Proactive Thermal Management using Memory-based Computing in Multicore Architectures

Subodha Charles, Hadi Hajimiri, Prabhat Mishra
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA

***Abstract*—Reliability is a major concern in modern electronic systems due to high defect rates and large parametric variations. A major contributor to reliability concerns is the potential thermal violations due to increasing transistor count coupled with the high clock rate in multicore System-on-Chip (SoC) designs. Dynamic thermal management is widely used to reduce the SoC temperature. Early work on using memory-based computing has shown promising results in improving SoC reliability when few functional units are defective or unreliable under process-induced or thermal variations. However, there are no prior efforts to explore the effectiveness of MBC for thermal management in multicore architectures. In this paper, we present a novel dynamic thermal management technique using proactive memory-based computing to reduce the peak temperature of applications in multicore architectures. The basic idea is to proactively transfer the profitable instructions with frequent operand pairs to memory. Experimental results demonstrate that the proposed computing in memory can significantly decrease the peak temperature to improve the SoC reliability with minor impact on performance.**

***Keywords*-Memory-based computing; Thermal management;**

I. Introduction

Increasing advances of chip manufacturing technologies have enabled the integration of more and more transistors in a single System-on-Chip (SoC). This increased complexity has led to high defect rates and device vulnerabilities due to parametric variations [1], [2]. With the increased demand for high performance computing and massively parallel workloads, SoCs consume high power and as a result have to endure high temperatures. This makes the devices more vulnerable to parametric variations. Dynamic thermal management (DTM) is one promising method to control temperature of computing platforms [3].

Memory based computing (MBC) is a widely studied solution for parametric variations as well as component defects [4]. MBC works on the principal that the functionality of execution units (EU) will be implemented by storing results of Boolean functions in lookup tables (LUT). When a certain component is defective or causes unreliability in the SoC, computations will be dynamically transferred to memory. For example, if an execution unit (ALU) is experiencing high temperature, the operations will be done in memory and can resume normal execution when temperature goes down [4], [5]. Figure 1 shows how MBC can be used to control thermal violations [6]. It compares the transient temperature of the ALU in a conventional system (solid black line with temperature exceeding the threshold marked in red) with temperature of an MBC based *reactive* architecture (dotted line). It is called reactive because the MBC engagement starts when the temperature crosses the threshold. As a result of the reactive nature, the temperature threshold is crossed for a short time before it operates in the reliable range. In addition, in a multi-core processor cooling down a hot core may become difficult as neighboring cores can get hot at the same time.

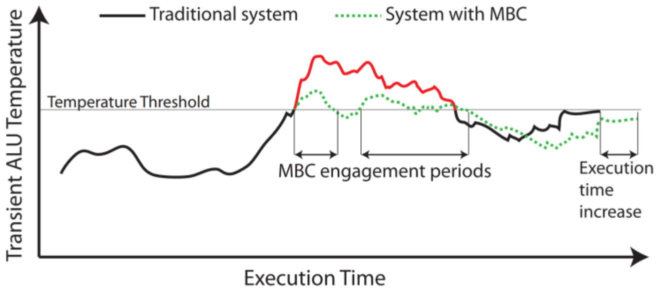

Figure 1: Example of MBC alleviating thermal violations [6]. A system is considered safe if it doesn't exceed the temperature threshold.

To address this problem, *proactive* thermal management is introduced to dynamically send specific instructions for MBC to reduce stress on EUs. There are two main challenges to be addressed in proactive MBC - (i) when to start transferring computations for MBC, (ii) what computations to transfer. If transfer starts too early or too many computations are transferred, it will result in major performance degradation. On the other hand, if the transfer is late or less computations are transferred, the thermal constraints might be violated. Figure 2 shows thermal profiles of a system running *bitcount* benchmark in a traditional setup as well as a system with proactive MBC in which all applicable computations are sent for MBC. In the MBC setup, the peak temperature gets reduced by 16°C, but the performance degrades by 34%. As a solution to the performance overheads, MBC uses existing on-chip caches to cache computation results as LUTs. It is important to note that MBC does not guarantee a complete prevention of thermal violations. MBC can be used in combination with other preventative techniques (e.g. DVFS) if such requirement is defined in the system design specification.

