Proactive Thermal Management Using Memory Based Computing

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

Abstract*—Nanoscale devices provide the capability of gigascale integration in modern electronic systems. However, such systems suffer from high defect rates and large parametric variations. The surge of transistor count with the increased clock rate elevates the processor temperature which makes these systems even more unreliable and unstable. Dynamic Thermal Management (DTM) approaches considerably increase application's run-time in order to lower the peak temperature. 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. In this paper, we present a novel DTM technique using proactive MBC to reduce the peak temperature of applications. We propose an efficient technique to proactively transfer the instructions with frequent operand pairs to memory. Experimental results demonstrate that the proposed proactive thermal management can significantly decrease the peak temperature to improve the system reliability with minor impact on performance.

I. Introduction

Scaling down the transistor dimensions enables to integrate more and more transistors in a single System-on-Chip (SoC). Technology scaling also introduces major challenges such as high defect rate and device parameter variations [1]. Increasing process-induced variations and high defect rate in nanometer regime leads to reduced yield [3]. Operating in higher temperature due to higher power consumption of these chips makes these systems even more vulnerable to unreliability caused by parametric variations.

Dynamic Thermal Management (DTM) techniques have been widely studied and employed to control the temperature for computing platforms. Memory-based computing (MBC) is a promising alternative to improve system reliability in the presence of both manufacturing defects and parametric (process or thermal-induced) failures [2]. Existing approaches [2][15] address reliability problems due to thermal variations by dynamically transferring activities of a functional unit (FU) to memory when the FU experiences high temperature. The basic idea is to store the results of Boolean functions in lookup table (LUT) and use caches to implement the functionality of different execution units. As a result, reconfigured caches can be used as a private or shared reconfigurable computing resource for on-demand computing.

Fig. 1 depicts how MBC can be used to alleviate thermal violations. The solid line represents the transient temperature of ALU in a traditional system thoughout the execution of an application. This line is depicted in red where the temperature crosses the threshold temperature. A system is considered

This work was partially supported by NSF grants CNS-0746261 and CCF-1218629.

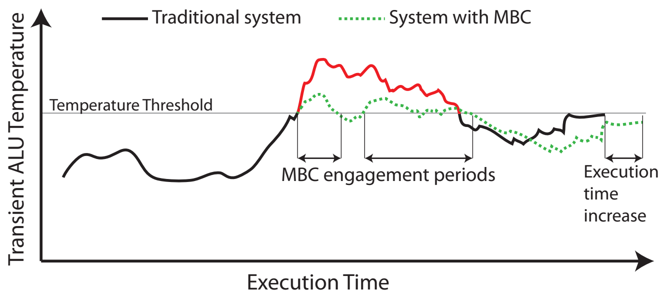


Fig. 1: Using MBC to prevent thermal violations.

reliable when the temperature remains below the threshold. In an MBC-enabled system (dotted line in the picture) instructions supported by MBC are transferred to the MBC unit after the temperature violation is triggered (reactive). Since the MBC activation is reactive to thermal violation, the ALU temperature actually crosses the threshold by a few degrees, due to the response delay, before it starts to cool down. In order to alleviate this problem MBC can be used proactively in which specific instructions can be sent to MBC to reduce activities of a functional unit.

There are two major challenges in implementing the proactive MBC: i) when to start the transfer of computations, and ii) what percentage of computations needs to be transferred to memory? If the computation transfer starts too early and/or too many instructions are transferred to memory, it can lead to unacceptable performance overhead. If the transfer starts too late and/or less than required number of instructions are transferred, the temperature may cross the threshold. Fig. 2 shows a system in which all applicable operations are sent to MBC. It can be observed that the peak temperature is reduced drastically (up to 16° Celsius). However, the execution time of this application is increased by 34%. This performance overhead may not be acceptable in many systems. In this paper, we propose an efficient proactive MBC that significantly reduces the peak temperature of a running application with minimal performance overhead. We devise an efficient method to selectively send operations to MBC that have the lowest MBC latency by exploiting the locality of most frequently used operand pairs. Our methodology improves system reliability by considerably reducing the peak temperature with minor impact on overall performance.

