Energy-Efficient Primary/Backup Scheduling Techniques for Heterogeneous Multicore Systems

Abhishek Roy, Hakan Aydin
Department of Computer Science
George Mason University
Fairfax, Virginia 22030
aroy6@gmu.edu, aydin@cs.gmu.edu

Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
dakai.zhu@utsa.edu

***Abstract*—In this paper, we consider energy-efficient and fault-tolerant scheduling of real-time tasks on heterogeneous multicore systems. Each task consists of a main copy and a backup copy which are scheduled on different cores, for fault tolerance purposes. Our framework deliberately delays the backup tasks in order to cancel them dynamically when the main task copies complete successfully (without faults). We identify and address two dimensions of the problem, i.e., partitioning tasks and determining processor voltage/frequency levels to minimize energy consumption. Our experimental results show that our proposed algorithms' performance levels are close to that of an ideal solution with optimal (but computationally prohibitive) partitioning and frequency assignment components.**

I. Introduction

Energy management remains a crucial component for the design and implementation of embedded systems, including those deployed in safety-critical and time-critical applications such as those in industrial control, avionics, and high-confidence medical systems. Recently, *heterogeneous (asymmetric)* multicore systems have been embraced by the industry due to their power-efficient design and the flexibility they offer in dealing with different types of workloads. These systems typically combine the high-performance "big" cores with "little" cores that consume less power, at the cost of providing more modest performance. ARM's big.LITTLE systems that include out-of-order and fast cores (such as ARM Cortex-A15) and in-order, energy-efficient cores (such as ARM Cortex A-7) are among the most well-known examples [1].

The main idea in deploying heterogeneous multicore systems is to execute the workload at hand by the core most suitable for the current performance objective (high performance or energy savings). The research community has recently addressed several aspects of heterogeneous multicore systems with a multi-dimensional effort [2], [3].

Another increasingly important dimension, in particular for safety-critical embedded systems, is *reliability*. Those systems should be able to detect, and recover from, various types of faults in a timely manner [4]. The majority of run-time faults are categorized as *transient*, in that they are short-lived – they are typically induced by the phenomena such as electromagnetic interference and cosmic rays. However, they result in erroneous task computation, and typically a recovery task, in the form of an alternative task or a re-execution, is invoked [4], [5], [6]. It has been reported that the transient faults occurrence rate is increasing, in particular due to the use of aggressive power management techniques such as near-threshold voltage operation [7]. On the other hand, in case of *permanent* faults, a processing core becomes unavailable – this is typically due to the aging effects, harsh environmental conditions, and manufacturing defects. Tolerating *permanent* faults requires the deployment of additional hardware (such as another core) that can take over the execution of the tasks originally allocated to the affected unit [4].

In this paper, we propose implementing a fault-tolerant framework on heterogeneous dual-core systems while keeping the energy consumption at a minimum level. Specifically, we consider a set of real-time tasks where each task consists of a *primary (main)* copy and a *backup* copy, that are allocated to different cores. This allows the system to tolerate the permanent fault of any single core, since each processor has exactly one copy of each task (primary or backup) [4]. Moreover, the transient faults detected in all primary tasks can be recovered from by the execution of the respective backup task. Our work differs from the existing so-called *standby-sparing* frameworks [8], [9], in that: i.) we allow scheduling a mix of primary and back-up tasks on each processor, and, ii.) we consider heterogeneous multicore systems. Although in [10] we investigated a similar problem in the context of heterogeneous dual-core systems, the focus was again the standby-sparing configuration, and the mixing of primary and backup copies on a given core was not considered.

To keep the energy consumption under control, the backup tasks are delayed as much as possible on their corresponding processors, because a backup can be canceled as soon as the corresponding primary completes successfully (i.e., without a fault). This also gives a chance to apply Dynamic Voltage and Frequency Scaling (DVFS) with maximum efficiency during the execution of the primary tasks on each core. We develop and propose schemes, i.) to partition all primary and backup tasks, and, ii.) assign frequency (speed) to all the primary tasks to minimize the energy consumption, while meeting timing and fault tolerance constraints.

Our experimental results suggest that the list-scheduling based partitioning techniques, coupled with a speed assignment approach that dynamically avoids the overlaps with the

978-1-5386-3470-7/17/$31.00 ©2017 IEEE

backups, exhibit superior performance which is close to the theoretical lower bound in terms of energy consumption. Our framework directly incorporates a salient feature of heterogeneous cores, namely the fact that the energy consumption and execution time figures of different tasks scale by different ratios when executed on different cores [11].

II. System Model and Assumptions

A. Platform and Application model

We consider a heterogenous dual-core system with a high-performance (big) core and a low-power (little) core. The high-performance and low-power cores are denoted by HP and LP, respectively, throughout the paper. The cores are assumed to have the same instruction set architecture, implying that the executable of a task can run on either core.

The workload consists of n independent real-time tasks $\{\tau_1, ..., \tau_n\}$ that will be executed on this dual-core platform. We assume the *frame-based execution* model [6], [12] in which all tasks have the same period, which is equal to the common deadline D. Each processing core is equipped with the *Dynamic Voltage and Frequency Scaling (DVFS)* feature that allows changing the frequency (processing speed) at run-time. Moreover, the *Dynamic Power Management (DPM)* feature allows a given core to switch to a *low-power (idle)* mode when it is not actively executing tasks.

