Dynamic Cache Tuning for Efficient Memory Based Computing in Multicore Architectures

Hadi Hajimiri, Prabhat Mishra
CISE, University of Florida, Gainesville, USA
{hadi, prabhat}@cise.ufl.edu

Swarup Bhunia
EECS, Case Western Reserve University, Cleveland, USA
skb21@case.edu

***Abstract*—Memory-based computing (MBC) is a promising approach to improve overall system reliability when few functional units are defective or unreliable under process-induced or thermal variations. A major challenge in using MBC for reliability improvement is that it can introduce significant energy and performance overhead. In this paper, we present an efficient dynamic cache reconfiguration and partitioning technique to improve performance and energy efficiency in MBC-enabled reliable multicore systems. We use genetic algorithm to search effectively in a large and complex design space. Experimental results demonstrate that the proposed cache reconfiguration and partitioning approach can significantly improve both performance and energy efficiency for on-demand memory based computing without sacrificing reliability.**

I. Introduction

Ensuring long-term reliability is an essential goal for all microprocessor manufacturers. With device miniaturization, design and process error margins are shrinking. Increasing process-induced variations and high defect rate in nanometer regime lead to reduced yield [1]. Process variation affects the propagation delay in CMOS circuits, which may lead to delay failures. In Static Random Access Memory (SRAM) these variations may cause data retention or read/write failures [8]. Existing approaches address reliability concerns during design time or by post-silicon correction and compensation solutions. Increasing reliability using redundant functional units at design time places a significant permanent area/energy overhead on all manufactured chips in their entire lifetime.

Memory-based computing (MBC) has been shown to be a promising alternative to improve system reliability in the presence of both manufacturing defects and parametric (process or thermal-induced) failures. It assumes that one or more logic units can be nonfunctional while on-chip memory is functional. Although memory is also prone to errors, it is easy to handle failures using error correcting code (ECC) based protection (soft memory errors), suitable redundancy, and remapping techniques [1] (parametric failures). Unfortunately, such low-overhead methods are not suitable for handling defects in processor logic circuits. MBC is promising to address reliability concerns in functional units with significantly less area overhead [15] (only 9.5% compared to duplication based redundancy methods). Existing studies have demonstrated the utility of MBC in both single-core [15] and multicore [3] architectures. The existing approaches have two major limitations. First, the cache area limitation was not considered as a constraint in system design and the study assumed the presence of large caches. Storing lookup tables for MBC may need huge storage capacity. For example, authors in [15] used large dedicated caches in their setup in order to lower performance overhead. Second, allocating cache space to MBC lookup tables can result in significant performance/energy overhead. For example, Fig. 1 shows that improving reliability by using MBC slows down performance significantly when we poorly choose the MBC L2 cache usage for *applu* benchmark leading to a similar increase in cache subsystem energy consumption[1]. In this example we used an 8-way associative L2 cache and restricted the L2 cache ways accessible by the application from 1 to 8.

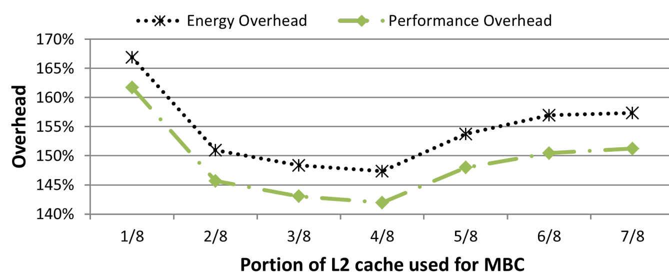

Fig. 1. The impact of reduced available cache for instruction/data on performance for benchmark *applu*. The remaining is used by MBC.

Dynamic cache reconfiguration (DCR) is widely known as one of the most effective techniques for cache energy optimization [2]. By tuning the cache configuration at runtime, we can satisfy data and instruction memory access behavior as well as MBC requirements of different applications so that significant amount of energy savings can be achieved without violating deadline constraints. Cache partitioning (CP) is another promising approach to improve performance and energy consumption. DCR and CP are promising techniques that can be used to alleviate the performance/energy overhead. However, there are several challenges. It is difficult to determine a profitable portion of cache that could be set aside for MBC. Increasing MBC cache space improves MBC performance while reduces the available space for instruction and data and may decrease the overall performance. We need to determine a profitable partitioning between instruction/data cache and MBC cache. When MBC supports many operations it is a major challenge to find the size of profitable dedicated space for each supported operation as the number of possible solutions increases exponentially.