The rest of the paper is organized as follows. Section II describes related research activities. Section III provides an overview of memory based computation. Section IV describes our proposed dynamic thermal management methodology. Section V presents our experiments. Finally, Section VI concludes the paper.

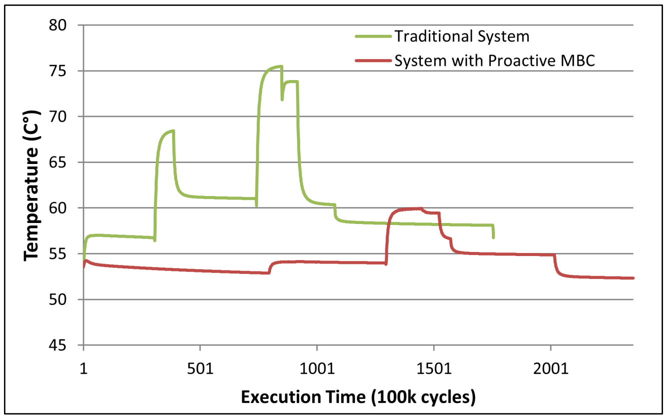


Fig. 2: Utilizing proactive MBC to prevent thermal violations using *bitcount* benchmark.

II. Related Work

Constraints such as power, energy, reliability, and temperature are among recent challenges today's microprocessor design is facing. Among these challenges, temperature-related issues have become especially important within the past several years. Temperature monitoring, thermal reliability/security, floor planning, microarchitectural techniques, and OS/compiler techniques are among the different approaches dealing with various aspects of thermal-aware microprocessor designs. We focus on microarchitectural techniques that involve Dynamic Thermal Management (DTM). These methods monitor temperature and throttle down the processors activity and hence power dissipation to protect against unexpected or malicious behaviors that exceed the capacity of cooling solution. DTM may engage during runtime of an application, and performance optimization becomes important to avoid the inevitable performance loss caused by DTM.

Brooks and Martonosi [6] evaluated the performance impact of many DTM techniques for high-performance microprocessors. They proposed DTM triggering, response, and initiation mechanisms focusing on reducing performance loss. When the temperature of the microprocessor reaches the predefined trigger temperature, there is an initiation delay before triggering DTM. After the DTM response is engaged, the microprocessor checks the temperature at each time interval. When the sensed temperature drops below the DTM trigger temperature, the DTM is disengaged and the microprocessor runs normally again. Their proposed DTM response mechanisms can be categorized into voltage/frequency scaling and throttling the instruction bandwidth of the microprocessor. They showed that ILP throttling has a much lower invocation overhead than DVFS invocation overhead. Jung and Pedram [7] proposed a stochastic dynamic thermal management technique which takes into account the stochastic nature of temperature variation. This technique utilizes DVFS for thermal management. Cochran and Reda [9] utilized processor performance counter readings to detect the phase changes of application at run time and adjust the operating frequency accordingly to avoid thermal violations. Jayaseelan and Mitra [8] proposed to dynamically adapt some micro-architecture parameters, such as instruction window size, issue width, and fetch gating level, to the application characteristics and hence control the processor temperature. To the best of our knowledge, our study is the first attempt to perform dynamic thermal management using proactive MBC.

III. Background: Memory Based Computing

Fig. 3 shows an overview of the memory based computing scheme [5]. If one or more functional units are defective, the operands for the faulty functional unit is used to form the effective physical address for accessing the LUTs corresponding to the mapped function. These LUTs are efficiently stored in the memory hierarchy.

