Workload Capacity Considering NBTI Degradation in Multi-core Systems

Jin Sun*, Roman Lysecky*, Karthik Shankar*, Avinash Kodi†, Ahmed Louri*, and Janet M. Wang*
*Department of Electrical and Computer Engineering, The University of Arizona, USA
†Department of Electrical Engineering and Computer Science, Ohio University, USA
*Email: {sunj, rlysecky, karthik1, louri, wml}@ece.arizona.edu, †kodi@ohio.edu

***Abstract*—As device feature sizes continue to shrink, long-term reliability such as Negative Bias Temperature Instability (NBTI) leads to low yields and short mean-time-to-failure (MTTF) in multi-core systems. This paper proposes a new workload balancing scheme based on device level fractional NBTI model to balance the workload among active cores while relaxing stressed ones. The proposed method employs the Capacity Rate (CR) provided by the NBTI model, applies Dynamic Zoning (DZ) algorithm to group cores into zones to process task flows, and then uses Dynamic Task Scheduling (DTS) to allocate tasks in each zone with balanced workload and minimum communication cost. Experimental results on 64-core system show that by allowing a small part of the cores to relax over a short time period (10 seconds), the proposed methodology improves multi-core system yield (percentage of core failures) by 20%, while extending MTTF by 30% with insignificant degradation in performance (less than 3%).**

I. Introduction

As device feature sizes continue to shrink, long-term reliability or permanent fault such as Negative Bias Temperature Instability (NBTI) affects system life-span, and leads to low yields and short mean-time-to-failure (MTTF) in multi-core systems. A number of new techniques have recently emerged to cope with permanent faults such as NBTI using post-manufacturing burn-in [1][2][3]. However, very little attention has been paid to device stress and its impact on system life-span and performance in the multi-core era. Device stress may happen after days of full workload operation, and requires days to relax before recovering. Letting the device completely wear-out will impact the system as defective cores have to be permanently removed from the pool of active cores. A meaningful approach would be relaxing cores when they are stressed long before they are completely wear-out. This approach would require solving three challenges: how to assess a core is stressed, how to assign workload to relax stressed cores, how to avoid additional performance cost associated with balancing workload. The proposed approach in this paper answered all these questions. We explain each part in turn.

Different from existing approaches [2][3] that focused on long-term stress using static NBTI models, we propose to use a fractional stress and recovery model to model partially stressed cores: cores alternate between a heavy workload phase (once the core has relaxed) and a light workload phase (when the core becomes stressed) with high frequency of alternation between phases. During the light workload phase, pmos transistors of idle gates are set to "1" to relieve the stress as discussed in [4]. The end result of fractional NBTI model is a core capacity rate (CR) that indicates how much additional workload one core can accept before getting over-stressed.

A number of recent works have focused on how to balance workload considering communication cost. Though not solving the same problem, some of these ideas are worth mentioning and provided a good initiative of the current work. For example, in [5] the authors proposed a dynamic workload balancing strategy called Dynamic Load Balancing using Expectation-Maximization (DLBEM). To characterize workload a mixture Gaussian model is employed to estimate workload distribution. Iterative load balancing approaches such as diffusive load balancing [6][7] exchange workload information between neighbor processors, thus avoid communication messages flooding in the entire system. [8] suggests a predictive method which estimates workload information based on historical workload data.

These above reviewed ideas and approaches work well in their applications but may not be applicable in our scenario. For example, DLBEM requires an explicit expression of probability distribution function (PDF) of the workload distribution, which is a significant limitation to our situation where explicit PDF only exists at device level. Besides, the job migration policy in DLBEM tends to cause considerable communication overhead when doing workload balancing. Iterative load balancing approaches ensure local workload balancing, and have been proved to be globally imbalance [9]. [8]'s approach was built on historical workload and thus could not provide us dynamic workload balancing.

The proposed approach has three components: NBTI introduced core performance difference estimation, Dynamic Zoning (DZ), and Dynamic Task Scheduling (DTS). Core performance difference is estimated using fractional NBTI model and indicated by capacity rate. Each core has its own capacity rate. Dynamic Zoning (DZ) algorithm groups cores into zones according to core capacity rates. It starts with a rectangular region as the initial zone, and adjusts gradually considering zone connectivity and core capacity rate to match workload from the assigned flow. Then, Dynamic Task Scheduling (DTS) algorithm allocates tasks in one zone to achieve maximum utilization with little communication cost. Fig. 1 demonstrates the flow of our proposed approach. Note that this method is iterative, the core capacity rate should be updated frequently because of changing core performance or aging effect.