In this paper, we present an energy and performance-aware dynamic cache reconfiguration and partitioning technique for improving reliability in multicore embedded systems. We developed an efficient genetic algorithm to utilize DCR and CP tech-

This work was partially supported by NSF grants CNS-0915376, CCF-0964514 and ECCS-1002237.

[1]Performance overhead (in percentage) is calculated by ((execution time of the application using MBC)/(execution time of the application without MBC)-1)x100

niques to find beneficial IL1/DL1 cache configurations as well as L1/L2 cache partition factors for each MBC operation and L2 instruction/data partition factors. Our experimental results demonstrate that our approach can significantly improve both performance and energy efficiency for on-demand computing in memory in the presence of deadline constraints.

The rest of the paper is organized as follows. Section II describes related research activities. Section III provides an overview of our memory based computation framework along with DCR and CP. Section IV describes the proposed genetic algorithm for reliability and energy improvement using DCR and CP. Section V presents the experimental results. Finally, Section VI concludes the paper.

II. Related Work

Reliable computation using unreliable components has been actively studied for a long time. A wide variety of solutions have been proposed over the years with the goal of dynamic detection and correction of defects and variation-induced failures [1][2], [6]. These techniques typically incur large performance overhead or do not address manufacturing defects [6]. Paul et al. [15] studied the utility of MBC for reliability improvement in single-core architectures. MBC is recently introduced in multicore architectures [3] where private MBC L1 caches were used to store MBC look up tables in each core. The existing approach has two major limitations. First, it assumed MBC cache can be easily added to the design and did not consider the associated area overhead (the use of large cache sizes) as storing lookup tables for MBC may need huge storage capacity. Second, allocating cache space to MBC lookup tables can result in significant performance overhead due to reduced available space for normal instruction and data. In this work, we have efficiently used DCR and CP to address the above challenges. While DCR and CP have been explored for performance and energy improvement in various contexts [2][9], these approaches cannot be applied to MBC due to the complexity of the design space, as we discuss in Section IV. To the best of our knowledge this is the first work that utilized DCR and cache partitioning to enable efficient memory based computing.

III. System Model and Problem Formulation

A. Architecture Model

Fig. 2 shows an overview of MBC in a multicore framework. Under normal circumstances, issue logic sends the instructions to the respective functional units. However, if the functional unit is not available (due to temperature stress), for certain types of instructions (addition, multiplication, etc.), issue logic bypasses the original functional unit for memory based computation. The operands are used to form the effective physical address for accessing the LUTs corresponding to the mapped function. The LUTs are stored in main memory and most recent accesses are cached for performance improvement [15].

In this paper, we investigate the role of cache partitioning in improving performance without sacrificing reliability. We have applied MBC to realize the functionality of the integer execution unit (adder and multiplier) in each core. This architecture has **m** cores each having it's own private L1 data and instruction caches. All the cores share an L2 combined (instruction+data) cache which is connected to main memory. Instruction and data L1 caches are highly reconfigurable in terms of effective capacity, line size and associativity. We adopt the underlying reconfigurable cache architecture used in [2].