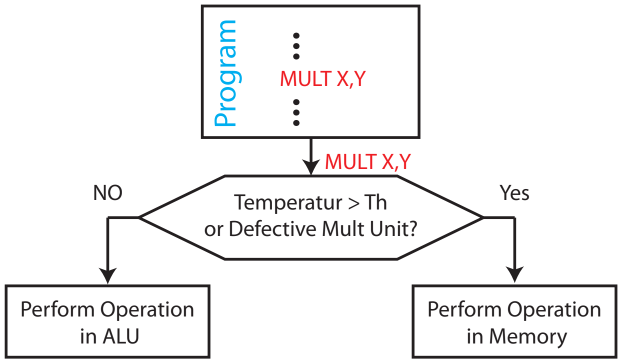


Fig. 3: An overview of memory-based computing

Fig. 4 shows 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 our earlier work [2], 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 [4].

In the MBC 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 [13]. For example, in Fig. 5, five 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. 5 is 5.

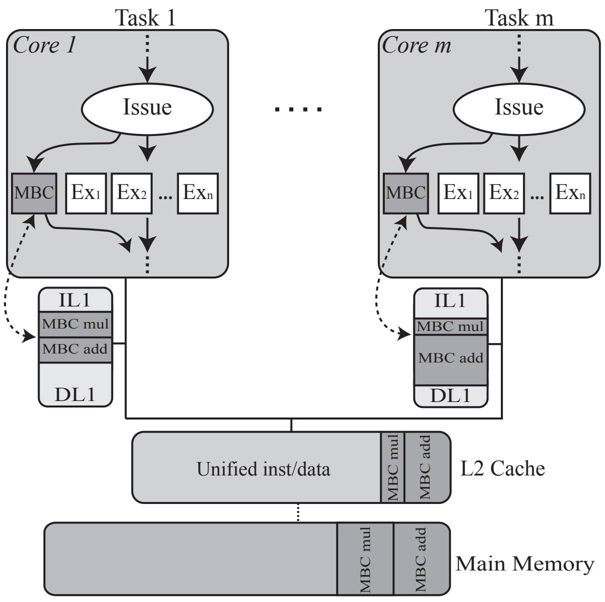


Fig. 4: Memory-based computing in multicore systems

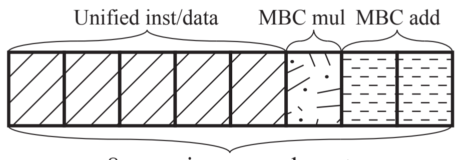


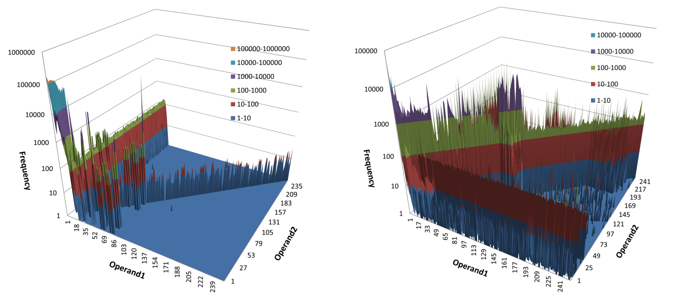
Fig. 5: Way-based cache partitioning example: 5 ways for inst/data, 1-way of MBC mul, and 2 ways for MBC add.

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. 4 *core1* uses half of private MBC cache for each MBC operation 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.

Existing MBC (we call it *reactive MBC*) is beneficial for reliability and performance improvement. It may be used for lowering the peak temperature. However, it has two disadvantages. It may violate the threshold temperature due to response delay. In addition, when the threshold temperature is reached, in a desperate attempt, it transfers all instructions to MBC regardless of their latency. This may cause significant performance overhead as some of the LUT accesses may not be present in the cache hierarchy and result in long latency memory accesses. Therefore, existing MBC is not effective in balancing both reliability and performance.

