A Novel Task Deployment Framework for Heterogeneous Multicore Systems Considering Circuit Aging

[1]Yu-Guang Chen, [2]Ing-Chao Lin, [2]Yu-Lin Chen, [2]Yi-Ping Chen
[1]Departmrnt of Electrical Engineering, National Central University, Taoyuan, Taiwan
[2]Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, Taiwan
andyygchen@ee.ncu.edu.tw

Abstract

Heterogeneous multicore systems are widely used nowadays to achieve a better trade-off between computing performance and power consumption. ARM big.LITTLE architecture is such an example which consists of high-performance big cores and low-power LITTLE cores to provide execution flexibility. On the other hand, the aging effect becomes a non-negligible threat in advanced CMOS technology. One of the most severe aging effects is NBTI, which can cause timing violations or even system failure. Previous studies proposed various techniques to mitigate the impact of NBTI. Most of them, however, only target on homogeneous multicore systems and are not directly applicable to heterogeneous multicore systems. Furthermore, none of these techniques consider real-time applications, where a task may be subject to a tight timing constraint even after circuit aging. Therefore, this paper investigates the characteristics of heterogeneous multicore systems and proposes an aging-aware framework to improve the system lifetime. In particular, it proposes using the asymmetric aging concept that keeps a few cores robust to address the critical tasks at later life stages and the task migration technique that executes a single task with different types of cores to provide a better trade-off between energy consumption and system lifetime. Experimental results show that the proposed framework can achieve 5.29x to 10.78x lifetime improvement and 11.8% to 23.8% average power consumption saving.

Keyword: Heterogeneous multicore system, NBTI effects, System lifetime, Asymmetric aging

1 Introduction

Homogenous multicore systems have been widely adopted in real-time computer systems nowadays to perform high-performance computing at the cost of vast power consumption. To achieve a better trade-off between performance and energy consumption, previous studies have proposed various approaches [1][2][3]. However, even though these techniques can partially reduce the huge power consumption, the identical computation units (cores) still form a barrier to future power saving. Therefore, heterogeneous multicore systems have been proposed. With the availability of various types of cores, including Central Process Units (CPU) and specific purpose accelerators, a heterogeneous multicore system can satisfy different performance requirements while keeping power consumption within specified limits. An example of this is the ARM big.LITTLE architecture comprises powerful big cores and power-efficient LITTLE cores, as shown in Figure 1[11]. The big cores achieve high performance but consume significant power, while the LITTLE cores provide power-efficient computation at the cost of performance loss. By deploying applications to appropriate cores, the system can satisfy the performance requirement while meeting power consumption constraints.

On the other hand, as transistor size shrinks, reliability becomes a non-negligible threat. One of the major reliability issues is Negative-Bias Temperature Instability (NBTI), which is caused by the instability of the Si-H bond. The dissociation of Si-H bonds along the silicon-oxide interface leads to the generation of interface traps, resulting in an increase in threshold voltage (V_{th}) in absolute value as well as the transition delay of transistors. In the long term, NBTI may cause timing violations and even system failures.

To mitigate the impact of the NBTI, many researchers have proposed different approaches to address this threat. The authors of [1][2][3] proposed different task-to-core mapping algorithms on the homogenous multicore system. The key idea is to distribute the workload evenly among all cores (symmetric aging), thereby mitigating the aging rate of each core can be reduced. However, these approaches are not applicable to heterogeneous multicore systems since they do not consider the heterogeneity of cores. The approaches proposed in [8][9][10] focus on extending the lifetime of heterogeneous multicore systems, especially the ARM big.LITTLE architecture.

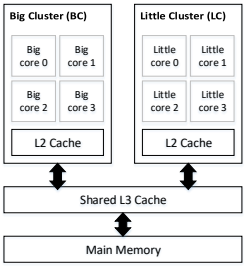


Figure 1. ARM big.LITTLE architecture [11]

Although the previous approaches cleave a path to construct reliable heterogeneous multicore systems, they predominantly rely on the concept of symmetric aging. As detailed in session 2.3, we find that the symmetric aging-based approaches may fail to obtain a better system lifetime with the presence of critical tasks in real-time applications. Here, we define a system that ends its lifetime if a given task cannot be completed within its timing constraints by any core in the system. In contrast, we propose asymmetric aging-based approaches, which keep a few robust cores and use them to execute critical tasks after circuit aging. However, applying asymmetric aging to heterogeneous multicore systems is not straightforward due to the variety of cores. Therefore, we propose a novel aging-aware task deployment framework on ARM big.LITTLE architecture [11] for real-time applications. It is worth noting that this proposed framework can also be extended to other heterogeneous multicore systems with minor adjustments.

In this paper, we carefully examine the characteristics of the big core and the LITTLE core in the architecture and employ different aging tolerance strategies for different types of cores. Specifically, we adopt an asymmetric aging-based approach in the big cluster (i.e., a set of big cores) to reserve an appropriate number of robust big cores for critical tasks that appear at a later life stage. Meanwhile, we adopt a symmetric aging-based approach in the LITTLE cluster (i.e., a set of LITTLE cores) to maximize the utilization of LITTLE cores for power-saving purposes. Moreover, in this framework, we adapt the task migration technique [5][9], which allows a single task to be executed by both big clusters and LITTLE clusters in serial order. With this flexibility, we can either obtain even better lifetime extension by further exploiting big cores or save more power by applying a portion of the task to a LITTLE core. Our framework includes task preprocessing, task scheduling, execution scenario classification/migration decision, and task-to-core assignment.