A task τ_i that requires C_i number of cycles on a given core may take up to $W_i = C_i/f$ units of execution time on that core, if executed at the frequency level f. Due to the architectural differences, a task's required number of cycles, and hence execution time, can be different on the HP and LP cores. Therefore, we use superscripts HP and LP to denote the variables on the HP or the LP core (C_i^{LP}, W_i^{LP}, C_i^{HP}, W_i^{HP}). We define the *nominal utilization* of a task τ_i as (C_i^{HP}/D). The maximum frequency levels supported by the HP and LP cores are denoted by f_{max}^{HP} and f_{max}^{LP}, respectively. We assume $f_{max}^{HP} = 1.0$, and normalize all other frequency values with respect to that value.

Associated with each (primary) task τ_i, there is a backup task B_i with exact same timing parameters as those of τ_i. τ_i and B_i are allocated to different cores: should a permanent fault affect any of the processing cores, the alternative core can take over and finish the workload before deadline. When a primary copy completes, the *acceptance* (or, *sanity*) tests [4] are performed to check the existence of errors induced by *transient* faults. If a fault is not detected, the corresponding backup copy (or, its remaining part) on the other core is canceled. Otherwise, the backup copy runs to completion.

B. Power Model

The power consumption characteristics of the HP and LP cores differ by design. For any processing core, the dynamic power consumption of an executing task τ_i is modeled as, $P_i(f) = a_i f^3 + \alpha_i$, where a_i denotes the switching capacitance, α_i denotes the frequency-independent power consumption, and f is the processing frequency of the task. Due to the asymmetry of the cores, these parameters are different for each core and again we use superscripts HP and LP to denote core-specific power parameters (e.g., P_i^{HP}, α_i^{HP}).

Each core executes tasks in the *active* state, dissipating power as determined by the characteristics of the current task and processing frequency. When a core does not execute tasks, it remains in the low-power *(idle)* state. The low-power (idle) power consumption of the high-performance and low-power cores are denoted by P_{idle}^{HP} and P_{idle}^{LP}, respectively. We assume those figures include the static power consumption of the corresponding core as well. The energy consumption during a time interval is given by the aggregate power consumption during the same interval.

Existing research indicates that scaling down the frequency below a certain threshold is no longer effective for saving energy, due to the impact of the frequency-independent power component [12]. This threshold frequency, known as the *energy-efficient frequency (f_{ee})* can be derived through analytical techniques [12].

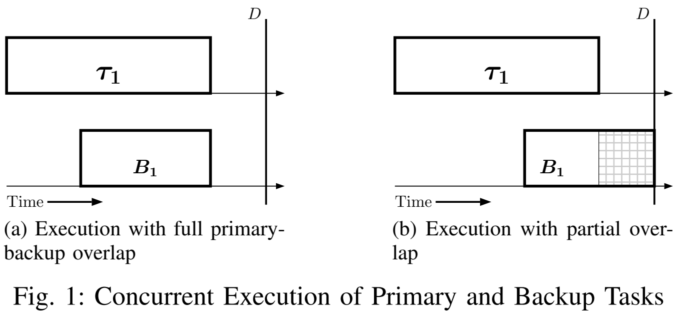
Problem Statement: Given a set of real-time tasks and a heterogeneous dual-core system, minimize the energy consumption by determining

1) The allocation of tasks such that the primary and backup copy of each task are assigned to different cores, and,
2) The processing frequency (speed) assignment to individual tasks.

In the following section, we investigate these two interconnected dimensions and propose several efficient schemes.

III. Proposed Schemes

Before describing the specific algorithms that we propose, we present a number of general principles that guide our solution framework. To start with, in general, the concurrent execution of a primary task and its backup, though possible, is not desirable because it incurs the full energy cost of the back-up execution (Figure 1a). However, in case when the backup's execution can be delayed, by the time the primary completes successfully, its remaining part can be cancelled (Figure 1b)[1].



(a) Execution with full primary-backup overlap
(b) Execution with partial overlap

Fig. 1: Concurrent Execution of Primary and Backup Tasks

This further suggests that on a given core, all the primary tasks must execute *before* the backup tasks allocated to that core. Moreover, provisions are made to execute all *backup tasks* at the maximum frequency on their respective cores,

[1]Throughout the paper, we show the cancelled part of the backup tasks by dashed patterns in all the figures.

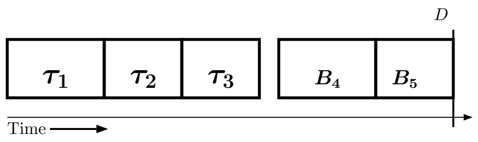


Fig. 2: Canonical Execution Order

should there be a need – obviously this choice minimizes their overlap with their respective primary tasks on the *other* core, and in addition, since faults are rare events, the full speed execution of the backups has only a minimal impact on the average-case energy consumption. Clearly, this choice also leaves maximum slow-down opportunities for the primary tasks scheduled on that core through DVFS.

Thus, we define the *canonical execution order*, in which on a given core all primary tasks are started as soon as possible, whereas backup tasks are delayed as much as possible subject to the deadline constraints, and executed at the maximum frequency if needed. Figure 2 shows a *canonical execution* on a single processing core to which three primary tasks (τ_1, τ_2, τ_3), and two backup tasks (B_4 and B_5) are assigned. In the rest of the paper, we commit to this canonical execution order to execute primary and backup copies of tasks on all cores, once the partitioning is done.