In this paper, we propose an efficient proactive thermal management system for a multi-processor architecture that significantly reduces peak temperature of an SoC with min-

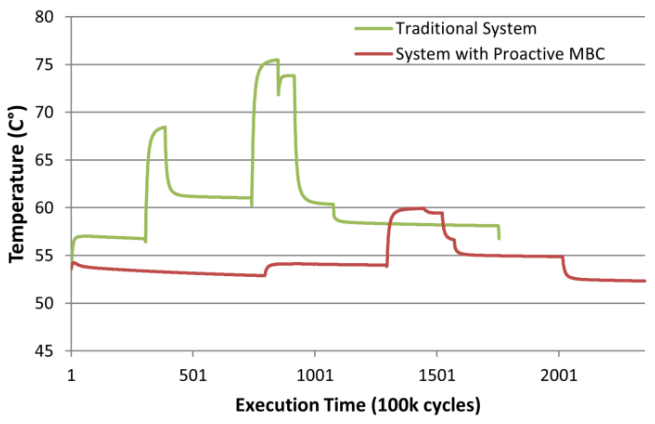

Figure 2: Proactive MBC for temperature management when running *bitcount* benchmark [6].

imal performance overhead. Proactive thermal management in a uniprocessor setup has been studied by Hajimiri et al. [6]. However, there is no existing study on proactive thermal management in a multi-core setup. Designing a dynamic thermal management solution for multi-core processors is more challenging compared to a single-core processor. In a multi-core processor, neighboring cores affect each other's performance and temperature due to shared resources and thermal conductivity. Our solution is optimized to reduce the overall processor peak temperature by focusing on the hottest core. It is important to note that it may not be possible to access the desired execution unit temperature as today's multi-core processors generally only provide a temperature sensor per core. Our approach does not rely on the exact measured EU temperature (though it may increase the precision and lead to better results) and can use approximate temperature based on available sensors.

The rest of the paper is organized as follows. Section II discusses related work. Section III provides a background on MBC based thermal management for single-core architectures. Section IV describes our proposed multi-core dynamic proactive thermal management methodology. Section V presents experimental results. Finally, Section VI concludes the paper.

II. Related Work

Among the inter-operability constraints - power [7], [8], [9], performance [10], reliability [11], temperature [12] and security [13] faced by state-of-the-art microprocessor design, temperature has become increasingly significant, especially with the introduction of high performance computing. Thermal-aware SoC design has experimented on many techniques such as floor planning, microarchitectural changes, temperature monitoring, thermal reliability/security, and OS/compiler techniques. Our focus on this work are the microarchitectural techniques that include DTM. These techniques monitor the package and component temperatures during runtime and make sure that there are no thermal violations.

However, DTM comes at the cost of performance. Brooks and Martonisi explored the impact of DTM techniques on performance and proposed several countermeasures to reduce performance loss [3]. In their work, the DTM routine is triggered when the temperature reaches a pre-defined threshold. After the DTM response engages, it periodically checks if the temperature goes below the threshold and once it does, DTM disengages and it enters normal operation. Jung and Pedram [14] introduced a stochastic dynamic thermal management technique that took the stochastic nature of temperature variations into account. By observing that different phases of an application can have different frequencies without violating its timing constraints, Cochran and Reda [15] proposed to monitor processor performance counter readings to detect these phases and to adjust frequencies accordingly to avoid thermal violations. Jayaseelan and Mitra [16] experimented DTM techniques by tuning architectural parameters such as instruction window size, fetch gating level and issue width to dynamically adapt to application requirements. Instruction-level parallelism (ILP) throttling techniques achieve linear reduction in power, while dynamic voltage and frequency scaling (DVFS) techniques are able to achieve a cubic reduction in power and hence more effective in reducing temperature [17]. However, ILP throttling can be engaged with much lower latency than changing clock frequency or voltage [18].

Existing *reactive* MBC techniques are beneficial for reliability and performance improvement. However, it has few major drawbacks. Since the DTM routine is triggered once the threshold is reached, it might violate the thermal constraints. On the other hand, once it goes above the threshold, to lower the temperature fast, it transfers all of the computations to memory. This can cause significant performance degradation as some LUT access will not be available in the cache and hence take longer to execute. Therefore, reactive MBC is not ideal in meeting both reliability and performance requirements at the same time. As a solution, the work done by Hajimiri et al. [6] applied proactive MBC to a uniprocessor architecture to improve reliability while minimizing performance overhead. Yet, their work didn't address the unique challenges in a multi-core architecture.