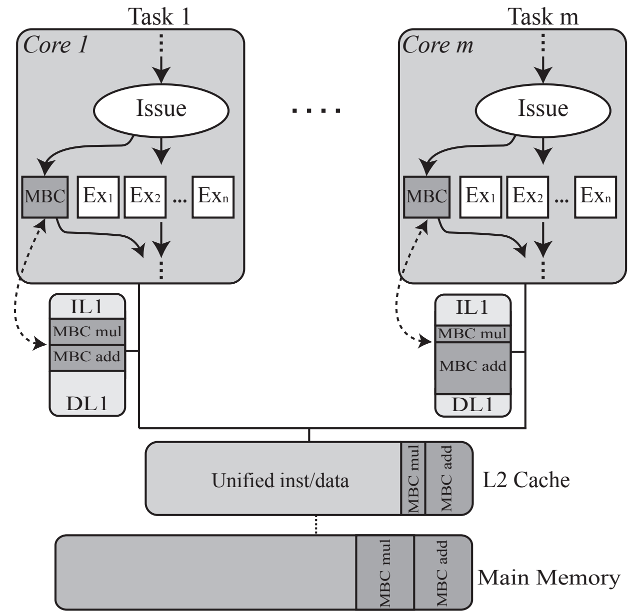

Fig. 2. Memory-based computing architecture in multicore systems

In our framework, both private L1 cache associated with each core and the unified shared L2 cache can be partitioned. Unlike traditional LRU replacement policy which implicitly partitions each cache set on a demand basis, we use a way-based partitioning in the shared cache and private MBC caches [11]. For example, in Fig. 3, 5 ways are reserved for normal instruction/data caches, whereas multiply and addition LUTs (for MBC) received 1 and 2 ways, respectively. We refer the number of ways assigned to each functionality as its *partition factor*. For example, the L2 partition factor for instruction/data cache in Fig. 3 is 5.

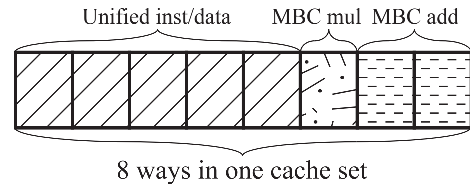

Fig. 3. Way-based cache partitioning example: 5 ways for inst/data, 1-way of MBC mul, and 2 ways for MBC add.

To support MBC, each core also has an L1-level MBC cache that stores most frequently accessed entries of the LUTs. The existing private L1 cache can be partitioned into two parts: one part dedicated for MBC cache to store most frequently used LUTs, and the other part will be used for conventional data/instruction accesses. For example, in Fig. 2 *core1* uses half of private MBC cache for each of MBC operations whereas *core m* needs less than half for *mul* operation (assigning more to *add* operation). Similarly, shared L2 cache can be partitioned to make space for MBC LUTs.

B. Problem Formulation

In this work, we use static cache partitioning as applications are known a priori. In other words, partition factors are predetermined during design time and remain the same throughout the system execution. Dynamic profiling and cache partitioning [9] requires online monitoring, runtime analysis and sophisticated OS support, therefore, may not be feasible for a wide variety of systems. Furthermore, embedded systems normally have highly deterministic characteristics (e.g., task release time, deadline, input set), which make off-line analysis most suitable. By static profiling, we can potentially search much larger design space and thus achieve better optimization results. We statically profile each set of tasks and store the analysis results in a profile table which is fully utilized to dynamically reconfigure the cache hierarchy at runtime. In our framework we tune a number of different cache parameters. Parameters for IL1 and DL1 caches in each core are comprised of cache size, line size, and associativity. The available cache ways in the unified shared L2 cache are divided into partitions for normal inst/data and different MBC operations. Similarly, we set aside part of DL1 caches (in each core) for MBC operations. The system we consider can be modeled as:

- A multicore processor with m cores $\mathcal{P}\{c_1, c_2, ..., c_m\}$.
- A list of operations supported by MBC $\mathcal{F}\{o_1, o_2, ..., o_l\}$
- A β-way associative shared L2 cache with way-based partitioning enabled.
- A set of n independent tasks $\mathcal{T}\{\tau_1, \tau_2, ..., \tau_n\}$ with a common deadline D[2].

Suppose we are given:

- A task mapping $\boldsymbol{M} : \mathcal{T} \rightarrow \mathcal{P}$ in which tasks are mapped to each core. Let ρ_k denote the number of tasks on core c_k.
- An L1 cache configuration assignment $\boldsymbol{R} : C_I, C_D \rightarrow \mathcal{T}$ in which one IL1 and one DL1 configuration are assigned to each task. This mapping determines the associativity of DL1 cache, namely α_k for core c_k, for each core and delimits the maximum size of MBC cache that can be stored in L1 MBC caches. The DL1 cache supports way-based partitioning.
- Private L1 MBC cache partitioning scheme for each operation $o_i \in \mathcal{F}$:

$$\boldsymbol{P}^p_{Ck}\{f^p_{O1,k}, f^p_{O2,k}, ..., f^p_{Ol,k}, f^p_{d,k}\}, \forall k \in [1, m]$$

  in which core $c_k \in \mathcal{P}$ allocates $f^p_{Oi,k}$ ways of the private DL1 cache to MBC operation o_i and $f^p_{d,k}$ ways to normal data.
- Shared L2 cache partitioning scheme $\boldsymbol{P}^s\{f^s_1, f^s_2, ..., f^s_l, f^s_{ID}\}$ in which operation $o_i \in \mathcal{F}$ is allocated f^s_i ways of the shared L2 cache and f^s_{ID} is assigned for instruction and data.
- Task $\tau_{k,i} \in \mathcal{T}$ (i^{th} task on core c_k) has execution time of $t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s)$. Let $E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s)$ and $E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s)$ denote the total energy consumption of all the L1 caches and the shared L2 cache, respectively.