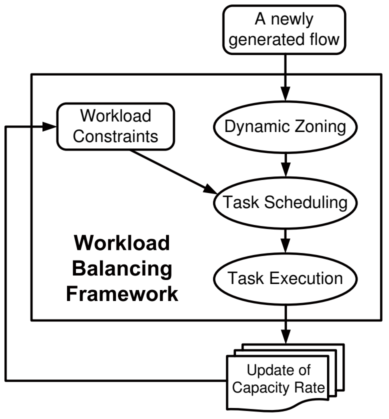

Fig. 1. The general flow of the proposed workload balancing methodology

Specifically, the new contributions of this paper are: 1) a system NBTI stress estimation model, 2) a new workload balancing strategy considering core's performance difference, 3) a novel scheduling scheme including data packet size and communication cost, and 4) a new insight into the relationship between core recovery time, stress time, and workload, and their impact on core life-span. This strategy has shown good system yield improvement and extended MTTF with insignificant impact on latency or communication traffic overhead. Experimental results on a 64 core system show that by allowing a part of cores (approximately 12.5%) to relax over a short time period (10 seconds), the proposed methodology improves multi-core system yield by reducing core failure rate by 20%, and extending MTTF by 30% with insignificant degradation in performance (less than 3%).

The remainder of this paper is organized as follows. Section II introduces the NBTI device and core model. Section III discusses the proposed workload balancing methodology considering NBTI introduced performance difference. Experimental results are included in Section IV. Finally, Section V concludes the paper.

II. NBTI Model for Multi-core Performance

NBTI limits lifetime in nano-scale integrated circuits and continues to worsen with device scaling beyond 90nm [1][2][3]. When PMOS is negatively biased, the electrical field across the gate oxide produces a complicated electro-chemical reaction that consequently increases the PMOS threshold voltage over time. The impact of NBTI may take days or months to ultimately affect timing and circuit delay, eventually leading to system failure.

The NBTI impact causes the core to alternate between stress and recovery phases. In general, recovery and stress period are fairly symmetric. Suggested by the sampling time of NBTI sensors [10], we currently assume a 10-second duration for each period. The NBTI introduced threshold voltage change ΔV_{th} in stress phase follows a similar style as reported in [3]:

$$\Delta V_{th} = \left(K_v \left(T(t) \right) (t - t_0)^{\frac{1}{2}} + \Delta V_{th0}{}^{\frac{1}{2n}} \right)^{2n} \tag{1}$$

where $T(t)$ represents temperature fluctuation with regard to time. In recovery phase, the V_{th} degradation can be represented as:

$$\Delta V_{th} = V_{th0} \left(1 - \frac{2\epsilon_1 + \sqrt{\epsilon_2 C(T(t))(t - t_0)}}{2t_{ox} + \sqrt{C(T(t))t}} \right) \tag{2}$$

Considering the temperature changes with regard to time, we include the time dependency in parameters K_v and C, as they both are functions of $T(t)$. The complete expressions of the associated parameters in (1) and (2) can be referred to [3].

The change in threshold voltage in turn affects timing and leakage power of the core. Take timing issue for example. According to [11][12], delay model considering V_{th} change can be estimated by using first-order Taylor expansion. We extend this model to critical path delay model. The i-th critical path delay can be represented as:

$$d_i = d_{i0} + \left(\partial d_i / \partial V_{th} \right) \Delta V_{th}. \tag{3}$$

Here the delay is modeled as Gaussian distribution perturbed around its nominal value d_{i0}. For a single core, we may have a number of critical paths. The worst-case is the critical path with largest variation:

$$\Delta d_{max} = \min \left\{ \max \left(\left(\partial d_i / \partial V_{th} \right) \Delta V_{th} \right), 3\sigma \right\}. \tag{4}$$

The greater the delay variations, the less workload the core should accept. We define *Capacity Rate* (CR) as an indication of how much workload one core can accept. We relate delay variations with core's capacity rate using the following percentage model:

$$CR^{(d)} = 1 - \Delta d_{max} / 3\sigma \tag{5}$$

where σ indicates the delay variance subject to a Gaussian distribution. Apparently our concern is only positive delay fluctuation. Therefore when worst-case delay variation Δd_{max} is zero, this core's capacity rate is at 100% or 1. The opposite situation is that Δd_{max} is equal to its extreme value 3σ. In this case, capacity rate is at its lowest, 0. The majority capacity rate will lie in between these two extreme cases, indicating core's performance difference. A core with capacity rate 0.6 can accept only 60% workload compared with a core with full capacity rate.