The contributions of this work are listed as follows:

- This work is the first to apply different aging tolerance strategies to heterogeneous multi-core systems. We employ an asymmetric aging-based approach laterto the big cluster, reserving an appropriate number of robust big cores at the early life stage. These cores can be used to execute critical tasks at later life stages. On the other hand, we apply the symmetric aging approach to the LITTLE cluster to maximize the utilization of LITTLE cores and save power consumption.

- We propose a task migration mechanism that can either obtain even better lifetime extension by further exploiting big cores or save more power by allocating a portion of the task to a LITTLE core.
- Experimental results demonstrate that our framework can achieve a 5.29x to 10.78x system lifetime extension and a 13.6% to 31.5% average power reduction compared to the symmetric aging-based wearout-aware approach.

The rest of the paper is organized as follows: Section 2 gives a brief introduction. Section 3 illustrates the task and heterogenous multicore system models used in this work and formally formulates the problem. Section 4 details the proposed framework. Section 5 presents the results, and Section 6 concludes the paper.

2 Preliminaries

This section first reviews heterogeneous multicore systems, with a focus on the ARM big.LITTLE architecture. After that, it discusses details about the NBTI model and various NBTI tolerance/mitigation techniques for multicore systems. Finally, it derives the differences between the symmetric aging approach and the asymmetric aging approach, using an example..

2.1 Heterogeneous Multicore System

Heterogeneous multicore systems are widely used to execute various applications in computer systems, achieving a better trade-off between performance and power consumption. Conventionally, some cores in the system can provide excellent performance at the cost of significant power consumption to address real-time applications, while others can operate at lower voltage/frequency to save power. By analyzing the characteristics of tasks and appropriately mapping them to different types of cores, the system can exploit the most appropriate resources to meet performance requirements and save power simultaneously.

One of the widely used heterogeneous multicore systems is the ARM big.LITTLE architecture [11]. It comprises several cores of different types (big or LITTLE) with a single instruction set architecture (ISA). Figure 1 shows an example of the big.LITTLE architecture, where four big cores form a big cluster (left portion of the figure) and four LITTLE cores form a LITTLE cluster (right portion of the figure). The big cores can provide strong calculation speed, while the LITTLE cores can perform low-power operations. Each core has its L1 cache; cores in the same cluster share the L2 cache, and all cores share the L3 cache. This architecture facilitates the convenient swapping of a task between cores during task execution.

The key to perfectly exploiting the performance-power trade-off of heterogeneous multicore systems relies on efficient task management policies. The authors of [5] proposed a Performance Impact Estimation (PIE) model to predict the performance variation of a workload on different core types by collecting CPI stack, MLP, and ILP profile information. It can predict which task-to-core mapping is likely to provide the best performance. The authors of [6] proposed a software-based modeling technique that can estimate the performance and power consumption of workloads for different core types. The method proposed in [7] can predict changes in the energy consumption of a program when moving from one core type to another. However, none of the above approaches take the aging effect into consideration.

2.2 NBTI Model and Prior Works

To simulate the NBTI effect properly, in this work, we apply the fitting model proposed in [12]. The model calculates the ΔV_{th} with the following equation

$$\Delta V_{th} = A \cdot V_G^{\Gamma} \cdot \exp(\frac{-E_a}{kT}) t^n \cdot PDC^{\alpha}, \quad (1)$$

where A is a technology-spcific constant, V_G is gate voltage; Γ is the power-law voltage acceleration fator; k is the Boltzmann constant; Ea is the Arrhenius T activation energy; T is the temperature in kelvin, and t is the time in second; n is the power-law time exponent; PDC is the pluse duty cycle; α is the power-law duty exponent. We set the V_G to 1.0V for the big cluster and 0.8V for the LITTLE cluster in our experiment.

To mitigate the impact of NBTI on a multicore system, serval approaches have been proposed. These approaches can be classified into design time and runtime approaches. The design-time approaches add extra timing margin as guardband to tolerance NBTI. However, these methods may lead to overdesign which causes unnecessary power consumption. The runtime approaches monitor the system and dynamically adjust the operation scenarios (for example, task-to-core mapping, operating voltage, etc.) during circuit operation. The authors [1][2][3] proposed different techniques to monitor the various operating conditions as well as task-to-core mapping algorithms to appropriately assign tasks to cores to mitigate NBTI. However, all the above methods only concentrate on homogeneous multicore system and are hard to adapt to heterogeneous multicore systems since the performance and power efficiency benefits of different type of cores is not considered. Therefore, the approaches for the heterogeneous multicore system has emerged.

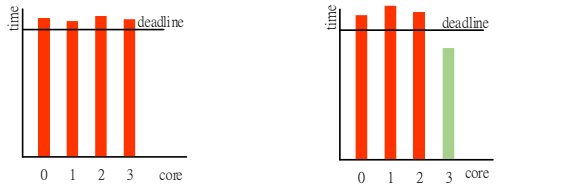
The authors of [8] proposed a lifetime reliability model based on Amdahl's Law, which analyzes core utilization, processor composition, and thread scheduling method for the heterogeneous multicore system. The study of [8] revealed that the number of big cores will influence the reliability of the system, and proposed a method to find an appropriate number of big cores in the system. The authors of [9] proposed an aging-aware task mapping algorithm that can replace the load balancing mechanisms in Linux-like runtime systems and cooperate with other components. It performs online characterization and run-time mapping of tasks to find energy-efficient mappings and meet performance requirements while reducing platform aging. The authors of [10] propose the Dynamic Reliability Management (DRM) framework that integrates resource management policies for the heterogeneous multicore system. The framework can trade-off between different metrics such as system lifetime, performance, and power consumption.