Coskun et al. [19] proposed an approach which predicts the future and adjusts the job allocation on a multiprocessor SoC to prevent peak temperature. They used an autoregressive moving average method to predict the future temperature. This approach faces two limitations. First, the accuracy of their approach depends on predictability of application's temperature profile which may be difficult for applications that do not present periodic predictable temperature profile. Second, it does not address the case where all cores run hot tasks at the same time. Our proposed approach addresses these concerns.

III. Background

A. Memory Based Computing

In a processor pipeline, once the instructions are decoded, the issue unit sends the instructions to their respective execution units. However, if those execution units are under thermal stress or defective, the instructions can be sent for MBC. Typically, only certain types of instructions (addition, multiplication etc.) support MBC. MBC is done based on LUTs stored in main memory and performance is enhanced using caches [5]. The operands in the instruction is used to calculate the physical address of LUTs to access for that particular instruction. Figure 3 shows an overview of MBC.

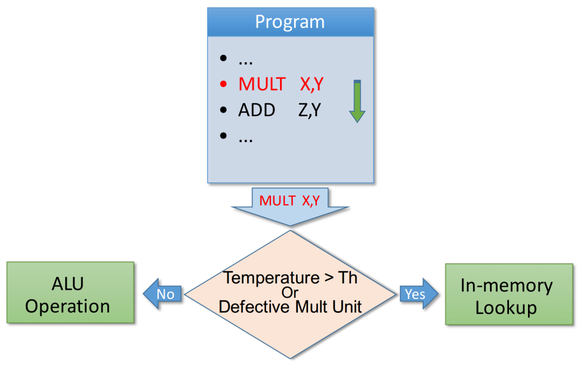

Figure 3: An overview of memory-based computing

Arithmetic operations, such as additions and multiplications, often involve large operands (e.g. two 32-bit or 64-bit operands). Storing a complete table of results for these operations using 32-bit or 64-bit operands requires large amount of memory spaces! However, such operations can be easily bit-sliced and hence efficiently represented in terms of LUTs. For example, carry-select addition of two 32-bit operands using memory based computation is shown in the Figure 4. If one of the operands is zero, the addition is completed in one cycle. If not, the 32-bit operands are bit-sliced into 8-bit operands. For each set of 8-bit operands, the addition result for both input carry zero and one is looked up from the cache. The input carry is then used to select one of the two results. The same operation is repeated for all the 8-bit operands. Thus the entire addition procedure is completed in two steps, a memory lookup and subsequent carry-select addition using the 8-bit operand addition results. Note that due to the commutative property of "add" (a + b = b + a), total memory required to store all the "add" results is halved and comes to 64KB. Considering the fact that the result for all the sub-operands $(X_i; Y_i)$ needs to reside in the on-chip memory, the worst-case evaluation time for two 32-bit operands is 4 cycles. Although this evaluation time is more than that of respective functional units, due to the fact that most of the operations (almost half of the integer operations) are narrow width [17], the average penalty in performance is not significant. The exact latency of operation depends on number of memory accesses as well as the number of cycles required to access the memory. The later is determined by the location of relevant LUT in the memory hierarchy.

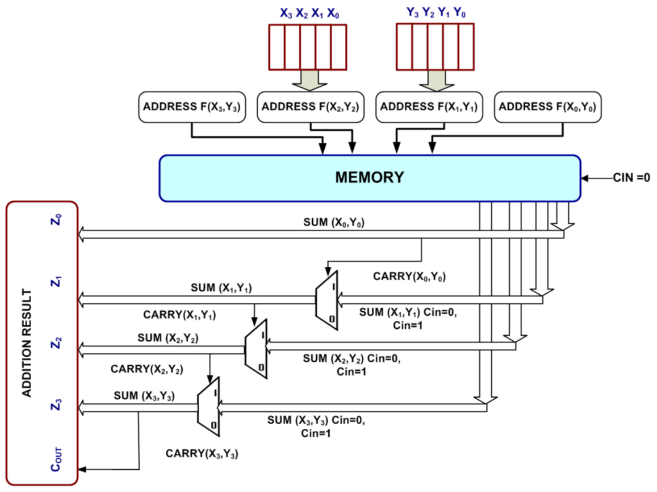

Figure 4: Implementation of memory based addition using carry-select addition [5].