On the other hand, V_{th} variation also has a dramatic impact on leakage power. The impact of leakage variation on core's capacity rate could be derived in a similar way (detailed in [13]). We denote it by $CR^{(L)}$. If we consider both delay and leakage power impacts at the same time, a simple average model provides:

$$\text{Capacity Rate} = 0.5 * CR^{(d)} + 0.5 * CR^{(L)}. \tag{6}$$

Thus, we have transferred the NBTI device model to core capacity rate interpretation. The NBTI model first explicates its impact on threshold voltage. Then we model the variation in threshold voltage on system-level delay and leakage power respectively. The end result is the capacity rate for each core in percentage. The generated capacity rate will be treated as an upper bound limit of assigned workload on each core. Since the impact of NBTI on threshold voltage changes over time, as will the corresponding capacity rate.

NBTI introduced threshold voltage fluctuation also affects system life-span, leading to short MTTF in multi-core systems. The MTTF estimation follows the derivation from [14][15] where MTTF is modeled as an exponential function of operating temperature T, and temperature T is dependent on threshold voltage change ΔV_{th}.

III. NBTI Aware Workload Balancing

This section introduces a new workload balancing framework. This proposed method groups cores into zones based on capacity rates. Each zone has one task flow. Task scheduling within the zone is formulated as a mixed-integer program (MIP) considering workload balancing and communication cost.

A. Dynamic Zoning

We propose Dynamic Zoning (DZ) to spread out the workload across the entire multi-core network. Workloads are fundamentally composed of sets of tasks, named task flows. Tasks within a particular flow may have dependencies among them, while tasks from different flows tend to have very few or no dependencies among them [16]. Therefore, to minimize communication cost, we limit the process of one task flow to a group of cores physically adjacent to each other. Such a group of cores is defined as a *zone*. The DZ algorithm aims at mapping a task flow onto one particular zone. We then schedule tasks belonging to this flow among the cores within the assigned zone.

Given a task flow presented as a DAG (Directed Acyclic Graph) $G = (V, E)$ in Fig. 2, nodes $V = \{v_1, v_2, \cdots, v_n\}$ represent a set of tasks to be executed. And the arcs $E = \{(i, j)\}$ specify precedence relationships: each arc (i, j) means that task v_i must be completed before v_j can start execution. Task weights w_i's represent the execution time of task v_i on a non-stressed core (i.e. a core with capacity rate 1). The numbers at the input and output of each node are the number of data tokens which represent the number of data packets are consumed (at input of node) or produced (at the output of each node) on each arc [17]. The data token concept is going to be revisited in Section III-B. The workload caused by this flow can be evaluated by the structure of the task graph and the task weights. These two factors provide sufficient information for workload estimation. Each zone, once generated for a particular

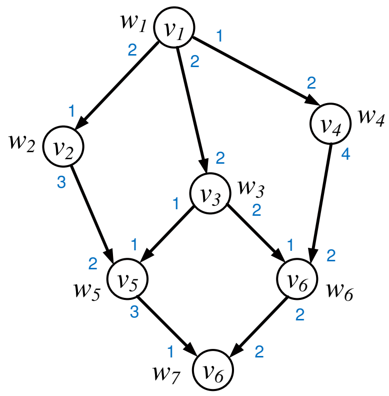

Fig. 2. A task graph representing a flow

flow, should be able to accept the workload evaluated based on the given flow. On the other side, as capacity rate specifies the upper bound of workload one core can accept, the sum of core capacity rates in one zone reflects the maximum total workload one zone can accept. This summation must exceed the estimated workload resulted from the flow assigned to this zone. We now explain how to evaluate the workload induced by a particular flow. We sum up all the task weights of its task graph, and average the total sum over the length of the worst-case path because the worst-case path determines the longest execution time. The estimated workload induced by this flow is defined as follows:

$$\text{Workload} = \frac{\sum_i w_i,\ \forall v_i \in V}{\sum_j w_j,\ \forall v_j \in WCP} \tag{7}$$

where WCP is the set of nodes existing on the worst-case path. The workload estimation defined in (7) is the minimally required capacity rate for a required zone to process this particular flow, since capacity rate indicates the upper bound of acceptable assigned workload.

Zones are not always in rectangle shapes in the current paper. Several factors determine the shape: capacity rates of included cores, total communication distances, and the size of zones (defined as the number of cores in each zone). We start zoning in a greedy way: start with a rectangular region for each zone and perturb gradually to meet the communication and capacity rate requirement. The requirement of capacity rate in this zone has been discussed above. Besides, to ensure low communication cost, a good zoning should have short Manhattan distances among cores in one zone. Thus, the problem of organizing an optimal zone for a particular flow can be described as:

$$\begin{aligned} \text{find} \quad & Z_k \\ \text{minimize} \quad & \sum_{(i,j)} d(i,j),\ \forall v_i, v_j \in Z_k \\ \text{subject to} \quad & \sum_i CR_i \geq \text{Workload},\ \forall v_i \in Z_k \end{aligned}$$

where Z_k is the zoning result consisting of an optimal set of adjacent cores, $d(i,j)$ denotes the Manhattan distance between any two cores in this zone, and CR_i represents the capacity rate for each included core. The workload information of this flow is evaluated according to (7).