Although the above methods can successfully analyze or mitigate NBTI on heterogeneous multicore systems, they do not take real-time applications which includes critical tasks with pretty tight deadline constraints into consideration. These approaches try to make the aging rate of each core consistent, either big cores or LITTLE cores. This results in an insufficient number of robust cores in the later life stage and system failure will occur if none of the cores can complete the critical tasks. Therefore, we propose an aging-aware task deployment framework that adopts different aging tolerance strategies on different core types. Our proposed framework considers the criticality of tasks and selects the appropriate cores according to the attributes of the task to prolong the system lifetime and minimize the power consumption.

2.3 Symmetric Aging vs. Asymmetric Aging

Here we use an example to illustrate why an asymmetric aging approach can benefit system lifetime with the presence of critical tasks. Assume a multi-core system with 4 cores from 0 to 3. The multicore system is used to execute a real-time application with a group of tasks. To simplify the complexity, we assume all tasks can be executed by all cores. Figure 2 shows the execution time for a given critical task with different cores after circuit aging where the x-axis shows different cores and the y-axis gives the execution time. Figure 2(a) depicts the result from the symmetric aging approach while Figure 2(b) shows the result from the asymmetric aging approach. From the figure, we can find that since all cores age at a similar speed, none of the core can execute the given critical task within its deadline. On the other hand, with the asymmetric aging approach, core 3 is reserved at the early life stage and can finish the critical task before its deadline. Therefore, the asymmetric aging approach can successfully extend the system lifetime. However, an efficient and

effective framework to integrate the concept of asymmetric aging to a heterogeneous multicore system is still in demand.



(a) Task execution on each core with symmetric aging

(b) Task execution on each core with asymmetric aging (core 3 is reserved in the early life stage, and used to executed critical tasks in the later life stage)

Figure 2. Symmetric aging vs. asymmetric aging

3 Problem Formulation

This section first describes the heterogenous multicore system model and the real-time task model use in this paper. After that, we formally formulate the reliability aware task deployment problem for a heterogeneous multicore system.

3.1 Heterogeneous Multicore System Model and Real-time Task Model

We first detail the heterogeneous multi-core system model used in this paper. The model is adopted from the ARM big.LITTLE architecture [11], which contains two types of clusters, a big cluster, and a LITTLE cluster. The big cluster (BC) contains a set of big cores and the LITTLE cluster (LC) contains a set of LITTLE cores. The instruction set architecture of each core types are identical, thus, the tasks can be executed on any core. Besides, we assume that the tasks can be switched between different clusters to meet a given constraint.

We assume that the system continues executing a set of periodic tasks, which are expressed in the form of a Directed Acyclic Graph (DAG), $G = (T, E)$. Vertices set $T = \{t_0, t_1 ..., t_{n-1}\}$ represents n tasks and edges set $E = \{ e_{ij} \}$ $(i < j < n)$ represents the dependency of tasks. The existence of edge e_{ij} means that task t_j cannot start its execution until task t_i is completed. Based on the characteristics of tasks, we classify tasks into two categories, the big tasks, and the LITTLE tasks. A task is classified as a LITTLE task if it can successfully be completed within the deadline by a core in the LITTLE cluster. Otherwise, the task is classified as a big task. To meet the timing constraint, a LITTLE task can be assigned to either a big core or a LITTLE core, but a big task can only be executed by a big core.

Following we introduce some terms as well as equations that will be used in our algorithm, which includes the execution time, the ready time, the slack, the waiting time, the remaining time, and the criticality of a task.

The execution time of task t_i on core k is estimated using Equation (2)

$$ET_k(t_i) = N(t_i) \cdot CT_k \quad (2)$$

where $N(t_i)$ is the number of clock cycles needed of task t_i, CT_k is the clock cycle time of core k. Each task t_i has a hard deadline $d(t_i)$ and we use $RdyT_k(t_i)$ to represent the ready time for task t_i to be executed by core k as

$$RdyT_k(t_i) = max(RdyT_k, Rdy(t_i)) \quad (3)$$

where $RdyT_k$ represents the ready time of core k (i.e., core k finishes its job and is available for executing task t_i) and $Rdy(t_i)$ shows the ready time of task t_i. Therefore, the slack of task t_i on core k can be estimated using Equation (4).

$$Slack_k(t_i) = d(t_i) - (RdyT_k(t_i) + ET_k(t_i)) \quad (4)$$

where $RdyT_k(t_i) + ET_k(t_i)$ indicates the completion time of task t_i. Note that to avoid timing violation and system failure, the slack should be larger or equal to 0.