IV. Proactive MBC For Thermal Management

In this section, we propose a set of smart select functions to reduce the peak temperature of applications with minimal performance overhead. To alleviate the reliability and performance issues associated with reactive MBC, we need to transfer instructions to MBC well before the temperature threshold is reached. Clearly, sending all instructions with any operand pair values to MBC may have significant performance overhead. From Fig. 2, we can observe that the execution time of *bitcount* application is increased drastically when all addition/multiplication operations are sent to MBC. This is due to the fact that not all MBC accesses perform a one cycle LUT access and MBC accesses may take up to 7 cycles[1]. We call this *Naive Proactive MBC*.



(a) bitcount (b) parser

Fig. 6: Operand pair frequency profile of (a) *bitcount* and (b) *parser* benchmarks.

In order to identify and transfer only instructions with low-latency MBC accesses, we explored the operands value distribution patterns. We profiled the frequency of each possible pair of operands for *addition* operation (dynamic instruction count). Fig. 6 shows frequency of operands of all dynamic *addition* instructions. It can be observed that operand distribution has a very high spatial locality in applications. For example, for *bitcount* benchmark (Fig. 6(a)), most of the MBC accesses for this benchmark have operand1 between 12 and 20. Also the diagonal line where operand1 equals operand2 is among the frequent MBC accesses. In the *parser*'s case (Fig. 6(b)), the diagonal line along with operand2 equals 41 will give most frequent accesses. Although, the operand distribution is not quite clean as *bitcount* benchmark, we can capture most of the accesses if we choose operand pairs on the diagonal along with the line with operand2 as 41.

In order to exploit the operand patterns we devise an efficient method to only transfer instructions to MBC that have low latency, i.e. the results of most frequent operand pairs are stored in the MBC cache. We create an application-based smart select function that selects instructions when their operands are within the most frequent region. Finding such functions is challenging. First, the function should be very simple as it is implemented in hardware and should be very fast. Secondly, this simple function should identify the most frequent operand pair region with the lowest possible error. Third, this function should be able to work using the predefined cache size. We call this function *Decision* function as the output of this function

[1]It is infeasible to build LUTs for 32 bit operands with $2^{32} \times 2^{32}$ entries. Therefore, a 32-bit operation is essentially performed using mulitple result lookups involving 8-bit operands [15].

determines whether to send an instruction to MBC or not. Fig. 7 shows the overview of our proactive MBC approach. The basic idea is to preload the result of the most frequent operand pairs in the MBC cache and send instructions to MBC only if the operands are within the most frequent region. First, the issue unit detects whether the operation is supported by MBC. If yes, the decision function circuitry checks whether the operands satisfy the decision conditions. If the operands satisfy the decision function, the operation will be transferred to MBC.

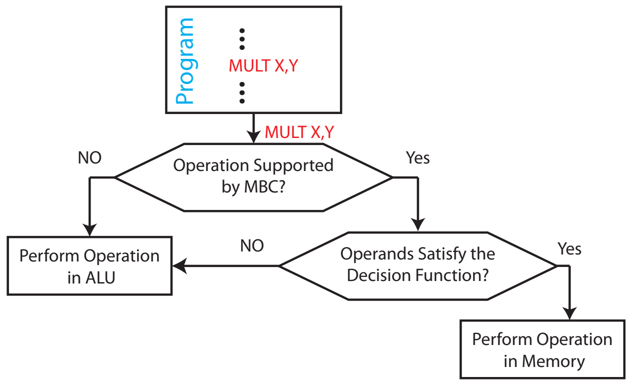

Program
MULT X,Y
MULT X,Y
Operation Supported by MBC?
NO
Yes
Perform Operation in ALU
Operands Satisfy the Decision Function?
NO
Yes
Perform Operation in Memory

Fig. 7: Proactive memory-based computing

The decision function is defined as a binary function:

$$D(i,j) = \begin{cases} 1, & \text{if } a <= i <= b \text{ and } c <= j <= d. \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

where i and j refer to operand1 and operand2, respectively; and $0 <= \{a, b, c,\ and\ d\} <= 255 \in N$ are defined as bounds. This represents general select functions in the form of:

$$\begin{cases} a <= operand1 <= b. \\ c <= operand2 <= d. \end{cases}$$

that cat be fitted to meet the needs of each application.