Algorithm 1 describes the proposed DZ method. When a new task flow comes into the system, DZ algorithm first searches for the maximally empty contiguous region of available cores (‘Find_Max_Region’ operator). Then starting from one corner of the explored empty region, ‘Initial_Rectangle’ operator initializes a zone in rectangular shape. Rectangular shape leads to short Manhattan

Algorithm 1 Dynamic Zoning

```
Input: a task flow to be allocated;
       a multi-core system with core capacity rate identified
Output: the optimal zoning result to execute the given flow
 1: $S \Leftarrow$ Find_Max_Region ()
 2: $DZ\_opt \Leftarrow$ Initial_Rectangle ($S$)
 3: $Dist\_opt \Leftarrow$ Manhattan_Distances ($DZ\_opt$)
 4: $k \Leftarrow 0$
 5: while $k <$ N and $Dist\_opt > D_{th}$ do
 6:     $DZ\_new \Leftarrow$ Perturbation ($DZ\_opt$)
 7:     $Dist\_new \Leftarrow$ Manhattan_Distances ($DZ\_new$)
 8:     if $Dist\_new < Dist\_opt$ then
 9:         $DZ\_opt \Leftarrow DZ\_new$;  $Dist\_opt \Leftarrow Dist\_new$
10:     end if
11:     $\Delta Dist \Leftarrow Dist\_new – Dist\_opt$
12:     if exp($–\Delta Dist/N$) $>$ random (0,1) then
13:         $DZ\_opt \Leftarrow DZ\_new$;  $Dist\_opt \Leftarrow Dist\_new$
14:     end if
15:     $k \Leftarrow k+1$
16: end while
17: return $DZ\_opt$
```

distances, and therefore is expected to be a good initial solution. ‘Manhattan_Distances’ operator calculates the total communication distances in this zone. Then ‘Perturbation’ operator makes slight adjustments to this initialized zone to explore better grouping solution, according to a heuristic procedure described in Algorithm 1. For each adjustment, the heuristic compares the total distances between the initialized zone and the adjusted current zone. The heuristic not only accepts changes that improve the objective, but also some changes that deteriorate it. The latter are accepted probabilistically following simulated annealing algorithm. Above searching procedure is repeated until the termination condition is satisfied (the total distances of current zone is less than a pre-determined threshold).

B. Task Scheduling In One Zone

As mentioned above, the execution of tasks belonging to the same flow are restricted within one zone, because of no inter-zone dependency. Therefore after zoning, the next step is to schedule tasks in each zone. We formulate the Dynamic Task Scheduling (DTS) problem as a mixed-integer program (MIP). The objective is to achieve maximum system utilization with minimum communication cost under workload constraints. A mixed-integer program is an optimization model in which some of the decision variables (not all of them) have to be of type integer or binary. We choose the MIP solver integrated in LINGO software to solve the within-zone task scheduling problem. The solver is equipped with advanced heuristic implementation speeding the finding of feasible solutions on many difficult MIP problems.

To formulate the DTS problem into MIP form, we introduce a binary matrix M to represent all task-core mapping relationship. Let $V = \{v_1, v_2, \cdots, v_n\}$ denote a set of tasks to be executed within a zone of m cores. The task-core mapping matrix is thus of size $m \times n$. Each entry in this mapping matrix M_{ij} is a decision variable of binary type, with “1” representing task v_i assigned to the j-th core and “0” as the opposite situation. Fig. 3 shows an example of a task-core mapping matrix which assigns 6 tasks to 3 cores. The definition of this matrix obviously imposes the first set of constraints:

$$\sum_{i=1}^{m} M_{ij} = 1,\ \text{for } j = 1, 2, \cdots, n. \tag{8}$$

which indicates that one task can be assigned to only one core. Using this binary mapping matrix, we conveniently determine task assignments.

	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
Core 1	1	0	0	0	0	1
Core 2	0	0	1	0	1	1
Core 3	0	1	0	1	0	0

Fig. 3. An example of a task-core mapping matrix

Other than the mapping matrix, another set of decision variables are the starting times for each task, denoted by S_i's. Task starting times are combined with the task-core mapping matrix to determine the job sequences of all involved cores. An optimal schedule can be fully specified by determining the mapping matrix M_{ij}'s and task starting times S_i's: each row in the mapping matrix indicates a set of tasks to be executed on a particular core; the starting times help determine the execution sequence of all assigned tasks. Here we explicit the second set of constraints in this optimization model, in order to keep the precedence relationships among tasks. For each arc (i, j) of the task graph, the precedence relationship requires that task i must be finished before the execution of its successor j:

$$S_i + T_i + T_{comm}(i, j) \leq S_j, \tag{9}$$

where S_i and T_i represents the starting time and execution time for task v_i respectively, while T_{comm} represents the communication time between task i and j.