We use criticality to reflect the flexibility of executing a task. The criticality of task t_j is defined as the legal execution duration divides by its minimum execution time, as shown in Equation (5)

$$C(t_j) = \frac{d(t_i) - max\_d(t_i)}{min\_ET_k(t_j)} \ \forall i \text{ if } e_{i,j} \text{ exists}, \forall k \quad (5)$$

where $d(t_j)$ is the deadline of task t_j, $max\_d(t_i)$ is the maximum deadline of all of the tasks t_i with $e_{i,j}$ exists (i.e., the predecessors of vertex t_j in the DAG), and $min\_ET_k(t_j)$ is the minimum execution time of task t_j (with the most robust core). The smaller the $C(t_j)$ is, the higher criticality of this task has. Higher criticality implies the task must be executed with relatively more robust cores.

The waiting time of task t_i on core k can be calculated as

$$WT_k(t_i) = \begin{cases} 0, \text{if } RdyT_k(t_i) < RdyT_k \\ RdyT_k(t_i) - RdyT_k, otherwise \end{cases} \quad (6)$$

Note that $WT_k(t_i)$ is greater than 0 only when task t_i is ready to be executed before the core k is available.

3.2 Aging-Aware Task Deployment Problem

The goal of the aging-aware task deployment problem is to exploit the advantages of the different core types in the heterogeneous multicore system and adopt appropriate aging tolerance strategies to different types of cores. The task-to-core assignment algorithm will be developed based on these strategies such that the lifetime of the system can be extended and energy consumption can be minimized. Again, the system lifetime is defined as the first time when a timing failure occurs.

Figure 3 shows the problem formulation in this work. Given the big cluster BC, the LITTLE cluster LC, the task graph TG with periodic tasks, and the deadline set D with deadlines of each task, we want to find an appropriate task execution sequence as well as task-to-core mapping which can lead to the longest system lifetime with minimal power consumption.

Inputs: (1) big cluster BC with big cores (2) LITTLE cluster LC with LITTLE cores (3) task graph TG with periodic tasks (4) deadline set D
Output: The appropriate task-to-core assignment
Objective: Extend the system lifetime and minimize the power consumption

Figure 3. problem formulation

4 Aging-Aware Task Deployment Framework

This section first presents the concept and the overall flow of the framework. Then, this section details each stage in the framework.

4.1 Framework Overview

To realize the above objective, we propose different aging tolerance strategies for the big cluster and the LITTLE cluster. For the big cluster, we propose using an asymmetric aging approach to reserve an appropriate number of big cores in the early life stage. These reserved big cores can be used to execute critical tasks after all the other cores are aged so the deadline of the task can be met. The lifetime of the system then can be extended. On the other hand, we propose using symmetric aging on the LITTLE cluster so the LITTLE cores can be sufficiently used and the power consumption can be minimized. Note that even all LITTLE cores are aged, a critical LITTLE task can still be executed by a big core with the cost of extra power consumption. Therefore, the timing failure will not occur.

Furthermore, we propose task migration techniques to execute a task with both a big core and a LITTLE core. When a big task with large slack pops up, we can firstly execute the task with a LITTLE core and then migrate it to a big core. With sufficient slack, the task still can be completed before its deadline with less power consumption. On the other hand, when no LITTLE core can complete a LITTLE task within the deadline, the LITTLE task can be migrated to a big core. With more powerful computation ability, the LITTLE task can be completed within the deadline. To realize the above idea, we propose an NBTI-aware task scheduling framework as shown in Figure 4.

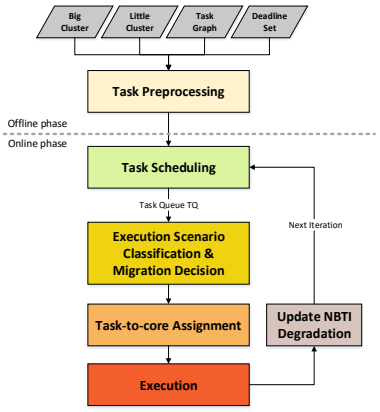


Figure 4. The flowchart of the proposed framework

Two phases exists in our framework: the offline phase and the online phase. The offline phase estimates the execution time of each task and classifies it into big tasks or LITTLE tasks. In the online phase, these tasks are continuously executed in the system through four stages: task scheduling, execution scenario classification and migration decision, task-to-core assignment, and task execution. The NBTI degradation of each core is updated for the next iteration. The following sub-sections details each stage.

4.2 Task Preprocessing

The first step of our framework is to classify tasks to either big tasks or small tasks. We use equation (2) to estimate the execution time of a task on all cores. If a task cannot be completed before its deadline with any of the LITTLE core, we classify it as a big task. This implies that the computational requirement of the task is too large to execute on a LITTLE core; therefore, a high-performance big core is required. On the contrary, if a task can be executed on a LITTLE core and meet its deadline, the task is marked as a LITTLE task. Therefore, the task can be executed on a power-efficient LITTLE core to save power consumption.

4.3 Task Scheduling

After preprocessing the tasks, we need to schedule the tasks and find an appropriate execution order. By the sequence of the tasks and the dependency in the task graph, we map each task which is ready to execute to a core one at a time. However, there may be a scenario that the number of available cores is fewer than the ready tasks. The situation that no core can execute these ready tasks can result in deadline violation. If the executing order is inappropriate, the lifetime of the system may be shortened substantially. To deal with this problem, we implement a task queue and push all ready tasks into it. Then these tasks are sorted by their deadline in increasing order, and the task with the smallest deadline in the task queue will have the highest order for task-to-core assignment. With this policy, we provide more flexibility to the task with a pretty tight deadline to avoid timing failure to occur.