A related framework is the so-called *energy-aware standby-sparing technique*, in which, one of the cores is designated for the primary tasks and the other one for the backup tasks exclusively [8], [9], [10]. In our framework, however, for maximum flexibility, we allow scheduling the primary and backup copies on both cores, when possible – for that reason, we call our framework *mixed primary backup (MPB) assignment.* The schemes we propose consist of *task partitioning* and *speed (frequency) assignment* phases which are described next.

A. Task partitioning

Task partitioning, in general, is an intractable problem; however, a well-known approach is based on the *list-scheduling* technique. We first describe two variants based on list scheduling for our task partitioning phase.

List-scheduling with Primaries (LSP). In this algorithm, we consider the primary copies of the tasks and employ list-scheduling algorithm to allocate them. First, the tasks are ordered according to their decreasing nominal utilizations. Then, each primary task is placed on a processing core that has the maximum *free capacity* after the placement. *Free capacity* on a core is defined by ($f_{max} - \sum_{\tau_i \epsilon \Gamma_p} \frac{C_i}{D}$), where Γ_p is the set of all primary tasks assigned to that core, augmented by the task under consideration. f_{max} and $\{C_i\}$ values are defined in the context of the core under consideration. Observe that the first few primary tasks will always go to the HP core, until its free capacity matches that of the LP core. Once the distribution of the primary tasks is complete, a backup copy for each primary task is allocated to the alternate core. Also, at each stage of the primary task allocation, the feasibility of both cores, in terms of time constraints, are checked.

TABLE I: Example Task Set 1

	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}
τ_1	13.2	30.4	14.63	5.58
τ_2	10.7	19.4	11.77	3.56
τ_3	10.6	18.8	11.66	3.45
τ_4	10.2	18.9	11.22	3.47

We illustrate the behavior of the algorithm on an example task set given in Table I. The table gives task execution times (in ms), and energy consumption (E_i^{HP}, E_i^{LP}) on both cores (in mJ), under respective maximum frequencies. The 4-task set is scheduled on a dual core system with $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$, and for all tasks, $a_i^{HP} = 1.0$, $a_i^{LP} = 0.3$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.03$. For demonstration, we use a simple runtime policy (called *static* policy) in which, each primary task is slowed down as much as possible without violating the frame deadline. The *canonical execution order* is adopted on each core.

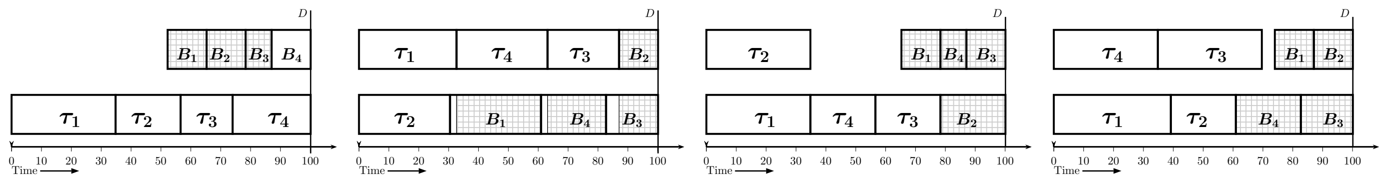
Figure 3b shows the task allocation under this scheme for our example task set in Table I. The first task, τ_1 is allocated to the HP core, because it has the most free capacity among the two cores. τ_2 is allocated to the LP core whose free capacity is higher at that time. Similarly, tasks τ_3 and τ_4 are allocated to the HP core. It should be noted that, in contrast to the standby-sparing configuration shown in Figure 3a (which uses the partitioning method *SlowerP*, one of the best-performing scheme in [10]), the extent of primary-backup overlapped executions is much less in the LSP solution.

List-scheduling with Backups (LSB). This algorithm works in the same way as LSP, but this time, the backup copies of the tasks are considered while partitioning. Once the backup copies are distributed, their corresponding primary copies are allocated to the respective alternate processing cores. By its very nature, this algorithm tends to allocate a few initial primary tasks to the LP core, before their backups are allocated to the HP core thanks to the LSB rule.

Figure 3c shows the task allocation under this scheme for our example task set in Table I. This partitioning is a mirror image of the LSP partitioning. It can be noted that, all primary-backup overlapped executions are avoided.

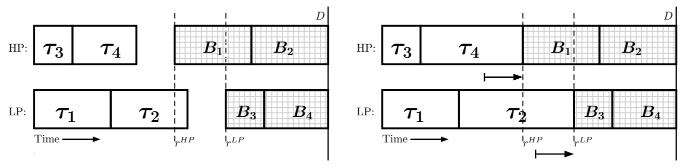
Fixed-Threshold Algorithm (FTH). In this algorithm, the primary tasks are at first ordered according to their decreasing nominal utilizations and processed one by one. Tasks are assigned to the LP core, as long as its load does not exceed a pre-defined *threshold* value. Otherwise the primary task is assigned to the HP core. After each primary task assignment, its backup copy is allocated to the counterpart core. The *threshold* value can assume any value between 0.0 and 1.0.

For our example Task Set 1, this heuristic produces the task-allocation shown in Figure 3d when the threshold value is 0.6. Tasks τ_1 and τ_2 are allocated to the LP core. When task τ_3 is processed, the total used capacity on the LP core exceeds 60% if it is assigned to the LP core. Therefore, it is assigned to the HP core. Similarly, τ_4 is allocated to the HP core.