There are a large number of possible choices for a, b, c, and d that makes it difficult to choose a suitable decision function for an application. We use static profiling in order to find the best fit decision function for each application. We define the benefit of a decision function as:

$$Benefit(D) = \frac{\sum_{0 \le i,j \le 255} D(i,j) * F(i,j)}{\sum_{0 \le i,j \le 255} F(i,j)} \quad (2)$$

where $F(i,j)$ is the number of dynamic instructions (frequency) of the specific operation type being profiled, respectively. The benefit is the summation of frequency of instructions selected by the decision function divided by frequency of all functions. We want to include as many operand pairs in dynamic instructions as we can. We increase the boundaries to include more operand pairs and therefore increase the benefit. However, stretching the boundaries increases the minimum MBC cache size required to store the result of the most frequent operand pairs. We add $i = j$ condition to the decision function to include the diagonal line if necessary for a specific application. We also explored similar simple functions. TABLE I shows the benefit of various functions using *lucas* benchmark.

TABLE I: Benefit of using various functions with their required cache size using *lucas* benchmark.

Function	**Benefit**	**Min. memory requirement**
$0 \le i < 13$ and $7 < j < 11$	0.02	1KB
$i\ mod\ 2 = 0$ and $j = 17$	0.52	2KB
$i = 1$ or $(i\ mod\ 2 = 0$ and $j = 20)$	0.78	3KB
$0 \le i \le 30$ and $0 \le j \le 30$	0.88	1KB
$0 \le i < 60$	0.91	15KB
$i = j$ or $(0 \le i \le 100$ and $0 \le j \le 37)$	0.95	4KB

Fig. 8 shows the benefit of the function:

$$D(i,j) = \begin{cases} 1, & \text{if } 0 <= i <= 20 \text{ and } 0 <= j <= 100. \\ 0, & \text{otherwise.} \end{cases}$$

for various benchmarks for *addition* and *multiplication* operations. Although this function only requires 2KB of MBC cache, it is very beneficial for some benchmarks. For example, it gains benefit of 0.92 for *vpr* benchmark capturing 92% of all instructions. It can be observed that a decision function that is beneficial for a benchmark may perform poorly for other benchmarks. For example, this decision function only achieves 0.16 in benefits for *lucas* benchmark.

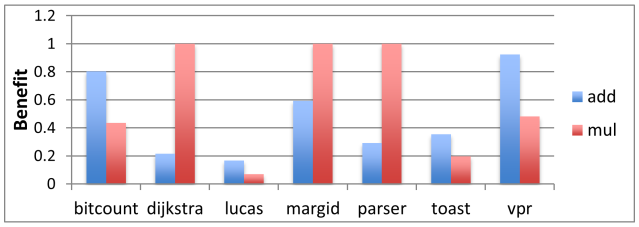


Fig. 8: Benefit of a decision function for various benchmarks.

In order to maintain the original performance of applications, we explore MBC cache sizes of 1KB, 2KB, 3KB, and 4KB. Static profiling is used to find the best fit decision function for each application. We have modified the genetic algorithm proposed in our earlier work [2] to generate best possible cache parameters when the L1 MBC cache size is limited to 1KB, 2KB, 3KB, and 4KB. The efficient L1 data/instruction cache sizes and L2 partitioning factors are computed by the proposed genetic algorithm. The overview of the genetic algorithm is shown in Fig. 9. 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 a generation and is repeated until the termination criteria is satisfied.