[2]Our approach can be easily extended for individual deadlines.

Our goal is to find $\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p$, and $\boldsymbol{P}^s$ such that the overall energy consumption of cache subsystem for the task set:

$$E = E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s) + E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s)$$

is minimized subject to:

$$Max(\sum_{i=1}^{\rho_k} t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}^p, \boldsymbol{P}^s)) \leq D, \forall k \in [1, m], \forall i \in [1, \rho_k] \quad (1)$$

$$\sum_{i=1}^{l} f^p_{Oi,k} = \alpha_k - f^p_{d,k}; f^p_{Oi,k} \geq 0, \forall k \in [1, m] \quad (2)$$

$$\sum_{i=1}^{l} f^s_i = \beta - f^s_{ID}; f^s_i \geq 0, \forall k \in [1, m] \quad (3)$$

Equation (1) guarantees that all the tasks in $\mathcal{T}$ are finished by the deadline D. Equation (2) ensures that the L1 partitioning $\boldsymbol{P}^p$ is valid (l is the number of supported MBC operations). Equation (3) guarantees that at most β ways of the shared L2 cache is assigned for MBC.

IV. Dynamic Cache Reconfiguration and Partitioning Algorithm

Our goal is to profile the entire task set $\mathcal{T}$ under all possible combinations of $\boldsymbol{P}^p$ and $\boldsymbol{P}^s$. Unfortunately, this exhaustive exploration is not feasible due to its excessive simulation time requirement (number of simulations can easily go over billions). We have developed a genetic algorithm (GA) to overcome this problem. Genetic algorithms transpose the notions of natural evolution, such as inheritance, mutation, selection, and crossover to the world of computers, and imitate natural evolution. They constitute a class of search methods especially suited for solving complex optimization and design problems [12]. Fig. 4 shows the major steps of our genetic algorithm. In step 1, the initial population is filled with individuals that are generally created at random. In step 2, each individual in the current population is evaluated using the fitness measure. Step 3 tests whether the termination criteria is met. If so the best solution in the current population is returned as our solution. If the termination criteria is not satisfied a new population is formed by applying the genetic operators in step 4. Each iteration is called generation and is repeated until the termination criteria is satisfied. We describe each of these steps in the following subsections with illustrative examples.

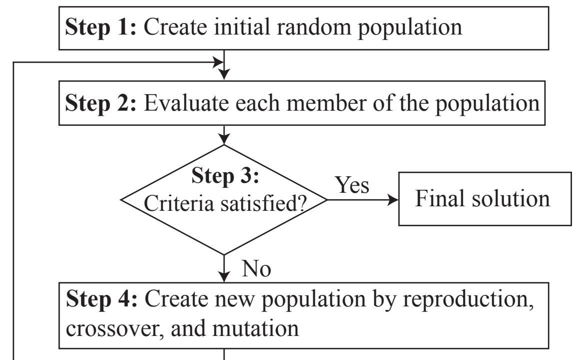

Fig. 4. Overview of our genetic exploration algorithm

A. Creation of Initial Random Population

In the first step, the initial population is filled with individuals that are randomly created. We initially choose random acceptable values for IL1/DL1 cache configurations (size, associativity, and line size) and partitioning factors (for MBC operation o_1, ..., o_l and inst/data) as well as L2 partitioning factors. We create small number of initial solutions (composing the initial population). For the ease of illustration let's assume the size of IL1 and DL1 are the only tunable parameters in a singlecore system with only one task and deadline of 10 milliseconds. Solutions are presented in tuples as (IL1_size,DL1_size) in bytes. Each tuple is associated with another tuple representing its execution time and energy consumption, e.g. (IL1_size,DL1_size):(time,energy). Consider (2048,4096), (4096,1024), and (2048,1024) are randomly generated in this step as the initial population.