The third set of constraints are workload constraints imposed by core capacity rates. Capacity rate is included as an upper bound limit for workload assigned to each core. A stressed core with low capacity rate indicates that light workload should be assigned to this core. the proposed DTS method dynamically scales down core operating frequency and thus leads to varying task execution time. T_i for task i depends on which core this task resides in. Therefore we need to take into account the scaling ratio when evaluating the assigned workload on a particular core. Let α_{ij} denote the frequency scaling ratio when core i is processing task j. For each core within the zone, its assigned workload can be estimated as:

$$WL_i = \frac{\sum_j \alpha_{ij} T_j, \ \forall M_{ij} = 1}{T_s} \leq CR_i, \quad \text{for } i = 1, \cdots, m. \tag{10}$$

WL_i in (10) records the assigned workload on i-th core. T_s represents the length of schedule, i.e. the execution time on the worst-case path. The value of core's workload lies into the interval $[0, 1]$, and therefore can be compared with its capacity rate. The workload assigned to each core should not exceed its capacity rate.

We now discuss the objective functions in the proposed DTS method. For an efficient system, it is important to achieve high utilization. *Core Utilization* is measured as the ratio of its busy time to its total active period of time. One goal in our optimization model is to optimize the overall system utilization summed over all cores. Under core workload constraints imposed by capacity rates, this total utilization is also bounded by total capacity rate of the zone.

Another goal in this DTS scheme is to minimize the total communication cost among the cores. In a task graph with different rates of data production and consumption [17], some data tokens have to be stored in the buffer on the arc. Therefore the communication cost consists of two components:

$$T_{comm} = T_{trans} + T_{buff}. \tag{11}$$

where T_{trans} denotes total transmitting cost, and T_{buff} denotes the total buffering cost (or storage cost). For each arc (i, j) in the task graph, the transmitting cost is computed as the multiplication of the number of token transmitted and the unit time to transmit one token. The total transmitting cost is summed up over all arcs with buffering operations:

$$T_{trans} = \sum_{(i,j)} N_c(i, j)\, c(i, j) \tag{12}$$

where $N_c(i, j)$ is the number of tokens transmitted on arc (i, j), $c(i, j)$ is the unit transmission cost on (i, j). The buffering cost is calculated and accumulated in a similar way:

$$T_{buff} = \sum_{(i,j)} N_b(i, j)\, b(i, j) \tag{13}$$

where $N_b(i, j)$ denotes the number of tokens to be buffered on arc (i, j) or loaded from (i, j), and $b(i, j)$ is the unit buffering cost. In case that the number of input data tokens is greater than that of output data tokens, $b(i, j)$ indicates the time it takes to buffer one unit token on (i, j). Under the opposite condition, $b(i, j)$ denotes the time it takes to load one token from the buffer on arc (i, j).

To sum up, the DTS problem in our workload balancing framework can be generalized by the following mixed-integer program:

$$\begin{aligned}
\text{minimize} \quad & \alpha \cdot \left(\sum_{i=1}^{m} (1 - U_i) \right) + \beta \cdot T_{comm} \\
\text{subject to} \quad & \sum_{j=1}^{n} M_{ij} = 1, \ \forall j = 1, 2, \cdots, n \\
& S_i + T_i + T_{comm}(i, j) \leq S_j, \ \forall (i, j) \subset E \\
& WL_i(M, S) \leq CR_i, \ \forall i = 1, 2, \cdots, m \\
\text{variables} \quad & M = [M_{ij}]_{m \times n}, \ S = \{S_1, S_2, \cdots, S_n\}
\end{aligned} \tag{14}$$

where U_i represents the recorded utilization of i-th core, T_{comm} calculates the total communication cost based on the schedule, α and β are simply two weighting factors. The decisions variables are the task-core mapping matrix, with each entry M_{ij} constrained to be a binary value, and the task starting times. The first constraint in this optimization model is to guarantee the uniqueness of task-core mapping relationship. The second constraint satisfies all the precedence relationships among the tasks. The third constraint reflects the workload constraint induced by NBTI introduced capacity rate, where WL_i denotes the assigned workload on i-th core, which is dependent on the mapping relationships and task starting times (refer to (10)), and CR_i represents its corresponding capacity rate.