V. Experiments

A. Experimental Setup

To evaluate the effectiveness of the proposed approach, we incorporated tools broadly used by research community

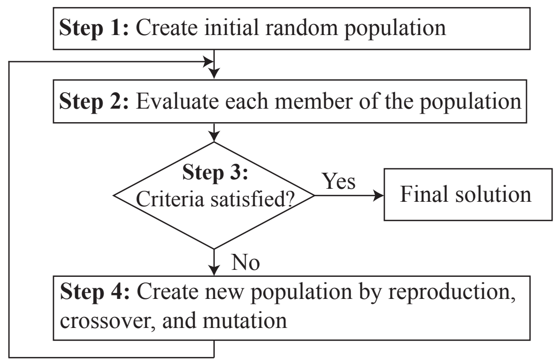

Step 1: Create initial random population
Step 2: Evaluate each member of the population
Step 3: Criteria satisfied?
Yes
Final solution
No
Step 4: Create new population by reproduction, crossover, and mutation

Fig. 9: Overview of our genetic exploration algorithm

including M5 multicore Simulator [14], HotSpot [17], and McPAT [16]. Fig. 10 shows our experimental framework. We integrated these tools at the source code level to generate one executable application that efficiently encompasses all of them. Each of these tools have a large initialization time and externally invoking them at each iteration (thousands of iterations for simulation of each application) would require extremely long simulation time. The integrated implementation was able to reduce simulation time drastically (e.g. from 15 hours to 12 minutes). The M5 simulator takes an application program along with system configuration information and produces processor as well as cache/memory architectural performance statistics. We feed these statistics to McPAT, an integrated power, area, and timing modeling framework for multicore architectures, to produce detailed power dissipation of each unit in the system. Since McPAT uses an XML as its input interface, we implemented a parser program to translate the M5 generated statistics to McPAT XML format. The power profile is then fed into "HotSpot 2.0" tool [17] in order to estimate the temperature of the integer ALU units. We used the Alpha 21264 floor plan and configurations for HotSpot, M5, and McPAT. The temperature is calculated at regular intervals during simulation of each application (once per 50,000 cpu cycles) in M5 to generate the ALU temperature trace.

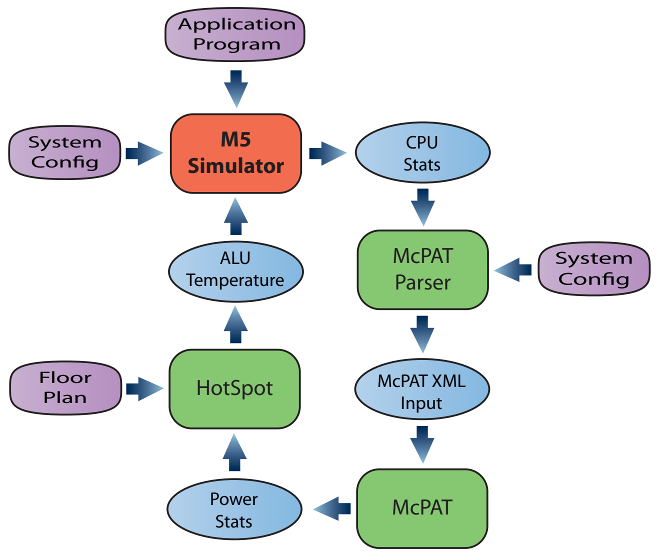


Fig. 10: Experimental framework

We implemented the computation transfer mechanism in 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 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 [4], 1KB L1 cache can only be direct-mapped as three of the banks are shut down. Similarly, 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 [4]. The memory size is set to 256MB. The L1 cache, L2 cache and memory access latency are set to 2ns, 20ns and 200ns, respectively.

TABLE II: Multi-task benchmark sets.

Set 1	mgrid,lucas	**Set 4**	parser,toast
Set 2	vpr,qsort	**Set 5**	bitcount,swim
Set 3	toast,dijkstra	**Set 6**	toast,mgrid

We used benchmarks selected from MiBench [12] (*bitcount, CRC32, dijkstra, qsort,* and *toast*) and SPEC CPU 2000 [10] (*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 [11]. TABLE II 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.