D
B_1 B_2 B_3 B_4
τ_1 τ_2 τ_3 τ_4
τ_1 τ_4 τ_3 B_2
τ_2 B_1 B_4 B_3
τ_2 B_1 B_4 B_3
τ_1 τ_4 τ_3 B_2
τ_4 τ_3 B_1 B_2
τ_1 τ_2 B_4 B_3
0 10 20 30 40 50 60 70 80 90 100
Time

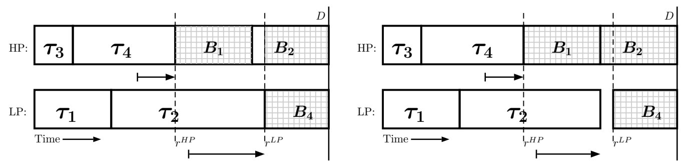
(a) Standby-sparing [10] (b) List-scheduling with primaries (c) List-scheduling with backups (d) Fixed-threshold algorithm

Fig. 3: Task partitioning algorithms



(a) Initial partitioning (b) After speed assignment

Fig. 4: Static Speed Assignment



(a) Dynamic Backup Cancellation (b) Dynamic Backup Cancellation with Minimum Overlap

Fig. 5: Dynamic Policies

B. Speed assignment

Once the task partitioning phase is complete, the next step is to determine the speed (frequency) of the primary tasks on each core, while committing to the canonical execution order. Speed assignment to the primary tasks is critical not only because it determines directly the primary's energy consumption, but also indirectly, that of the corresponding backup whose overlap extent may change as a result of that assignment. Below we propose three speed assignment policies.

Static Speed Assignment (SSA). Figure 4 illustrates the basic principles of the SSA policy. The scheme reserves capacity for each allocated backup task (which runs at the maximum frequency of the core), and assigns a latest-start-time to each of them such that no deadlines are missed. In Figure 4a, r^{HP} and r^{LP} denote the latest start time for the first backup task on the HP and LP cores, respectively. Primary tasks are slowed down as much as possible, subject to the energy-efficient frequency bound (f_{ee}). Letting r denote the start time of the first backup task on a specific core, and Γ_P denote the set of all primary tasks on that core, then, the common frequency that finishes all these primary tasks before time r is given by $f_U = (\sum_{\tau_i \epsilon \Gamma_P} C_i)/r$. Then, each primary task τ_i is assigned the frequency $f_i = Max(f_i^{ee}, f_U)$. Figure 4b shows the extended execution times for primary tasks, derived through this principle.

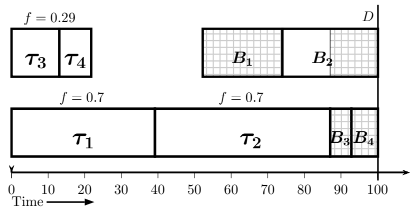
Dynamic Backup Cancellation (DBC). In this scheme, as in SSA, the processing capacity is reserved for backup tasks and primary copies are slowed down as much as possible, subject to the energy-efficient frequency. However, the speed assignment routine is re-invoked at runtime: each time a primary task completes without fault, the reserved capacity for its backup copy is deallocated and used to further slow down the next primary tasks on that core. For example when τ_3 finishes without error, the reserved capacity for B_3 on the LP core is reclaimed to further slow-down τ_2 (Figure 5a). Note that, this introduces some overlapped execution for B_2. In general, when task τ_i is about to run at time t, its speed is chosen as $f_U = (\sum_{\tau_i \epsilon \Gamma} C_i)/(r - t)$, where Γ is the set of unfinished primary tasks on the same core, and r represents the earliest start time among the unfinished backup tasks, again on the same core. When a primary task completes without error, the earliest backup activation time on the alternate processing core is updated at runtime. The chosen speed value is subject to the energy-efficient frequency, therefore, for each task τ_i, the speed is set to $f_i = Max(f_i^{ee}, f_U)$.

Dynamic Backup Cancellation with Minimum Overlap (DMO). This scheme works as the DBC scheme; but when setting the speed of the primary tasks at run-time, it attempts to minimize the overlapped-execution with back-ups. As shown in Figure 5b, when DVFS is applied to τ_2 at the beginning of its execution, it is not maximally slowed down; instead, the overlapped execution with B_2 is avoided by running somewhat faster than the DBC policy. Under this policy, the speed of τ_i is chosen to be $f_i = Min(f^{max}, f_i^*)$ where $f_i^* = \frac{C_i}{r_i - t}$, where r_i is the latest time the backup copy of τ_i can be activated (on the alternative core) without violating any deadlines, and t is the current time. This speed is subject to the deadline constraint and the energy-efficient speed, therefore, f_i is updated as $f_i = Max(f_i, f_i^{ee}, f_U)$. In this scheme, f_U is re-computed with a dynamically updated r value as in the DBC scheme.