4.4 Execution Scenario Classification and Migration Decision

The task scheduling stage will pick a ready task, and the next step is to find the appropriate core to execute it. Based on the classification result in task preprocessing and current system loading, our method will classify the execution scenario into four different situations. In the meantime, our method will identify whether migration will benefit from executing the task. The proposed algorithm is shown in Algorithm 1.

In our algorithm, we first identify the task type (line 1). If the selected task, t_i, is a big task but none core in the big cluster is available at this moment, we will check the availability of the LITTLE cluster (line 2). If at least one core is available at the LITTLE cluster, we will evaluate the possibility to execute the task with LC first and then migrate to BC (line 3, will be detailed in section 4.6.1/algorithm 6). Otherwise, task t_i will be assigned to the big cluster (line 5, will be detailed in 4.5.1/algorithm 4). On the other hand, if the task is a LITTLE task, then the slack of the task t_i is estimated in the LITTLE cluster (line 8). If task t_i is unable to finish within deadline on any LITTLE core (line 9), then task t_i will be migrated to the big cluster to speed up its execution (line 10, detail in 4.6.2/algorithm 7). Otherwise, task t_i is assigned to LC (line 12, detail in 4.5.2/algorithm 5).

Algorithm 1. Execution Scenario Classification and Migration Decision

Input: (1) task t_i and its type (*isBig*) in task queue (2) big cluster BC with big cores (3) LITTLE cluster LC with LITTLE cores
Output: The task to core mapping or migration for task t_i

```
1: if isBig(t_i) then
2:   if all big cores are executing tasks && at least a LITTLE core is free then
3:     Evaluate the possibility to execute task t_i at LC and then
4:     migrate it from LC to BC; /* Algorithm 6*/
5:   else
6:     Assign task t_i to BC; /* Algorithm 4 */
7:   end if
8: else
9:   Estimate the slack of task t_i in LC;
10:  if task t_i is unable to finish within the deadline in LC then
11:    Migrate task t_i to BC; /* Algorithm 7 */
12:  else
13:    Assign task t_i to LC; /* Algorithm 5 */
14:  end if
15: end if
```

4.5 Task-to-core Assignment

This subsection introduces the task-to-core assignment algorithm which includes task-to-big-core and task-to-LITTLE-core assignment.

4.5.1 Task-to-big-core Assignment

In this stage, we need to find an appropriate big core to execute the big task. We propose using the asymmetric aging approach in this stage. The key concept of asymmetric aging is to keep a few numbers of robust cores idle unless necessary in the early life stage of the system. After the system runs for a long time, all non-reserved cores will suffer from NBTI degradation and are unable to complete the critical tasks in the later life stage. These reserved cores now are used to execute critical tasks. Since they are all still robust, the deadline for the critical tasks can be met and the lifetime of the system is prolonged. However, doing so may decrease the number of available big cores at the early life stage and the big tasks may not able to be completed on time. Therefore, an efficient method is needed to decide when to use the reserved cores.

If the task is non-critical, we can select a relatively weak core to execute it. Otherwise, if the task is critical, we need to select a relatively robust core to execute the task. We use equation (7) and equation (8) to represent the ratio of robustness and weak.

$$Cost_{robust} = \frac{V_{th}(k) - minV_{th}}{minV_{th}} \quad (7)$$

$$Cost_{weak} = \frac{maxV_{th} - V_{th}(k)}{maxV_{th}} \quad (8)$$

The smaller $Cost_{robust}$ means that the core is relatively robust in the system since its V_{th} value is smaller. The smaller $Cost_{weak}$ means the core is relatively weak in the system since its V_{th} value is larger. We then adopt the weighted comprehensive criterion method (WCCM) [20] to merge the two equations into a single cost function to evaluate the cost of selecting core b as

$$cost(b) = P \times Cost_{robust} + (1 - P) \times Cost_{weak}. \quad (9)$$

Coefficient P is the weight of $Cost_{robust}$, where P is from zero to one. The complement of P, (1 - P), is the weight of $Cost_{weak}$. We use this coefficient to control whether the selected core is relatively robust or relatively weak. Suppose we want to choose a relatively robust core, we need to set a larger P. Conversely, we can set a smaller P to select a relatively weak core. The

equation to obtain P is shown in equation (10). The V_{th} value of each core with non-negative slack will be recorded in a set, *vth_set*, and the size of *vth_set* represents the number of cores that can successfully execute this task.

$$P = \frac{\#\ big\ core - size\ of\ Vth\_set}{\#\ big\ core}. \quad (10)$$

Consider two extreme situations: If the size of the *vth_set* is equal to the number of big cores, then P is zero. This means many cores can successfully execute this task, and this task is less critical. Thus, $Cost_{weak}$ will dominate the cost function. Otherwise, if the size of the *vth_set* is zero, then P is 1. This implies the task is critical. Thus, $Cost_{robust}$ will dominate the cost function. Algorithm 2 summarizes the task to big core assignment algorithm.