C. NBTI-Aware Workload Balancing

This section discusses how to dynamically spread the workload across the entire network based on the proposed DZ and DTS methods. The balancing strategy is adaptive to the frequent update of core capacity rate. As explained in Section II, core capacity rate varies at different time points. In most situations, some of the cores may be in "quasi-defect" situations as they are over-stressed. Under this situation, the over-stressed cores cannot be assigned heavy workload at that moment. On the other hand, when they are released at a later time, they may be available again. In this sense, the capacity rate for a core is not a constant number but has to be updated frequently.

In subsequent part we focus on two important moments to explain our proposed workload balancing policy: the moment when a new flow comes in, and the moment when a flow is finished execution.

When a new flow comes into the system, the DZ algorithm takes effect immediately to generate an appropriate zone to allocate the

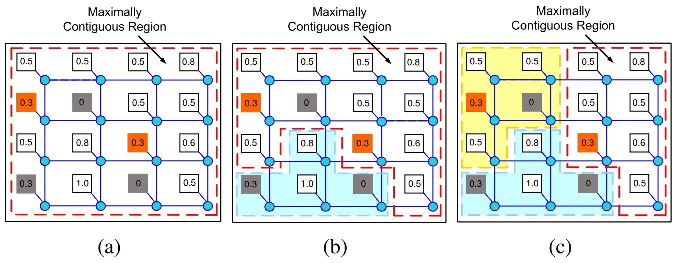
Maximally Contiguous Region
Maximally Contiguous Region
Maximally Contiguous Region
(a)
(b)
(c)

Fig. 4. The generation of a new zone

flow tasks. The DZ algorithm first searches for available cores and explores the maximally contiguous region. Starting from the bottom-left corner of the region, the DZ algorithm determines the optimal grouping solution according to the heuristic described in Algorithm 1. After the cores are grouped in a zone for task execution, the DTS algorithm is responsible for mapping the tasks onto particular cores within this zone. Note that due to the generation of a new zone, the maximally contiguous region will then be updated. The procedure of generating a new zone is illustrated in Fig. 4.

The case of relaxing a zone is relatively simply. When a zone finishes processing all assigned tasks, all the cores within this zone will be relaxed to join other available cores for the processing of new flows. Therefore the maximally contiguous region will be changed in next search iteration. This is done by a merging procedure. Moreover, after a period of task execution, the capacity rates of all involved cores also will be updated. Fig. 5 illustrates the relaxation procedure of an existing zone.

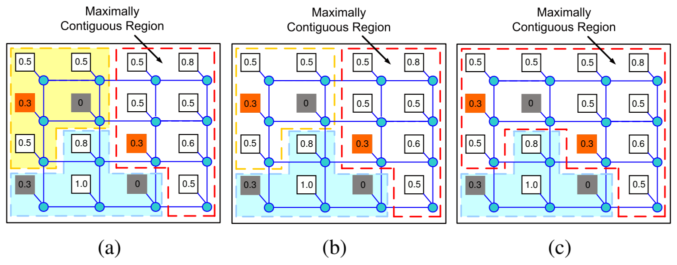
Maximally Contiguous Region
Maximally Contiguous Region
Maximally Contiguous Region
(a)
(b)
(c)

Fig. 5. The relaxation of an existing zone

IV. Experimental Results

This section presents the results of the proposed workload balancing methodology. We consider a 8×8 multi-core system. We choose 8/64 nodes to be stressed ($\sim$12.5%). To demonstrate the effectiveness of our proposed approach, all task graphs are generated from realistic applications from several benchmark suites, including MediaBench, MiBench, NetBench and EEMBC. Table I lists the scheduling results at 90 seconds after the start of simulation. This table includes core index, zone number each core is grouped in, each core's capacity rate and its assigned workload. Note that only the stressed cores are listed in Table I. In addition, the scheduling results at 100 seconds are further presented in Table II. We can observe that for each stressed core, its assigned workload is well bounded by its capacity rate. The tables show that those cores stressed at 90 seconds have been relaxed at 100 seconds, while a new group of cores alternate into stressed phase. The experimental results demonstrate that capacity rate is an indication of upper bound limit one core can accept workload. The DZ algorithm together with DTS algorithm effectively balances the workloads among the stresses cores.