B. Evaluation of Each Member in the Population

Our problem is a single-objective optimization problem in which the selection is done proportional to the fitness function. Each individual in the current population is evaluated using the fitness measure. We define our fitness value based on:

$$\Psi = E: \textit{ and Equations } (1), (2), \textit{ and } (3) \textit{ must hold}$$

The lower fitness measure implies better solution (lower energy consumption). In this step, we profile (using an architectural simulator) each solution in the population and sort them based on energy consumption. The exploration goal is to find the solution with minimum energy consumption. We remove solutions that do not satisfy the deadline from the current population. In our example (2048,4096):(9,20), (4096,1024):(8,24), and (2048,1024):(10.5,18) are the outcome of this step. Solution (2048,4096):(9,20) represents that with the assignment of 2048 and 4096 for IL1 and DL1 respectively the task takes the execution time of 9 milliseconds and consumes 20 nano Joule. As the execution time of solution (2048,1024):(10.5,18) is larger than the deadline, this task will be removed from the current population.

C. Termination Criteria Check

If the termination criteria is met, the best solution is returned. We set the criteria to be a fixed number of generations (in this work 20). Note that the solution can converge prior to this time when the current population is the same as previous generation. In this case, the best solution in the current generation is returned as our final solution.

D. Generation of New Population

In this step, we create new candidates that are slightly different from the individuals in the current population with the aim of finding more energy efficient solutions. From the current population individuals are selected based on the previously computed fitness values (lowest energy solutions). A new population is formed by applying the genetic operators (reproduction, crossover, and mutation) to these individuals. The selected individuals are called parents and the resulting individuals are referred as offspring. The genetic operators are as follows:

Reproduction: A part of the new population can be created by simply copying (without change) selected individuals from the present population. This gives the possibility of survival for already developed fit solutions, e.g. most energy efficient ones.

Crossover: New individuals are created as offspring of two parents. One or more so-called crossover points are selected (usually at random) within the chromosome of each parent, at the same place in each. In this paper, we use uniform crossover with the mixing ratio of 0.5 in which the offspring has approximately half of the genes (IL1/DL1 configurations and partitioning, L2 partitioning) from first parent and the other half from second parent. The parts delimited by the crossover points are then interchanged between the parents, as shown in Fig. 5. The individuals resulting in this way are the offspring.

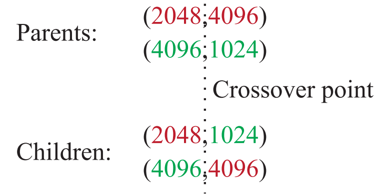

Fig. 5. An example of one point crossover

Mutation: A new individual is created by making modifications to one selected individual. We increase/decrease IL1/DL1 cache size (e.g., multiply/divide by 2), associativity, and line size. We also increase the number of the cache ways allocated to a particular MBCoperation (or instruction/data) for both L1 and L2. These cache ways are deducted from the ways assigned to other operations so that Equation (2) and (3) hold and we have a valid partitioning factor assignment. Fig. 6 shows mutation of individuals in current population using our simple example. Eight new individuals are generated in this step. In this figure, green color represents a reduction in cache size and blue shows cache enlargement. Note that the new solution will be ignored if it is not valid. For example, since the smallest possible DL1 cache is 1024, the solution (2048,512) (underlined in the figure) is skipped from the new population.

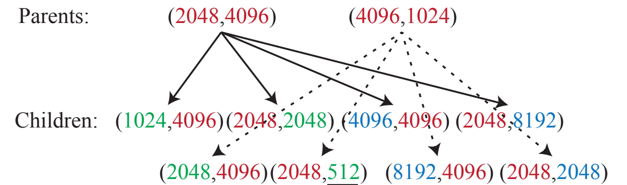

Fig. 6. Genetic mutation

Applying every genetic operator to all individuals in the current population leads to a huge number of new individuals as the number of tunable parameters increases. This makes the evaluation step very slow because a large number of simulations would be needed. Therefore by using different probabilities for applying these operators, we control the number of newly generated solutions in each iteration hence the speed of convergence.

V. Experiments

A. Experimental Setup

To evaluate the effectiveness of the proposed approach, we implemented the computation transfer mechanism in a widely

TABLE I
MULTI-TASK BENCHMARK SETS.