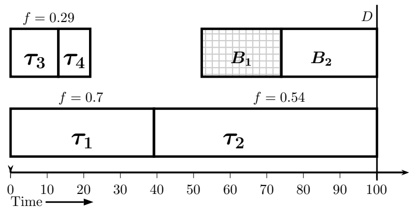
TABLE II: Example Task Set 2

	W_i^{HP}	W_i^{LP}	E_i^{HP}	E_i^{LP}
τ_1	20.3	36.8	22.33	6.76
τ_2	19.1	39.4	21.01	7.23
τ_3	4.3	10	4.73	1.84
τ_4	1.5	3.02	1.65	0.55

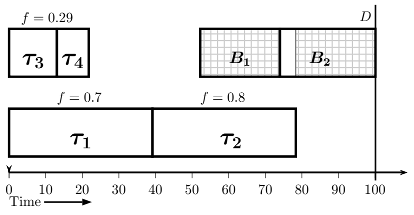
To contrast the impact of these schemes, we use the 4-



(a) Static speed assignment (SSA)



(b) Dynamic backup cancellation (DBC)



(c) Dynamic cancellation with minimum overlap (DMO)

Fig. 6: Execution under different schemes

task set in Table II with $a_i^{HP} = 1.0$, $a_i^{LP} = 0.3$, $\alpha_i^{HP} = 0.1$ and $\alpha_i^{LP} = 0.03$ for each task. The task set is executed on a dual core system with $f_{max}^{HP} = 1.0$ and $f_{max}^{LP} = 0.8$. We also assume $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$. Figure 6a shows the execution of the task set under LSB partitioning and static speed assignment. The HP core (at the top) uses the energy-efficient frequency for tasks τ_3 and τ_4, and the LP core (at the bottom) is slowed down maximally ($f = 0.7$) so that all backup copies (B_3 and B_4) can make their deadline. The overall energy consumption is 24.7 mJ.

Figure 6b shows the execution of the same task set under LSB partitioning and DBC policy. The scheme reclaims the reserved capacity for the backup copies B_3 and B_4 whose primaries complete without fault, and uses this capacity to further slow down the primary task τ_2 to speed $f = 0.54$. However, this introduces overlapped execution for B_2, and in this case, hurts the energy savings. The overall energy consumption of this system is 36.7 mJ. Finally, Figure 6c shows the execution under LSB partitioning and DMO runtime policy. Although this scheme could use all the reclaimed capacity from B_3 and B_4, it runs τ_2 at the maximum speed of the LP core ($f = 0.8$) to minimize the overlap with B_2. This execution yields an overall energy consumption of 20.2 mJ, which is 18% lower than that of the static policy.

IV. Experimental Evaluation

We evaluated the energy consumption performance of the proposed algorithms in a discrete event simulator. We simulated dual core systems with $f_{max}^{HP} = 1.0$ and f_{max}^{LP} varied from 0.6 to 1.0. Due to space limitations, we will show the results for $f_{max}^{LP} = 0.8$, and analyze the impact of varying f_{max}^{LP} separately in Section IV-C.

It is known that the power parameters and required number of cycles for different tasks scale differently on heterogeneous systems [11]. Therefore, as in [10], we define $tscale_i = \frac{C_i^{LP}}{C_i^{HP}}$, which models how execution time changes on the LP core for a given task, τ_i. Moreover, following [10], we define $pscale_i$ to be the ratio of power consumption of τ_i on the LP core to that on the HP core. Therefore, $pscale_i = \frac{P_i^{LP}}{P_i^{HP}}$, which is also assumed to be the same as $\frac{a_i^{LP}}{a_i^{HP}} = \frac{\alpha_i^{LP}}{\alpha_i^{HP}}$.

For each experiment, the simulator generates a task set containing n tasks, and a given total utilization, U. The utilization value is calculated with respect to the LP core (which is more constrained in terms of performance) and normalized considering its maximum speed. Hence, $U = (\sum \frac{C_i^{LP}}{D})/f_{max}^{LP}$. Based on the target U, we use the *RandFixedSum* algorithm [13] to assign a random utilization (according to uniform distribution) to each task such that the total utilization equals U. We set the frame deadline $D = 100ms$. Next, for each task a $tscale_i$ and a $pscale_i$ value are chosen randomly within ranges suggested in [11]. Specifically, $1.4 \leq tscale_i \leq 2.3$ and $1.4 \leq 1/(tscale_i * pscale_i) \leq 2.1$ hold. We assume for all tasks, $a_i^{HP} = 1.0$ and $\alpha_i^{HP} = 0.1$. In addition, $P_{idle}^{HP} = 0.05$ and $P_{idle}^{LP} = 0.02$ for all experiments.

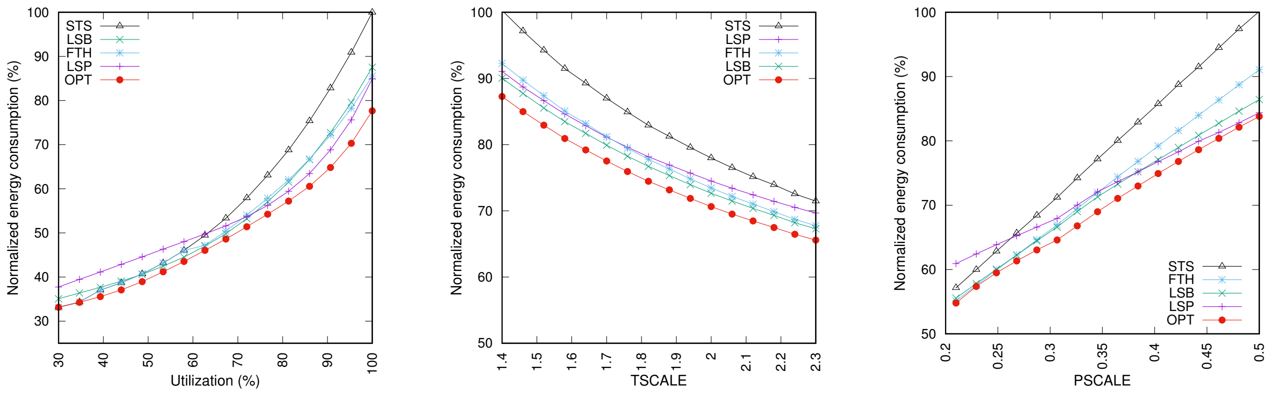
Each generated task set is partitioned upon the HP and LP cores according to one of the proposed partitioning algorithm. For every partition obtained in this way, we simulate the execution according to the speed assignment policies that we suggested, and record the energy consumption. Every combination of a partitioning scheme and a speed assignment algorithm gives us a valid overall algorithm, whose name is indicated by the concatenation of the member schemes (e.g., LSP-SSA, FTH-DMO). We use task sets with $n = 10$ in all the results shown, but we discuss the impact of varying the number of tasks in Section IV-C. Every reported data point is the average of 3000 runs.