B. Proactive MBC for Thermal Management

Hajimiri et al. [6] studied proactive thermal management in a uniprocessor setup. It addressed two main problems;

1) What instructions to send for MBC - if all instructions with any operand values are sent to MBC, it results in unacceptable performance overhead. This is because, LUT accesses for MBC can take upto 7 cycles [5].
2) When to send them - instructions should be transferred before the temperature threshold is exceeded. However, transferring earlier than required can incur performance penalty.

An application based *decision function* (Equation 1) was implemented to decide which instructions to send for MBC. After profiling frequency of operands for different types of instructions, it was observed that operand distribution has very high spatial locality in applications. Using this, the results of the most frequent operand pairs were stored in MBC cache which gave low latency access to LUTs. An overview of the proposed approach is shown in Figure 5. The issue unit first checks if the instruction type is supported by MBC. If yes, it is sent to the decision function to decide whether to transfer to MBC. MBC results are fetched from main memory upon the first access and will be readily available for subsequent accesses.

The decision function is given by;

$$F(i,j) = \begin{cases} 1 & \text{if } w \le i \le x \text{ and } y \le j \le z \\ 0, & \text{otherwise} \end{cases} \qquad (1)$$

where i and j refer to two operands and $0 \le \{w, x, y, z\} \le 255 \in N$ are defined as bounds which can be decided to

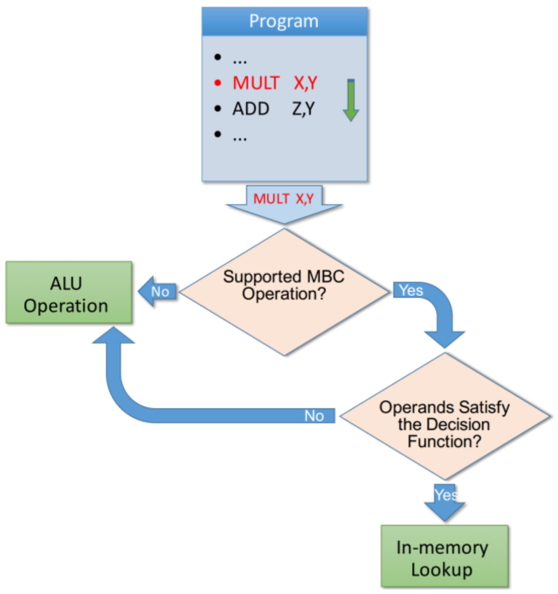

Figure 5: Proactive memory-based computing overview

Table I: Benefit values of several decision functions with their required cache size obtained for *lucas* benchmark [6].

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

fit the characteristics of each application. As the w, x, y, z variables can take many possible values, a static profiling approach with a *benefit function* (Equation 2) is defined to find the best fit decision function for each application.

$$B(F) = \frac{\sum_{0 \leq i,j \leq 255} F(i,j) \times N(i,j)}{\sum_{0 \leq i,j \leq 255} N(i,j)} \tag{2}$$

where $N(i,j)$ is the count of instructions of the instruction/computation type being profiled (add, multiply etc.). Increasing the boundaries can give more benefit, but it will consume more capacity from the cache. Table I shows benefit values of several decision functions obtained for *lucas* benchmark.

IV. Multi-Core Thermal Management

This section is organized as follows. First, we describe the architectural aspects of memory-based computing in multicore systems. Next, we present our dynamic thermal management technique using MBC in multi-core architectures.

A. Memory-based Computing in Multi-core Architectures

Figure 6 shows our multi-core architecture with MBC. It has m cores with shared L2 cache, private instruction (IL1) and data (DL1) caches [20]. L1 cache can be reconfigured by changing its capacity, linesize and associativity. To achieve cache reconfigurability without too much overhead, we use the reconfigurable cache architecture proposed in [21].

As discussed in Section III, most recent LUT accesses for MBC are cached to improve performance. MBC LUTs are cached in both L1 and L2 caches. To accommodate space for this, L1 and L2 caches are partitioned into two parts - one for caching MBC LUTs and the other to cache normal instruction/data accesses. In the example shown in Figure 6, core 1 equally divides the MBC cache space between multiplication and addition LUTs, whereas core m allocates more than half of the MBC cache space to *add* operation.