	2-Core	4-Core
Set 1	mgrid,lucas	toast,lucas,vpr,parser
Set 2	vpr,qsort	qsort,bitcount,swim,lucas
Set 3	toast,dijkstra	lucas,mgrid,dijkstra,CRC32
Set 4	parser,toast	dijkstra, applu, parser, mgrid
Set 5	bitcount,swim	-
Set 6	toast,mgrid	-

used multicore simulator, M5 [14]. We enhanced M5 to make the required modifications in processor cores as well as in memory hierarchy. We modified memory hierarchy to support cache partitioning, to introduce L1 private MBC caches and shared L2 MBC cache. We configured the simulated system with a two-core and four-core processor each of which runs at 500MHz. The DerivO3CPU model [14] in M5 is used which represents a detailed model of an out-of-order SMT-capable CPU which stalls during cache accesses and memory response handling. A 128KB 16-way associative cache with line size of 32B is used for L2 cache. For both IL1 and DL1 caches, we utilized the sizes of 1 KB, 2 KB, 4 KB, and 8 KB, line sizes ranging from 16 bytes to 64 bytes, and associativity of 1-way, 2-way, 4-way, and 8-way. Since the reconfiguration of associativity is achieved by way concatenation [2], 1KB L1 cache can only be direct-mapped as three of the banks are shut down. 2KB cache can only be configured to direct-mapped or 2-way associativity. Therefore, there are 18 (=3+6+9) configuration candidates for L1 caches. For comparison purposes, we used the *base cache* configuration set to be a 4 KB, 2-way set associative cache with a 32-byte line size, a common configuration that meets the average needs of the studied benchmarks [2]. The memory size is set to 256MB. The L1 cache, L2 cache and memory access latency are set to 2ns, 20ns and 200ns, respectively. The temperature threshold for the integer execution unit was set at $100°C$.

We used benchmarks selected from MiBench [10] (*bitcount, CRC32, dijkstra, qsort,* and *toast*) and SPEC CPU 2000 [4] (*applu, lucas, mgrid, parser, swim,* and *vpr*). In order to make the size of SPEC benchmarks comparable with MiBench, we use reduced (but well verified) input sets from MinneSPEC [5]. Table I lists the task sets used in our experiments which are combinations of the selected benchmarks. We choose 6 task sets for 2-core and 4 task sets for 4-core scenarios, each core running one benchmark. The task mapping is based on the rule that the total execution time of each core is comparable. “Hotspot 2.0” tool [16] was used for estimating the temperature profile of the integer ALU units. In order to estimate the die thermal profile from Hotspot, power dissipation values of the individual functional units were obtained from Wattch 1.0 [7] at regular time intervals.

B. Results

Fig. 7 shows the yield improvement due to the proposed activity migration scheme in case of manufacturing defects. Performance overhead of more than 10% due to failing functional unit is defined as chip failure. We considered a specific implementation of integer and floating point units obtained from [13] which consisted of 600K transistors. The cache memory organized into 32x32 blocks with built-in redundancy of two columns per block. A random distribution of a range of defect rates were inserted into the transistors of both logic and memory units. Some defects were tolerated using redundant columns in memory. Performance degradation is obtained from cycle accurate simulations using our benchmark sets. Region A in Fig. 7 represents low defect rates. It can be observed that even for low defect rates the percentage of chips that do not meet the target performance using the baseline configuration (no MBC) is as high as 60%. For these low defect rates, however, the performance overhead for the MBC scheme was lower than the 10% tolerance (improvement of up to 60% in yield). As the defect rate increases (Region B), more functional units fail and less memory blocks are usable due to device failures. Hence, yield for the proposed scheme also starts to degrade. However, the chip failure rate is considerably better than the baseline configuration. In case of high failure rates (Region C), the MBC scheme also faces increased chip failure but still lower than the baseline. Results in Fig. 7 confirms that the proposed activity transfer method provides considerable benefit in yield and reliability at the expense of small loss in performance.