Algorithm 2. Task-to-big-core Assignment

Input: (1) big task t_i and its deadline time $d(t_i)$, ready time $RdyT_b(t_i)$
(2) big cluster BC with big cores
Output: *cand_b*: The candidate big core with minimum cost

1: initialize P;
2: initialize threshold voltage set *vth_set*;
3: **for each** core b in BC **do**
4: $ET_b(t_i)$ = Execution time of task t_i on core b;
5: $slack_b(t_i) = d(t_i) - RdyT_b(t_i) - ET_b(t_i)$;
6: **if** $slack_b(t_i) > 0$ **then**
7: record the V_{th} of core b in *vth_set*;
8: **end if**
9: **end for**
10: $P = \frac{\#\ big\ core - size\ of\ vth\_set}{\#\ big\ core}$;
11: **for each** core b where its V_{th} is in *vth_set* **do**
12: $cost(b) = P \times Cost_{robust} + (1 - P) \times Cost_{weak}$;
13: **end for**
14: **return** *cand_b* = the core b with the minimum cost;

4.5.2 Task-to-LITTLE-core Assignment

This algorithm selects a LITTLE core in the LITTLE cluster to execute LITTLE task t_i. We use the symmetric aging approach to deal with this problem. There are two reasons for using symmetric aging instead of asymmetric aging. First, we have reserved big cores in the big cluster, and these reserved big cores can execute both the big tasks and LITTLE tasks; therefore, it is not necessary to reserve LITTLE cores. Second, since some big tasks may migrate to the LITTLE cluster, we should always keep the larger availability of LITTLE cores. To achieve symmetric aging in the LITTLE cluster, we use the cost function in equation (11) to determine the robustness of a LITTLE core:

$$cost(l) = \frac{Vth(l) - minV_{th}}{minV_{th}} \quad (11)$$

The smaller $cost(l)$ of core l is, the more robust it is. Then, we execute the task by selecting the most robust core at all times.

4.6 Task Migration

This section details the task migration mechanism including migrating big task to the LITTLE cluster and migrating the LITTLE task to the big cluster.

4.6.1 Migrate a big task to LITTLE cluster

In this sub-section, we detail how to evaluate the possibility to execute a big task with LC first and then migrate to BC with the lack of available big cores. The propose of doing so is to shorten the completion time of the big task so the probability of timing violation to occur can be minimized and the system lifetime can be extended.

Figure 5 shows the flow of the estimation procedure. The waiting time of big task t_i is first estimated on each big core by equation (6). A candidate big core (denoted as *cand_b*) with minimum waiting time is selected. Then the execution time of task t_i on *cand_b* is estimated by equation (2). On the other hand, we select a candidate LITTLE core (denoted as *cand_l*) with minimum execution time as the target core of task t_i to migrate. After that, we need to set the maximum execution duration of task t_i on *cand_l* since task t_i is a big task and cannot completely be executed on any LITTLE core. We set the maximum execution duration of task t_i on *cand_l* as the waiting time for *cand_b*. Once *cand_b* is available, task t_i will be continuously executed on *cand_b* to meet its deadline. However, since the clock period differs from the big core and LITTLE core, the execution time on *cand_b* can be calculated by equation (12):

$$RT_{cand\_b}(t_i) = ET_{cand\_b}(t_i) - (\ pET_{cand\_l}(t_i) \cdot BL_{ratio}\ ) \quad (12)$$

where

$$BL_{ratio} = \frac{\text{clock period } (cand\_b)}{\text{clock period } (cand\_l)} \quad (13)$$

and $pET_{cand\_l}(t_i)$ denotes the execution time of task t_i on the LITTLE core. The BL_{ratio} gives the conversion ration based on the clock periods. Besides, the migration cost is also considered and is calculated by 2% of execution on *cand_l* which is referred to [15].

With the above estimations, we can calculate the completion time of the task with and without migration. The completion time of migrating task t_i ($T_{migrate}$) is calculated as the sum of the portion execution time on *cand_l*, the remaining execution time on *cand_b,* and the migration cost. The cost without migrating task t_i (T_{stay}) is obtained as the sum of waiting time and execution time on *cand_b*. Finally, we compare the completion time of the two situations and select the one with earlier completion time. If T_{stay} is lower, task t_i will be stalled until *cand_b* is available and then execute on it. Otherwise, if $T_{migrate}$ is lower, task t_i will be executed on *cand_l* while waiting for *cand_b*. After *cand_b* is available, the remaining portion of the task t_i will be completed on *cand_b*. Algorithm 3 summarizes the estimation procedure.

Algorithm 3. Migrate a big task to LITTLE cluster

Input: (1) big task t_i (2) big cluster BC with big cores (3) LITTLE cluster LC with LITTLE cores
Output: Task-to-core mapping

/* STEP1: Estimate the timing information of task t_i on each core and find the candidate core for each cluster */
1: **for each** core b in BC **do**
2: $WT_b(t_i)$ = the waiting time of task t_i on core b;
3: **end for**
4: *cand_b* = the candidate big core with minimum waiting time;
5: $WT_{cand\_b}(t_i)$ = the waiting time of *cand_b*;
6: $ET_{cand\_b}(t_i)$ = the execution time of task t_i on *cand_b* ;
7: **for each** core l which is free in LC **do**
8: $ET_l(t_i)$ = the execution time of task t_i on core l ;
9: **end for**
10: *cand_l* = the candidate LITTLE core with minimum execution time;
11: $pET_{cand\_l}(t_i)$ = the execution time of partially completed task t_i on *cand_l*;
12: $RT_{cand\_b}(t_i)$ = the execution time of remaining portion of task t_i (will be executed on *cand_b*);
13: $MG_{cost} = ET_{cand\_l} * 0.02$;
/* STEP2: Migration decision based on the cost */
14: $T_{stay} = WT_{cand\_b}(t_i) + ET_{cand\_b}(t_i)$;
15: $T_{migrate} = pET_{cand\_l}(t_i) + RT_{cand\_b}(t_i) + MG_{cost}$;
16: **if** $T_{stay} > T_{migrate}$ **then** /* With migration */
17: Execute task t_i on *cand_l* within $pET_{cand\_l}(t_i)$;
18: Execute remaining portion of task t_i on *cand_b* within $RT_{cand\_b}(t_i)$;
19: **else** /* Without migration */
20: Stall task t_i until *cand_b* is ready;
21: Assign task t_i to *cand_b*; /* *Algorithm 4* */
22: **end if**