The private L1 caches, shared L2 cache as well as the private MBC caches are partitioned using way-based partitioning [22]. For example, in the cache set shown in Figure 7, five ways are dedicated for the unified instruction and data caches, two reserved for multiply LUT used for MBC and one for addition LUT. Number of ways assigned to each functionality is known as its *partition factor*. For example, the L2 partition factor for instruction/data cache in Figure 7 is 5.

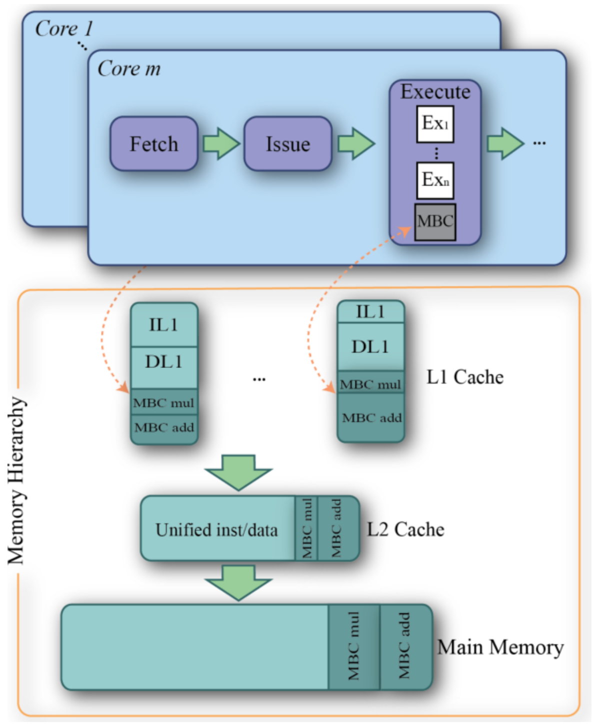

Figure 6: Memory-based computing in multicore systems

B. Proactive Dynamic Thermal Management for Multi-core

Proactive thermal management using MBC has been studied for a uniprocessor architecture in [6]. Our approach extends that dynamic thermal management approach to a multi-core framework. It is important to note that a naive

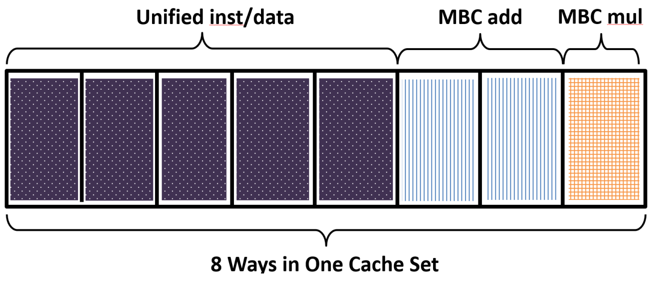

Figure 7: Way-based cache partitioning example: 5 ways for inst/data, 1-way of MBC mul, and 2 ways for MBC add.

extension of the approach proposed in [6] would not be beneficial for multi-core systems. For example, if we apply that approach for each core independently in a multi-core system, it may not be optimal when we move to multi-core framework since a hot core affect other cores and may increase the temperature in the neighboring cores. If the peak temperature at neighboring cores coincide with each other at the same time it makes the situation even worse. This can be observed based on the results shown in Figure 8. The single core solution was utilized with 1K MBC cache for both applications running on a 2-core processor. We observe that *bitcount's* peak temperature increases by nearly 4 degrees when it is executed with *swim* benchmark compared to running with *qsort* benchmark. This is due to the fact that the *swim* is also a hot task that raises the temperature of the neighboring core of the one that executes *bitcount*. In addition to thermal conductivity, MBC performance for each application running on a core is affected by applications running on other cores since the L2 MBC cache is shared among all cores. Choosing a large MBC cache size (4K) for all applications results in high L2 cache misses as all cores are competing for cache space. The prolonged L2 access latency causes delay in execution time which indeed would be good for reducing peak temperature. However, it may severely impact the performance.

A major challenge is to decide on the MBC L1 cache sizes in a way that serves both objectives:

- Reduced peak temperature
- Fastest execution time

One way to solve this problem is to dynamically adjust the MBC L1 cache sizes based on actual core temperatures at runtime. A Central MBC Optimizer (CMO) unit is added to the MBC architecture that arbitrates the MBC L1 cache sizes. The general strategy is to increase MBC L1 cache size for cores that are reaching a peak temperature and reducing MBC L1 cache for cores that are not experiencing a high temperature. Deciding based on temperature alone may not be the best approach since increasing MBC L1 size may not necessarily increase the benefit. For example, as it can be seen for benchmark *swim* in Figure 9, allocating 2KB generates near maximum benefit for this benchmark and further increasing the cache size does not increase the benefit for this application. However, the increased cache size for *swim* suggests reduction of the L1 MBC size for other cores to prevent overloading of the L2 cache, which adversely affects the MBC performance for other cores. Therefore, even if *swim* is approaching a high peak temperature allocating more than 2KB may not have major effect.

