et al., "Dynamic and aggressive scheduling techniques for power-aware real-time systems," *RTSS*, 2001.
[25] C. Rusu et al., "Maximizing the system value while satisfying time and energy constraints," *RTSS*, 2002.
[26] R. Jejurikar et al., "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," *ISLPED*, 2004.