4.6.2 Migrate a LITTLE task to big cluster

In this stage, we have estimated the LITTLE task t_i is not able to complete in the LITTLE cluster within its deadline. We then migrate this LITTLE task to a big core to execute it. The first step is to estimate the slack of each big core in the big cluster. We choose the big core with maximum slack to execute task t_i, which is the most robust big core in the big cluster. If we execute this task on a weak big core, the execution time will be longer and may affect subsequent tasks. Moreover, since the task is a lightweight LITTLE task, it will not cause too much burden on the big core. The procedure is similar to the big task migration estimating procedure. It is not repeated due to space limitation.

5 Evaluations

5.1 Experimental Setup

Our benchmark circuits are from ISCA'85 and ITC'99. The original benchmark circuits are used as LITTLE cored and are expanded as big cores. We then implement all the algorithms mentioned above in C++, and construct an in-house simulator for big.LITTLE architecture with 4 big cores and 4 LITTLE cores. We set the operating voltage to 1.0V for all big cores and 0.8V for all LITTLE cores. To simulate the influence of the NBTI effect, we customize PTM model cards [13] with different aging scenarios and use HSPICE simulation to obtain corresponding delay and power information.

We use TGFF [14] to generate task graphs, and each task graph contains 20 tasks. These 20 tasks include big or LITTLE tasks, and each big/LITTLE task is either a critical or non-critical task. We use three different ratios of critical tasks for experiments: 20%, 40%, and 60% (i.e. 4, 8, 12 critical tasks in a task graph). Our in-house simulator operates with HSPICE to take these task graphs as input, and the system lifetime and corresponding power consumption can be obtained.

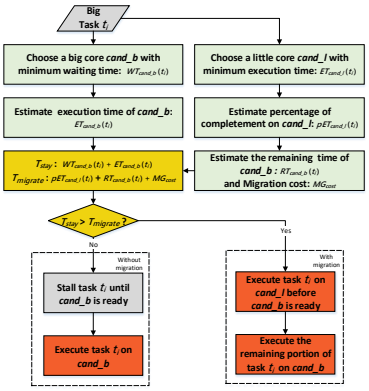


Figure 5. big task migration estimating procedure

There are five policies in our experiment: Sym, Asym, Spare, Sym_mig, and Asym_mig. Sym adopts the symmetric aging based method in the big cluster. Asym adopts the asymmetric aging based method in the big cluster. Spare uses an extra big core to handle critical tasks in a big cluster. Sym_mig and Asym_mig further adopt the task migration technique on Sym and Asym, respectively. Note that all of these policies adopt symmetric aging in the LITTLE cluster. The generated task graphs are executed by each policy with each benchmark pair (e.g. c432_c432d) and the average lifetime and power consumption are obtained when the system ends its lifetime.

5.2 Lifetime and Energy Evaluation

Figure 6(a), Figure 6(b), and (c) present the lifetime of each policy under 20%, 40%, and 60% critical tasks respectively. The lifetime of each policy is normalized to the Sym policy (baseline). The Spare policy can achieve 3.29x lifetime improvement compared to baseline, but it will require an additional area overhead of 25% due to the extra big core that handles critical tasks. Asym can obtain at most 2.09x lifetime improvement. Among the policies with the task migration technique, Sym_mig achieves 3.4x to 7.32x lifetime improvement compared to the baseline, and Asym_mig achieves 5.29x to 10.78x lifetime improvement compared to the baseline. Both approaches achieve a higher lifetime improvement than the policies without task migration. The reason for such an improvement is part of the big tasks are migrated to the LITTLE cluster, which reduces the waiting time and congestion of big tasks in the big cluster. Besides, it gains some recovery time for big cores and the timing degradation caused by NBTI is slow down. Compare Sym_mig and Asym_mig, Asym_mig adopts the asymmetric aging so that the reserved big cores can be used to handle critical tasks in the later life stage of the system, thus, extend the lifetime of the system. Next, we discuss the average power consumption of each policy. Figure 7(a), (b), and (c) show the average power consumption under 20%, 40%, and 60% critical tasks of the five policies. Since the Spare policy uses an extra big core to execute critical tasks, its average power consumption is 1.09x to 1.22x compared to the baseline. With the task migration technique, the power consumption of Sym_mig is 0.9x to 0.87x compared to the baseline, which is reduced by 11.1% to 14.9%. Asym_mig, which adopts the asymmetric aging, has further improved the power consumption by 0.88x to 0.76x compared to the baseline and 0.99x to 0.97x compared to Sym_mig.