In order to optimize for both objectives (reduced peak temperature and fastest execution time), CMO uses both runtime temperature and benefits table for each application based on various MBC L1 cache sizes (a table similar to the Table I that is statically profiled and available at run-time). We formulate the multi-constraint objective function, temperature-benefit function (TB), as:

$$\begin{cases} Maximize TB = \sum_{0\le i\le \#cores} exp(cT_i)B_i(C_i) = 1 \\ \\ Subject\ to \begin{cases} C_i \in \{1KB, 2KB, 3KB, 4KB\} \\ \sum_{0\le i\le \#cores} C_i < A \end{cases} \end{cases} \tag{3}$$

where T_i is the current temperature at core i. C_i is the MBC L1 cache size chosen for core i. Note that $B_i(C_i)$ is the maximum benefit achievable for the application running on core i for the chosen cache C_i. The central MBC optimizer finds the best cache sizes for all cores by maximizing TB function at regular intervals. The constant c tweaks how sensitive the TB function is to temperature changes versus the benefit function. A determines aggressiveness of the approach. Increasing A tweaks the approach to be more aggressive (since it results in selecting larger caches and increased use of MBC, hence leads to reduced peak temperature and increased execution time). Similarly, reducing A makes the approach more conservative. CMO finds the best allocation of memory for MBC caches for each MBC operation at regular intervals according to the multi-constraint objective function. The multi-objective function defined in Equation 3 can be easily implemented in hardware. $B_i(C_i)$ values are pre-calculated offline and stored in a small table. $exp(cT_i)$ can be also an estimated value fetched from a pre-computed table for various values of T_i in the feasible range. Using a few multipliers, adders and comparators, $MaximumTB$ can be found. The number of these hardware elements can vary depending on the desired quality of the solution (how close to optimal) the CMO needs to get based on the design decision.

V. Experiments

A. Experimental Setup

To implement our architecture, we used the widely used multi-core simulator - gem5 [23]. The gem5 simulator takes an application and a set of configuration parameters