We report the average energy consumption in fault-free executions, since faults are very rare events. The obtained energy consumption numbers are normalized with respect to the maximum energy consumption (observed in the considered parameter spectrum) of a standby-sparing system with static speed assignment and in which all the primary copies are allocated to the LP core [10].

Due to the multiple dimensions of the problem and large number of scheme combinations, in our evaluation, we will adopt a hierarchical approach. We will first discuss the performance of the partitioning algorithms by fixing the speed assignment policy. Next, we will compare the performance of the proposed speed assignment policies, and also investigate the impact of the chosen threshold value on the FTH algorithm. Finally, we show the effect of the maximum speed of the LP core and the effect of the number of tasks.

A. Evaluation of Partitioning Algorithms

We implemented the following partitioning schemes in our simulator:



(a) Impact of utilization (b) Impact of tscale (c) Impact of pscale

Fig. 7: Performance of partitioning algorithms

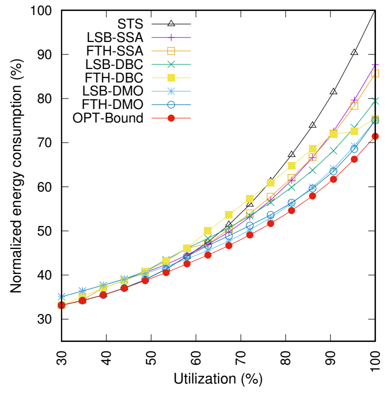
- List-scheduling with Primaries (LSP)
- List-scheduling with Backups (LSB)
- Fixed-threshold Algorithm (FTH)
- Standby-sparing (STS)
- Optimal Partitioning (OPT)

The optimal partitioning we show in the plots is obtained by exhaustively enumerating all possible task allocations, and measuring their runtime energy consumption, then choosing the best. This is implemented by the exhaustive search which becomes impractical when the number of tasks grows beyond 15. The STS algorithm is adopted from the *SlowerP* scheme in [10], because it is shown to be the best-performing one in its respective context. The threshold value for the FTH algorithm is fixed as 0.6. The energy consumption of the partitioning algorithms is shown using the static speed assignment algorithm (SSA); we obtained similar trends with the other (DBC and DMO) algorithms.

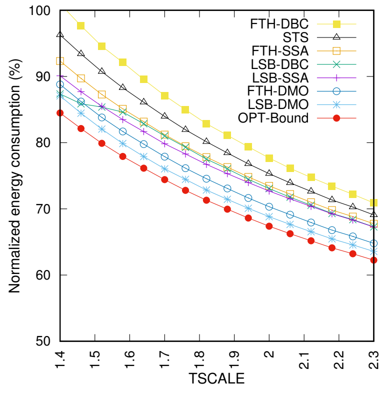
Impact of Utilization. In Figure 7a, we show the impact of utilization on normalized energy consumption. When the utilization is low, the FTH algorithm's performance approaches the optimal one, suggesting that allocating all primary tasks to the LP core, and all the (delayed) backups to the HP core is the best strategy. This is because under low load, LP can finish the primary workload quickly and in a power-efficient way, allowing the backup tasks to get cancelled on the HP core early. This is evident for the STS scheme too, because it allocates all the primary workload to one core as well. As the load increases, FTH drifts from the optimal scheme and LSB becomes a comparable scheme. This is due to the fact that, as the load grows, a more balanced partitioning is preferable which can allow a suitable distribution of the reserved space for backup copies such that their activation is seldom needed. Both FTH and LSB give relatively balanced partitionings, but LSB generally allocates more primary copies to the LP core, with an energy advantage. LSP scheme, performing very poorly on the low-load case, starts to outperform both LSB and FTH when the utilization exceeds 80%, and comes within 5% of the optimal scheme. For heavy load, executing primary copies on the HP core is preferable because in this case, the backup copies cannot, in general, get cancelled and executing them at the maximum speed of the LP core is preferable to executing them at the maximum speed of the HP core. For the same reasons, STS performs the worst for heavy load cases.