TABLE I
THE SCHEDULING RESULTS FOR STRESSED CORES AT 90S

Stressed Core Index	Zone # Grouped in	Capacity Rate	Assigned Workload
9	1	0.7	0.6248
12	1	0.5	0.5000
28	3	0.5	0.3892
33	3	0.4	0.4000
37	6	0.4	0.4000
39	6	0.5	0.4473
50	7	0.8	0.6482
57	7	0.9	0.8079

TABLE II
THE CAPACITY RATE AND CORE UTILIZATION FOR THE STRESSED CORES AT 100S

Stressed Core Index	Zone # Grouped in	Capacity Rate	Assigned Workload
14	3	0.4	0.3514
25	2	0.4	0.4000
31	5	0.5	0.5000
38	5	0.5	0.3892
41	6	0.7	0.5864
44	4	0.3	0.3000
52	6	0.6	0.5536
58	6	0.5	0.4293
59	6	0.2	0.2000

To compare system performance in execution time, we initially ran simulations at a normal load up to 10000 cycles, then reduce the load by 50% every 10000 cycles, i.e. at 10000 cycles the load is 0.5 of the offered load, and at 20000 cycles it will be 25% of the offered load. We terminate generation of new flows at 30000 cycles. We observe how long the task flows will run in such a 8×8 multi-core system. Here, while the stressed load on 8/64 nodes is reduced, other nodes can operate at the maximum rate. Each stressed core is able to execute these tasks at certain frequency, which is associated with its capacity rate. We distribute the tasks by using the proposed DZ algorithm and evaluate the execution time by using the DTS algorithm. Fig. 6 illustrates the overall task execution times at different levels of offered network load. "Non-Stressed" represents simulation results

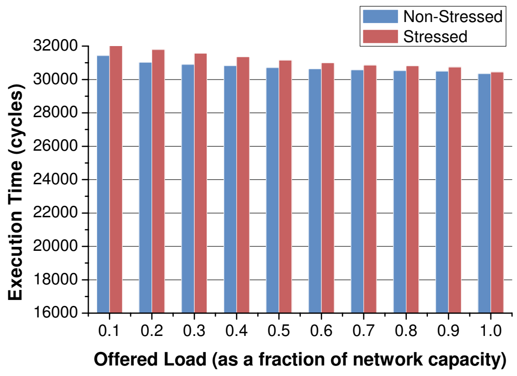

Fig. 6. Comparison of execution times between non-Stressed and stressed cases

without considering NBTI introduced stress and "Stressed" considers NBTI impact. An insignificant increase in execution time (<2%) can be observed when the offered load is above 0.2. When offered load increases beyond 0.5, very minor differences can be observed between the ideal "Non-stressed" and "Stressed" cases (<1%). For the worst case of offered load as 0.1, approximately a 3% increase in execution time is obtained.

To demonstrate the efficiency of our proposed methodology, we increase the number of stressed cores and observe the increase in system performance degradation. Fig. 7 shows the comparison of task execution time between “Stressed” and “Non-Stressed” cases. Here the offered load is fixed at full rate. Starting with 8 stressed cores, the execution time is almost identical before and after applying the NBTI stress model. As the number of stressed cores increases, a slight increase in the overall execution time can be observed. When the number of stressed cores grows to 16, an insignificant performance drop (approximately 3%) is observed, which is still acceptable.

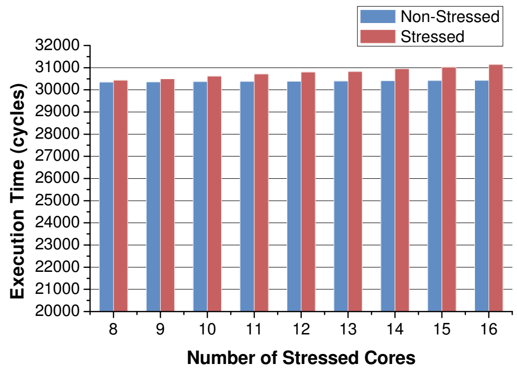

Fig. 7. Comparison of execution times with different number of stressed cores

Fig. 8 displays core failure percentage with regard to time. The x-axis represents the time in terms of year and y-axis represents the percentage of core failure. The red solid line represents the result of our new approach (denoted as “New”) while the blue dash line represents the case without (“Old”) the new approach. The difference in terms of yield becomes obvious after 2 years and begins to widen. We used a Monte-Carlo simulation in SPICE to monitor

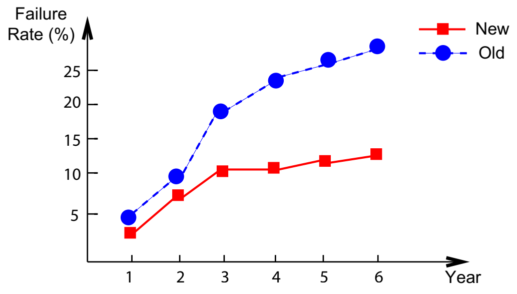

Fig. 8. Core failure percentage comparison between new strategy and without new strategy

the critical path delay and total leakage power for each core, and predicted the changes in MTTF. The MTTF computation models come from [15], including a variety of MTTF estimation approaches with regard to core activity in terms of workload. Fig. 9 shows the MTTF comparison between multi-core systems without the proposed methodology (“Old”) and with the proposed methodology (“New”). The x-axis presents the time in terms of years of operation. The y-axis is the MTTF result that shows the average MTTF of a 8/64 multi-core system. Though after about 3 years both cases observe decreases in MTTF. The “New” one shows about 30% less changes.