B. Results

Fig. 11 shows the transient temperature of *dijkstra* benchmark using different approaches. *No MBC* represents a traditional system without MBC. *Naive Proactive* transfers all applicable instructions to MBC. *Proactive 1K*, *Proactive 2K*, *Proactive 3K*, and *Proactive 4K* selectively transfer operations to memory where the MBC cache sizes are limited to 1K, 2K, 3K, and 4K, respectively. Running *dikstra* benchmark reaches a high peak temperature of 63.7° (Celsius) in a traditional system. Although using *Naive MBC* reduces the peak temperature by 9.4 degrees, it increases the execution time by 38%. *Proactive 4K* is able to achieve peak temperature reduction of 7.4 degrees and reduces performance overhead to 19%. *Proactive 1K* only poses a 10% performance overhead

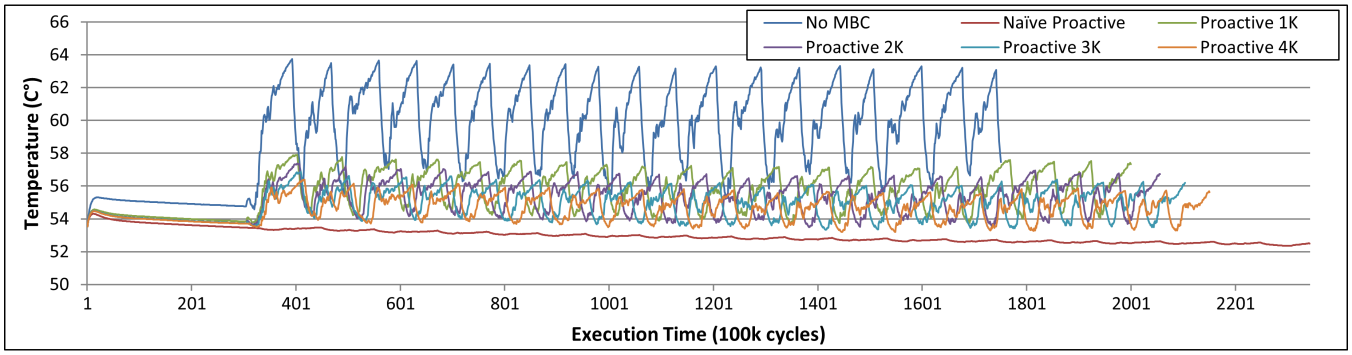


Fig. 11: Transient temperature of *dijkstra* benchmark using No MBC, Naive Proactive, and Proactive with various cache sizes

while reduces the peak temperature by 5.8 degrees. *Proactive 2K* and *Proactive 3K* achieve 6.4 and 6.9 degrees in peak temperature reduction with 13% and 16% performance overhead. As expected, transferring more operations (with larger cache sizes) reduces temperature but increases execution time. So the choice of different cache sizes creates a tradeoff between performance overhead and the peak temperature.

TABLE III shows the peak temperature and execution time of various applications using different approaches. For comparison purposes execution times are normalized to *No MBC* (the execution time is divided by the execution time of *No MBC*). On average, 8.6 degrees reduction in peak temperature (up to 19.8 degrees using *swim* benchmark) was achieved using *Naive MBC* with an average 25% performance overhead. *Proactive 1K*, *Proactive 2K*, *Proactive 3K*, and *Proactive 4K* reduce the peak temperature by 2.7, 3.4, 3.6, and 3.8 degrees on average with performance overhead of 4%, 5%, 6%, and 9%, respectively. Proactive MBC reduces the peak temperature by up to 13.9 degrees using *mgrid* benchmark with only 6% increase in execution time.