Impact of $tscale$. Figure 7b shows the impact of $tscale$ on the performance of the partitioning algorithms. $tscale$ is varied within the range of 1.4 to 2.3, which is obtained from [11]. In general, larger $tscale$ values indicate that tasks take much longer to complete on the LP core, despite its power-efficiency. In these experiments the utilization is fixed at 70%, and therefore, increasing $tscale$ implies additional unused capacity on the HP core. We see that LSB performs consistently within 3% of the optimal scheme throughout the entire range of $tscale$. This is because executing the primary copies of the workload on the power-efficient core results in less energy consumption, and LSB tends to allocate primary workload to the LP core. LSP, on the other hand, has a tendency to assign primary workloads to the HP core, and in general, it lags behind LSB. FTH comes very close to the performance of LSB as $tscale$ increases.

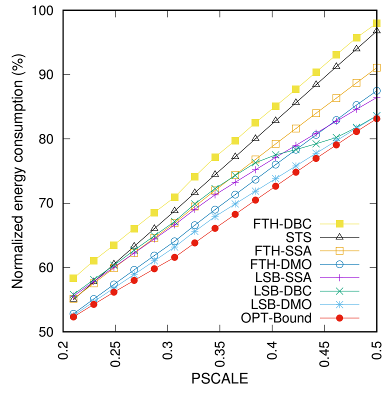
Impact of $pscale$. Figure 7c shows the impact of $pscale$ on the performance of the partitioning algorithms. When the LP core is very power-efficient, i.e., $pscale$ is low, FTH and LSB come very close to optimal scheme. This is because at the fixed 70% system load, FTH assigns most of the primary workload on the LP core, and that helps saving energy. As $pscale$ grows, FTH drifts away from the optimal scheme the most, because it is no longer efficient to use the LP core for most of the primary workload. However, LSB can still perform within 5% of the optimal scheme, because it produces a more balanced partitioning with a bias to allocate the primary tasks to the LP core. LSP, which produces a balanced partitioning with a bias to assign the primary tasks to the HP core, performs poorly for low $pscale$, but starts to outperform LSB for $pscale$ greater than 0.4 and comes 2% of the optimal scheme.



(a) Impact of utilization



(b) Impact of tscale



(c) Impact of pscale

Fig. 8: Performance of the speed assignment algorithms

B. *Evaluation of Speed Assignment Algorithms*

We implemented the following speed assignment policies.

- Static Speed Assignment (SSA)
- Dynamic Backup Cancellation (DBC)
- Dynamic Backup Cancellation with Minimum Overlap (DMO)
- Bound

The *Bound* algorithm is implemented as a yardstick speed assignment algorithm. After partitioning the tasks the executions slots are still reserved for backup tasks – those slots are dynamically released (as in DBC), but no extra energy consumption is recorded for the overlapped execution of the backup tasks at run-time. Since the backup executions essentially incur zero energy cost, no speed assignment algorithm can outperform *Bound*. We matched *Bound* with the exhaustive search based *Optimal* partitioning algorithm, obtaining a combined scheme denoted by *OPT-Bound*, which gives the lower bound on the performance of any realistic MPB algorithm. Given the large number of partitioning/speed assignment scheme combinations, for other schemes, we are showing only the results we obtained with the best performing partitioning algorithms, namely LSB and FTH. We are using the *Overlap-Aware* speed assignment scheme for STS, as it is shown to be the best performing scheme for standby-sparing in [10].

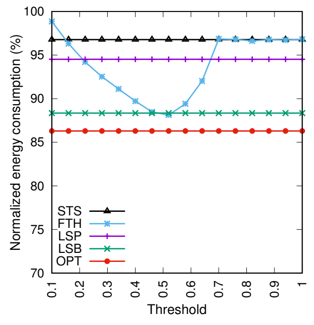
Impact of Utilization. In Figure 8a, we see that both FTH-DMO and LSB-DMO perform within 2% of Opt-Bound. This is because dynamically reclaiming the capacity for backup tasks and minimizing overlap while applying DVFS is a very effective strategy, as done within DMO. This is also true for STS at low-load, because it allows some carefully calculated overlapped execution. As the load increases, STS drifts away from Opt-Bound the most, because it has the restriction that it cannot allocate primary and backup copies on the same processor. FTH-DBC and LSB-DBC perform poorly for moderately loaded systems due to the large overlapped executions that it creates. However, for heavy load, backup copies need to run until deadline anyway, therefore the performance of the DBC scheme improves. Both FTH-SSA and LSB-SSA offer decent performance levels unless the load is very high.

Impact of $tscale$. As we change $tscale$ value (when the load is fixed at 70%), LSB-DMO performs the best and stays within 3% of Opt-Bound (Figure 8b). The next best performing scheme is FTH-DMO. This again suggests the superiority of DMO thanks to its dynamic but moderately aggressive approach in applying DVFS while avoiding overlaps. The plot also shows that FTH-DBC performs the worst, and LSB-DBC performs the worst among all the LSB algorithms. This is because DBC aggressively slows down a task without regard to the overlapped execution.

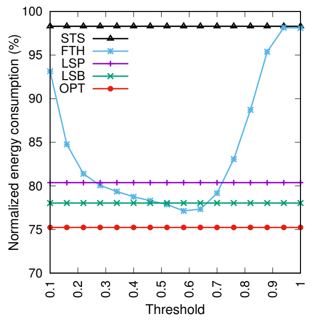
Impact of $pscale$. Varying $pscale$ yields similar trends (Figure 8c). LSB-DMO and FTH-DMO perform the best, within 3% of *Opt-Bound*, by exploiting the overlap avoidance strategy of DMO. LSB-DMO's performance, however, decreases as the LP core becomes less power-efficient ($pscale$ increases). This is because, with less power-efficient LP core, it is no longer favorable to assign primary workload to the LP core up to a threshold. Due to the aggressive frequency scaling of the DBC scheme, FTH-DBC performs the worst throughout the entire spectrum. LSB-DBC, performing poorly for low $pscale$, starts to improve when $pscale$ is greater than 0.4, and comes within 1% of Bound. This is because when the LP core is less power-efficient, slowing it down as much as possible proves helpful from the energy consumption perspective.