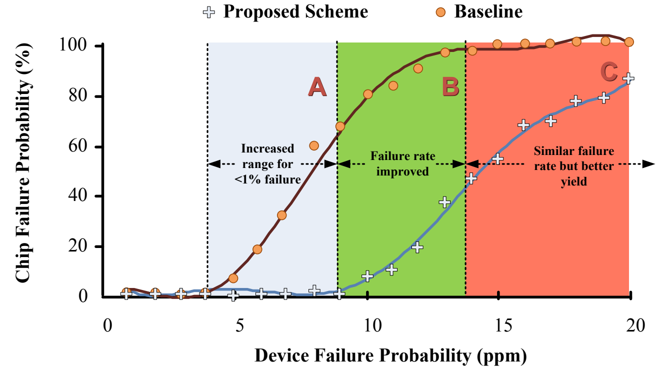

Fig. 7. Variation in chip failure probability for different device failure probabilities with and without the proposed approach.

Fig. 8 shows the energy improvements of our approach using MBC in the presence of thermal stress in 2-core framework. For comparison purposes we define a fixed sized MBC cache option *Fixed_Half*, a straightforward solution, where half of available cache is used (128KB of L2 and 2KB of L1 caches) for MBC for all task sets. As the search space in our problem is extremely large we compare our approach to a known search heuristic in which each cache parameter is optimized separately. For example, we start with *Fixed_Half* solution and replace IL1 configuration for core 0 in this solution with all possible configurations for IL1 and find the least energy one. Next, we find the least energy IL1 for core 1. Similarly we discover best IL1/DL1 configurations and partition factors for every core as well as L2 partition factors separately. We refer this method as *local optimum* in our comparison. Using *local optimum* provides up to 20% (15% on average) energy savings compared to *Fixed_Half*. Performing *local optimum* for a 2-core scenario requires 307 (=2x(18+18+45)+145) simulations. To make our comparison fair we limit the number of generations in our approach to 10 so that the total number of simulations would be less than 300 (each generation creates 30 new solutions at most). Our genetic algorithm achieves 26% energy savings on

average (up to 41% for task set 6).

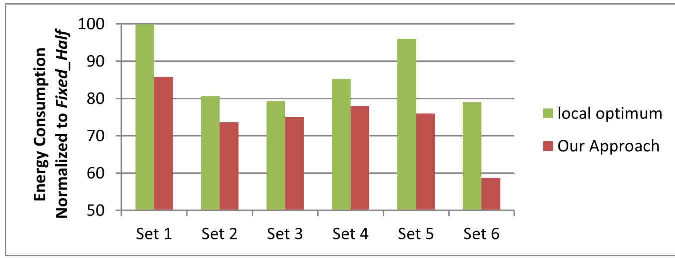

Fig. 8. Energy consumption for 2-core benchmark sets.

To investigate the speed of convergence in our approach we extended the number of generations to 20. Energy consumption of the most energy efficient solution found at each generation normalized to *Fixed_Half* is reported in Fig. 9. It can be observed that for the majority of task sets the energy consumption of the best solution at the 10th generation is very close to the solution at generation 20. Extension to generation 20 will achieve 2.4% more energy savings (up to 4.7% for Set 5). Fig. 10 illustrates energy consumptions in 4-core scenario. Applying *local optimum* in 4-core framework obtains 10% energy reduction compared to *Fixed_Half*. Utilizing genetic algorithm produces 18% energy savings on average (up to 24% using Set 4). We notice that the number of parameters (and thus the search space) increases exponentially with the number of cores making it more difficult to reduce energy consumption in a 4-core framework.

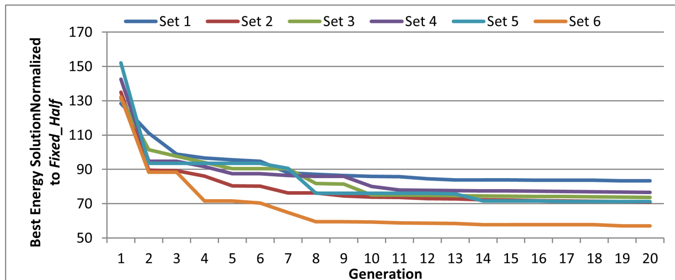

Fig. 9. Energy consumption of the least energy solution at each generation for 2-core benchmark sets normalized to *Fixed_Half*.