VI. Conclusion

We presented a novel thermal management technique using efficient proactive memory-based computing to reduce the peak temperature of applications. We used MBC to temporarily bypass the activity in functional units under thermal stress, thus providing dynamic thermal management by activity migration. The basic idea is to preload MBC LUT caches with the results of most frequent operand pairs in order to reduce the latency of MBC accesses. Experimental results demonstrated that the proposed proactive thermal management can decrease the peak temperature by up to 19.8 degrees (8.6 degrees on average) with nominal performance overhead.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation", *IEEE Micro*, 2005.
[2] H. Hajimiri. et al, "Dynamic Cache Tuning for Efficient Memory Based Computing in Multicore Architectures", *International Conference on VLSI Design*,January 2013.
[3] A. Agarwal et al, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies", *IEEE Trans. on VLSI*, 13,27-38, 2005.
[4] W. Wang et al., "Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multicore Systems", *DAC*, 2010.
[5] H. Hajimiri et al, " Reliability Improvement in Multicore Architectures Through Computing in Embedded Memory", *MWSCAS*, 2011.
[6] D. Brooks , And M. Aetonosi, " Dynamic thermal management for high-performance microprocessors", *International Symposium on High-Performance Computer Architecture (HPCA01)*, 2001.
[7] H. Jung and M. Pedram, " Stochastic dynamic thermal management: A Markovian decision-based approach", *In Proceedings of the IEEE International Conference on Computer Design (ICCD06)*, 2006.
[8] R. Jayaseelan, T. Mitra, " Dynamic Thermal Management via Architectural Adapting", *In Proc. of the esign Automation Conference,*, 2009.
[9] R. Cochran and S. Reda, , " Consistent Runtime Thermal Prediction and Control Through Workload Phase Detection", *In Proc. of the esign Automation Conference*, 2010.
[10] Spec 2000 benchmarks [Online], *http://www.spec.org/cpu/*.
[11] A. KleinOsowski and D. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research", *CAL g(1)*, 2002.
[12] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite", *WWC*, 2001.
[13] A. Settle et al., "A dynamically reconfigurable cache for multithreaded processors", *JEC*, Vol. 2, pp. 221-223, 2006.
[14] N. Binkert et al., "The M5 simulator: Modeling networked systems", *IEEE/ACM International Symposium on Microarchitecture*, 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] S. Li et al., "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures", *IEEE/ACM International Symposium on Microarchitecture*,2009.
[17] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture", *IEEE ISCA*, 2003.

TABLE III: Peak temperature ($^{\circ}C$) using proactive MBC.

Function	No MBC		Naive Proactive		Proactive 1K		Proactive 2K		Proactive 3K		Proactive 4K	
	Peak Temp.	Time	Peak Temp.	Time	Peak Temp.	Time	Peak Temp.	Time	Peak Temp.	Time	Peak Temp.	Time
parser	57.33	1	54.39	1.21	56.04	1.04	55.39	1.05	55.34	1.06	55.29	1.07
toast	60.29	1	55.09	1.21	57.85	1.05	57.83	1.05	57.70	1.10	57.53	1.22
mgrid	70.86	1	54.30	1.22	62.74	1.03	56.94	1.06	59.17	1.08	58.49	1.18
lucas	57.54	1	54.34	1.22	57.52	1.07	55.57	1.07	55.42	1.07	55.42	1.08
vpr	55.36	1	54.29	1.15	55.36	1.01	55.36	1.02	55.37	1.02	55.35	1.04
qsort	58.44	1	54.69	1.15	56.32	1.02	56.21	1.02	56.13	1.02	56.05	1.03
bitcount	75.50	1	59.95	1.34	71.23	1.00	73.91	1.00	70.96	1.01	70.94	1.01
swim	74.19	1	54.38	1.34	73.94	1.00	73.79	1.00	73.64	1.00	73.49	1.01
dijkstra	63.75	1	54.31	1.38	57.91	1.10	57.38	1.13	56.86	1.16	56.39	1.19