6 Conclusion

In this paper, we propose an aging-aware task deployment framework for a heterogeneous multicore system. We suggest using asymmetric aging in the big cluster and symmetric aging in the LITTLE cluster, and migrating tasks between different types of cores to achieve a better system lifetime and lower power consumption. The experimental results indicate that our method can improve lifetime by 5.29x to 10.78x compared with the baseline symmetric aging method, and it can also lead to an 11.8% to 23.8% improvement in average power consumption.

References

[1] M. Basoglu, et al., "NBTI-Aware DVFS: A New Approach to Saving Energy and Increasing Processor Lifetime," in Proc. ISLPED 2010 , pp. 253-258

[2] A. Tiwari and J. Torrellas, "Facelift: Hiding and slowing down aging in multicores," in Proc. MICRO 2008 , pp. 129-140

[3] F. Patema, A. Acquaviva, and L. Benini, "Aging-aware energy-efficient workload allocation for mobile multimedia platforms," IEEE Transactions on Parallel and Distributed Systems, vol. 24, issue 8, pp. 1489-1499, 2013.

[4] Y. Chen, I. Lin and J. Ke, "ROAD: Improving Reliability of Multi-core System via Asymmetric Aging," in Proc. ICCAD 2019, pp. 1-8

[5] K. Van Craeynest, et al., "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in Proc. ISCA 2012, pp. 213-224.

[6] M. Pricopi, et al., "Power-performance modeling on asymmetric multi-cores," *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Montreal, QC, 2013, pp. 1-10.

[7] E. Vasilakis, et al., "Modeling energy-performance tradeoffs in ARM big.LITTLE architectures," in Proc. PATMOS, 2017, pp. 1-8.

[8] W. J. Song, et al., "Amdahl's law for lifetime reliability scaling in heterogeneous multicore processors," in Proc. *HPCA*, 2016, pp. 594-605.

[9] T. R. Mück, et al., "Exploiting Heterogeneity for Aging-Aware Load Balancing in Mobile Platforms," in IEEE Transactions on Multi-Scale Computing Systems, vol. 3, no. 1, pp. 25-35, 1 Jan.-March 2017

[10] A. Baldassari, et al., "A dynamic reliability management framework for heterogeneous multicore systems," in Proc. DFT 2017, pp. 1-6.

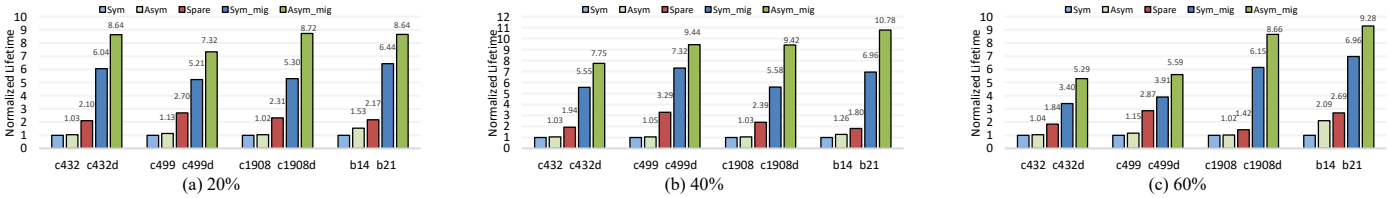
[11] Arm, big.LITTLE Processing technologies, https://www.arm.com/why-arm/technologies/big-little

[12] A. Thirunavukkarasu et al., "Device to Circuit Framework for Activity-Dependent NBTI Aging in Digital Circuits," in IEEE Transactions on Electron Devices, vol. 66, no. 1, pp. 316-323, Jan. 2019, doi: 10.1109/TED.2018.2882229.

[13] Predictive Technology Model (PTM). [Online]. Available http://www.ptm.asu.edu.

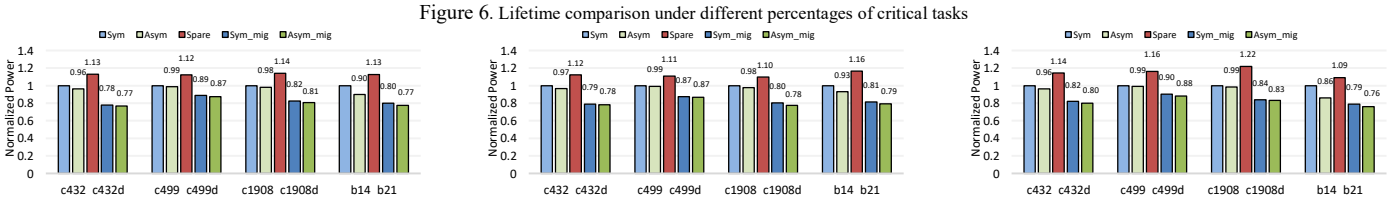
[14] R. P. Dick, D.L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in Proc. International Workshop on Hardware/Software Codesign, pp. 97-101, 1998.

[15] H.S. Chwa, et al."Energy and feasibility optimal global scheduling framework on big. LITTLE platforms," in Proc. of RTSOPS, pp. 1-11.



(a) 20% (b) 40% (c) 60%

Figure 6. Lifetime comparison under different percentages of critical tasks



(a) 20% (b) 40% (c) 60%

Figure 7. Power comparison under the different percentage of critical tasks