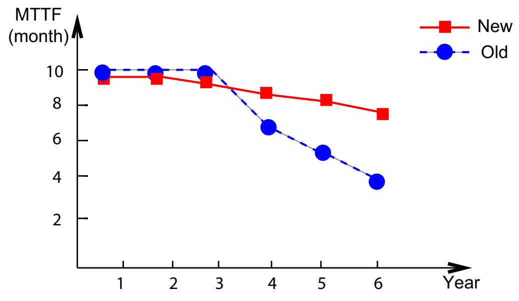

Fig. 9. MTTF comparison between new strategy and without new strategy

V. Conclusion

This paper presents a new design framework for multi-core system to include device wear-out impact. The new approach starts from device fractional NBTI model to evaluate core performance differences, providing a new NBTI-aware system workload model based on new DZ and DTS algorithms to balance workload among active cores while relaxing stressed ones.

References

[1] K. Constantinides, S. Plaza, J. Blome, V. Bertacco, S. Mahlke, T. Austin, B. Zhang, and M. Orshansky, “Architecting a reliable CMP switch architecture”, *ACM Trans. Architecture and Code Optimization*, vol. 4, no. 1, pp. 1-37, 2007.

[2] W. Wang, S. Yang, S. Bhardwaj, R. Wattikonda, S. Vrudhula, F. Liu, Y. Cao, “The impact of NBTI on the performance of combinational and sequential circuits”, in *Proc. DAC*, 2007, pp. 364-369.

[3] S. Bhardwaj, W. Wang, R. Vttikonda, Y. Cao, S. Vrudhula, “Predictive modeling of the NBTI effect for reliable design”, in *Proc. CICC*, 2006, pp. 189-192.

[4] J. JAbella, X. Vera, and A. Gonzalez, “Penelope: The NBTI-Aware Processor”, in *Proc. International Symposium on Microarchitecture*, 2007, pp.85-96.

[5] Han Zhao, Xinxin Liu and Xiaolin Li, “DLBEM: Dynamic load balancing using expectation-maximization”, in *Proc. IPDPS*, 2008, pp. 1-7.

[6] G. Cybenko, “Dynamic load balancing for distributed memory multiprocessors”, *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279-301, 1989.

[7] Y. F. Hu and R. J. Blake, “An improved diffusion algorithm for dynamic load balancing”, *Parallel Computing*, vol. 25, no. 4, pp. 417-444, 1999.

[8] D. Gu, L. Yang and L. R. Welch, “A predictive, decentralized load balancing approach”, vol.3, pp. 131b, in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[9] A. Cortes, A. Ripoll, M. A. Senar and E. Luque, “Dynamic load balancing strategy for scalable parallel systems”, in *Proc. International Conference on Parallel Computing*, 1997, pp. 735-738.

[10] E. Karl, P. Singh, D. Blaauw, D. Sylvester, “Compact in-situ sensors for monitoring Negative-Bias-Temperature-Instability effect and oxide degradation”, in *Proc. ISSCC*, 2008, pp. 410-623.

[11] M. Mani, A. Devgan, and M. Orshansky, “An efficient algorithm for statistical minimization of total power under timing yield constraints”, in *Proc. DAC*, 2005, pp. 309-314.

[12] H. Chang and S. S. Sapatnekar, “Statistical timing analysis considering spatial correlations using a single PERT-like traversal”, in *Proc. ICCAD*, 2007, pp. 621-625.

[13] J. Sun, A. Kodi, A. Louri, and J. M. Wang, “NBTI aware workload balancing in multi-core systems”, in *Proc. ISQED*, 2009, pp. 833-838.

[14] K. Waldschmidt, J. Haase, A. Hofmann, M. Damm and D. Hauser, “Reliability-aware power management of multi-core systems (MPSoCs)”, in *Proc. Dynamically Reconfigurable Architectures*, 2006.

[15] Y. Liu, W. Tang and R. Zhang, “Reliability and mean time to failure of unrepairable systems with fuzzy random lifetimes”, *IEEE Trans. Fuzzy Systems*, vol. 15, no. 5, pp. 1009-1026, 2007.

[16] H. El-Rewini, T. G. Lewis and H. H. Ali, *Task Scheduling in Parallel and Distributed Systems.* Prentice Hall, 1994.

[17] E. A. Lee, and D. G. Messerschmitt, “Synchronous data flow”, in *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235-1245, 1987.