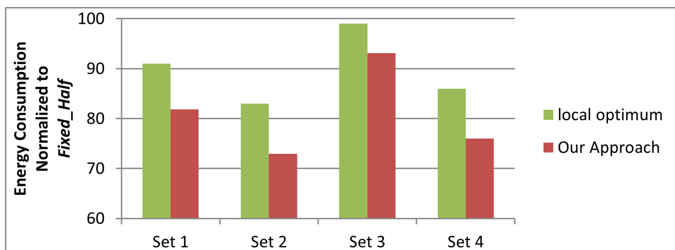

Fig. 10. Energy consumption for 4-core benchmark sets.

A closer look at performance overhead for different cache partitioning factors confirms that choosing a predetermined fixed partitioning factor for all benchmarks is not beneficial. Fig. 11 shows performance overhead for varying L2 instruction/data partitioning factor where partitioning factor for L2 MBC, L1 instruction/data and MBC are fixed to half of the available space. For instance it is beneficial to choose partitioning factor 7 for *vpr* benchmark whereas partition factor 13 generated the lowest overhead for *patricia*.

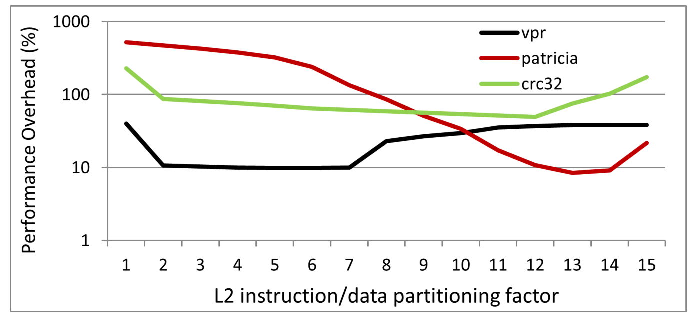

Fig. 11. Performance overhead of various partitioning factors for *vpr* , *patricia,* and *crc32* benchmark.

VI. Conclusion

We presented a novel energy/performance-aware cache partitioning technique for improving reliability with memory based computing. The basic idea is to use both private and shared caches as reconfigurable computing resources. We developed an efficient static profiling technique and cache partitioning algorithm to find beneficial L1/L2 cache partition factors for each MBC operation and L2 instruction/data partition factors. Our approach can be effectively used to tolerate permanent manufacturing defects in a processor core to improve functional yield of multicore architectures. It can also be applied to temporarily bypass the activity in functional units under time-dependent local variations, thus providing dynamic thermal management by activity migration. Our experimental results demonstrated significant reduction in energy consumption (down to 74% on average) when using MBC for reliability improvement in multicore architectures.

References

[1] A. Agarwal et al, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies", *IEEE Trans. on VLSI*, 13,27-38, 2005.

[2] W. Wang et al., "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multicore Systems", *DAC*, 2010.

[3] H. Hajimiri et al, " Reliability Improvement in Multicore Architectures Through Computing in Embedded Memory", *MWSCAS*, 2011.

[4] Spec 2000 benchmarks [Online], *http://www.spec.org/cpu/*.

[5] A. KleinOsowski and D. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research", *CAL g(1)*, 2002.

[6] D. Ernst et al, "Razor: A Low-power Pipeline Based on Circuit-Level Timing Speculation", *IEEE Micro*, 2003.

[7] D. Brooks et al, "Wattch: A framework for architectural-level power analysis and optimizations", *ISCA*, 2000.

[8] S. Mukhopadhyay et al., "Modeling of Failure Probability and Statistical Design of SRAM Array for Yield Enhancement in Nanoscaled CMOS", *T-CAD*, 2005.

[9] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches", *Micro*, 2006.

[10] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite", *WWC*, 2001.

[11] A. Settle et al., "A dynamically reconfigurable cache for multithreaded processors", *JEC*, Vol. 2, pp. 221-223, 2006.

[12] P. Bentley, "Evolutionary Design by Computers", *Morgan Kaufmann*, 1999.

[13] "Digital open source hardware,", [Online], http://opencores.org/.

[14] N. Binkert et al., The M5 simulator: Modeling networked systems", *IEEE Micro*, vol. 26, no. 4, pp. 52 -60, 2006.

[15] S. Paul and S. Bhunia, "Dynamic Transfer of Computation to Processor Cache for Yield and Reliability Improvement", *IEEE TVLSI*, 2011.

[16] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture", *IEEE ISCA*, 2003.