C. *Additional Results*

Impact of the threshold value in the FTH algorithm. The Fixed-Threshold (FTH) algorithm works by allocating all primary tasks to the LP core until a *threshold* utilization is reached. Figure 9a shows the impact of the threshold value on a system that is 70% loaded and with DMO policy. The results indicate that the energy consumption of *FTH* decreases as we increase the threshold value, and at about 0.45, it outperforms the otherwise best performing algorithm,

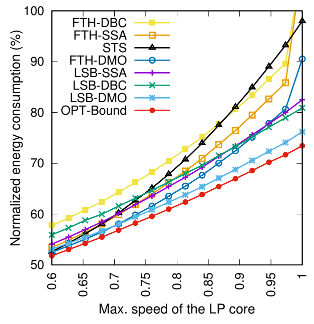


(a) Utilization = 70%

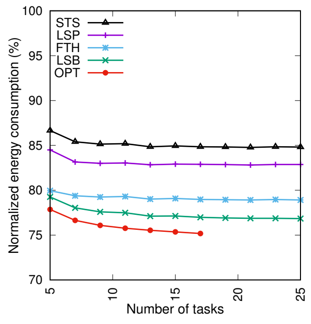


(b) Utilization = 90%

Fig. 9: Impact of threshold value in *FTH* algorithm



(a) Impact of the maximum speed of the LP core



(b) Impact of the number of tasks

Fig. 10: Additional Evaluations

LSB. Its energy consumption is minimized at some threshold value around 0.50. The energy consumption goes up as we increase the threshold and becomes constant at some point – because when the *threshold* value exceeds the utilization, all of the workload is assigned to the LP core. The threshold-independent algorithms, naturally yield a constant energy consumption. Figure 9b shows a similar pattern for a system with 90% load. The results suggest that choosing a threshold value in the range [0.5, 0.6] is generally a very good choice when using the FTH algorithm.

Impact of the maximum speed of the LP core. In this set of experiments, we varied the maximum speed of the LP core while fixing the load at 70% for each configuration (Figure 10a). The performance of LSB-DMO remains within 5% of Opt-Bound for the entire region, suggesting that it is applicable in a wide range of heterogeneous systems. FTH algorithms, on the other hand, tend to drift away from Opt-Bound as the maximum speed of the two processing cores become close to each other. We also see that the energy consumption of all schemes increases with increasing f_{max}^{LP}. This is because, when the utilization is kept fixed at 70%, when we increase f_{max}^{LP}, the effective amount of workload on the system is increased, which is reflected in the results.

Impact of the number of tasks. Figure 10b shows the impact of number of tasks for a system with utilization 70%. We see that for small number of tasks, the performance of all the schemes is affected. As the number of tasks grows, the average task size decreases and the performances of various schemes stabilize. We can see that LSB performs within 3% of the optimal scheme, and FTH is about 3% worse than LSB, for the entire region. LSP performs worse than the other two, but it also shows stable performance when the number of tasks grows. For the optimal scheme, we could only calculate energy consumptions for up to 17 tasks due to its prohibitive computational complexity.

V. Conclusion

In this paper, we proposed a fault-tolerant framework implemented on heterogeneous dual-core systems, and proposed techniques that can keep energy consumption at a minimum level. We devised task partitioning algorithms along with runtime frequency assignment policies while taking into account the different execution-time and power-parameter scaling factors for application tasks on heterogeneous cores. Our simulation experiments show that our proposed schemes perform very close to the theoretical lower bound.

Acknowledgments

This work was supported by the US National Science Foundation Awards CNS-1422709 and CNS-1421855.

References

[1] "ARM big.LITTLE Technology." http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[2] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proc. of ACM/IEEE DAC*, 2013.

[3] G. Singla, G. Kaur, A. K. Unver, and U. Y. Ogras, "Predictive dynamic thermal and power management for heterogeneous mobile platforms," in *Proc. of IEEE DATE*, 2015.

[4] I. Koren and C. M. Krishna, *Fault-tolerant systems.* Morgan Kaufmann, 2010.

[5] H. Aydin, R. Melhem, and D. Mossé, "Tolerating faults while maximizing reward," in *Proc. of IEEE ECRTS*, 2000.

[6] B. Zhao, H. Aydin, and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 2, p. 23, 2013.

[7] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[8] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 329–342, 2012.

[9] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *Proc. of IEEE ICCD*, 2011.

[10] A. Roy, H. Aydin, and D. Zhu, "Energy-aware standby-sparing on heterogeneous multicore systems," in *Proc. of IEEE/ACM DAC*, 2017.

[11] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. of IEEE CASES*, 2013.

[12] D. Zhu, R. Melhem, and D. Mossé, "The effects of energy management on reliability in real-time embedded systems," in *Proc. of IEEE ICCAD*, 2004.

[13] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Proc. of the Int. WS on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010.