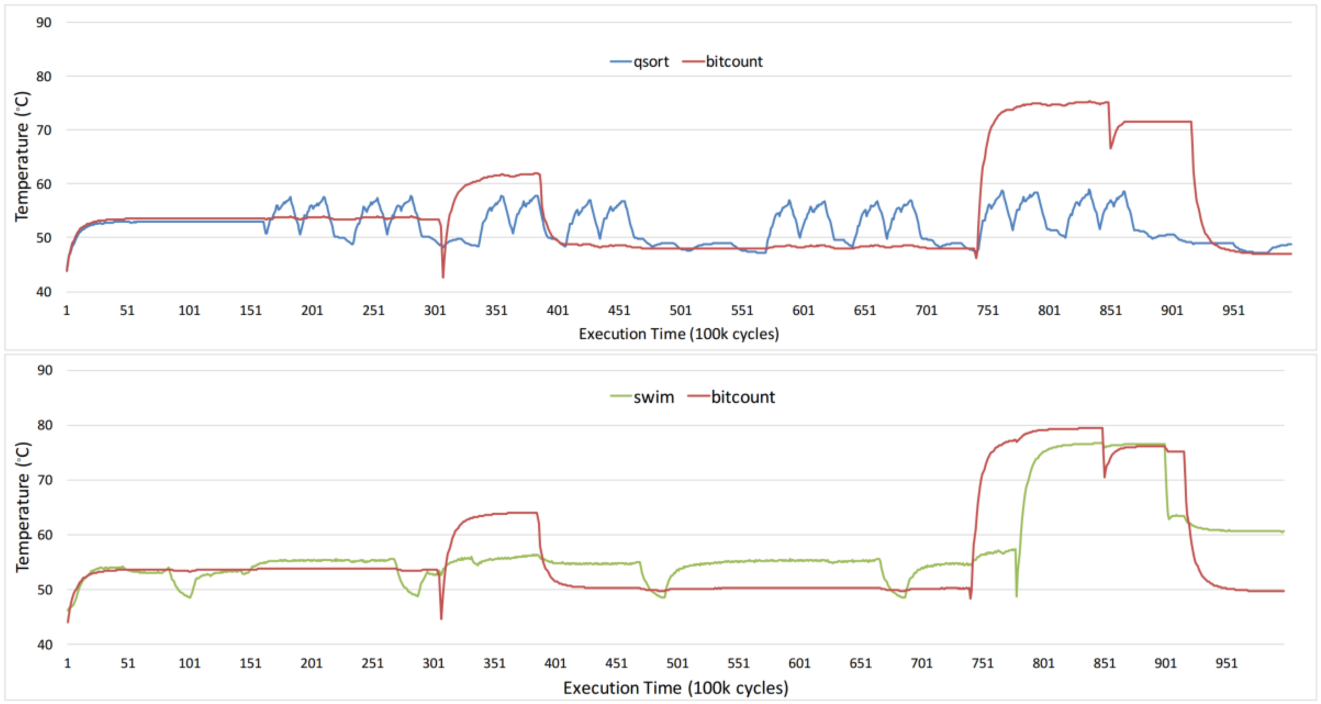

Figure 8: Transient temperature of *bitcount* benchmark running with *qsort* (top graph) and *swim* (bottom graph).

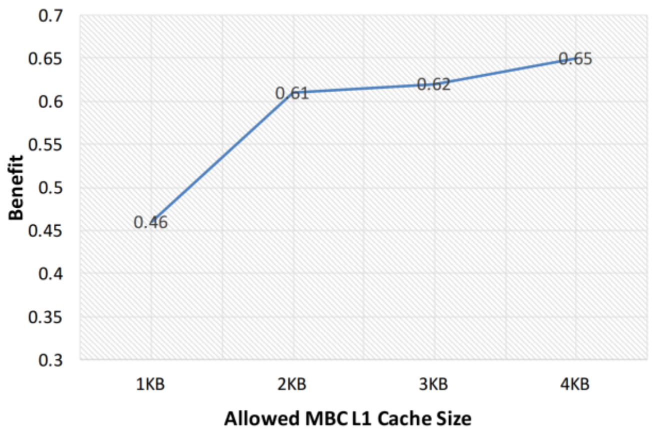

Figure 9: Achieved benefit for *swim* benchmark using various MBC L1 cache sizes.

and outputs complete architectural statistics which can be used to estimate power, performance and temperature. The architecture described in Section IV was implemented in gem5. A summary of configuration parameters are shown in Table II. For comparison, a *base cache* configuration was introduced which has 4kB capacity, 2-way set associativity and 32B line size. The base cache configuration was selected such that it meets the average requirements of the studied benchmarks [21].

The gem5 output was parsed to get the proper format and fed into the McPAT power modelling framework [24] to estimate power consumption. HotSpot 2.0 [25] takes the power profiles as input and estimates the temperature of integer ALU units. We considered multi-core processors comprised of Alpha 21264 cores placed side-by-side. Similarly, 4-core floor plan is constructed by 4 side-by-side Alpha cores. Temperature measurements were taken at every 50,000 CPU cycles using gem5 to generate the ALU temperature trace. As we are using multiple simulation frameworks sequentially, the simulations took extremely long time to finish. As a solution, we integrated all three simulators at source level to cut down the initialization and data transfer times. The source-level-integrated code drastically reduced the simulation time (15 hours reduced to 12 minutes). An overview of our experimental framework is shown in Figure 10.

Table II: System configuration parameters.

Processor Configuration	
Core frequency	500 MHz
CPU Model	DerivO3CPU (out-of-oder, SMT capable) [23]
Memory System Configuration	
DL1 and IL1 Caches	private, reconfigurable. size: 1kB, 2kB, 4kB, 8kB; associativity: 1-way, 2-way, 4-way, 8-way; line sizes ranging from 16B to 64B.
L2 Cache	reconfigurable, shared cache. 128kB capacity, 16-way associative, 32B line size
Memory capacity	256MB
L1, L2, memory access latencies	2ns, 20ns and 200ns respectively

We used 12 benchmarks selected from MiBench [26] (bitcount, CRC32, dijkstra, qsort, toast) and SPEC CPU [27] (applu, lucas, mgird, parser, swim, vpr) benchmark suites. To make the size of SPEC CPU benchmarks comparable with MiBench, reduced (but well verified) inputs sets from MinneSPEC [28] were used. In both the 2-core setup and 4-core setup, a benchmark is assigned to each core. Tasks were mapped to cores such that the total execution time of each core is comparable.

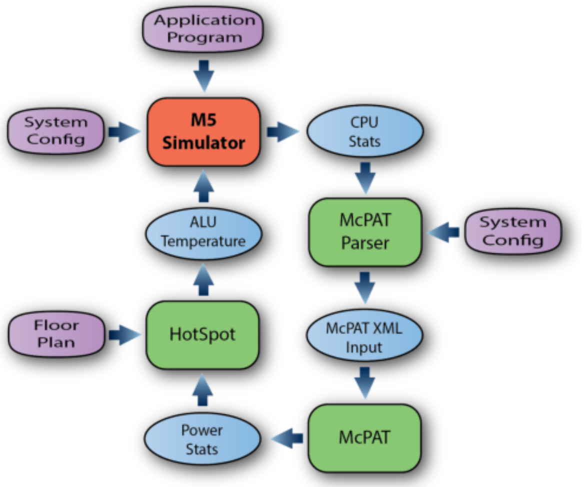

Figure 10: Overview of experimental framework

B. Results

For the multi-core scenario, we experimented various aggressiveness levels where the sum of the L1 MBC cache size is kept under certain percentage of the L2 cache (parameter A in Equation 3). Table III shows the peak temperature and execution time utilizing various aggressiveness levels for a two-core processor. In *MBC_A10*, we limit the sum of L1 MBC cache sizes, A, to a maximum of 10% of capacity of L2 cache. Similarly, A is set to 15%, 20%, and 25% for solutions *MBC_A15*, *MBC_A20*, *MBC_A25*, respectively. Notice that the peak temperatures for applications may be slightly higher in multi-core scenario compared to the single-core model. For example, the peak temperature for *swim* is 3.5 degrees higher in the multi-core setup (77.75 compared to 74.19). This is due to the fact that a hot core (*bitcount* in this case) may also increase it's neighbor's peak temperature. In the table, we have paired the results for the two tasks that were run on neighbors in parallel. The higher of the peak temperature in each task set is highlighted.

As expected, when the benchmark *toast* is paired with a hot task (*mgrid*), it's peak temperature rises by 2.67 degrees (up to 59.98 from 57.31) since MBC is not able to allocate resources to the colder task. *MBC_25* reduces the peak temperature for mgrid, when paired with toast by 11.16 degrees with only 20% increased execution time. *MBC_A10* only adds a mere 3% performance overhead while reduces the peak temperature by 6.3 degrees. *MBC_A15* and *MBC_A20* achieve 5.5 and 7.8 degrees in peak temperature reduction with 7% and 8% performance overhead for *mgrid* benchmark. Considering tasks individually, for 2-core setup, *MBC_A10*, *MBC_A15*, *MBC_A20*, and *MBC_A25* were able to reduce the peak temperature by 2.1, 3.7, 4.3, and 5.2 degrees on average with performance overhead of 4%, 5%, 6%, and 12%, respectively. It is interesting to note that the overall processor peak temperature (considering the hotter task on the two cores) is reduced by 2.8, 4.4, 5.4, and 6.7 degrees using *MBC_A10*, *MBC_A15*, *MBC_A20*, and *MBC_A25* which is more reduction compared to individual task average. This confirms that our multi-core dynamic temperature management solution is optimized for the overall peak temperature. Extending our approach to 4-core processor (Table IV), *MBC_A10*, *MBC_A15*, *MBC_A20*, and *MBC_A25* achieve reduction in peak temperature by 2.3, 2.7, 3.6, and 4.2 degrees on average with performance overhead of 4%, 5%, 6%, and 12%, respectively. The overall processor peak temperature considering all four cores is reduced by 3.7, 3.6, 5.5, and 6.0 degrees using *MBC_A10*, *MBC_A15*, *MBC_A20*, and *MBC_A25*.

VI. Conclusion

We presented a proactive MBC based dynamic thermal management system for a multi-core architecture to reduce peak temperature of applications. The basic idea is to send instructions with the most frequent operand values to be computed by MBC. MBC operations would generally be fast as the results would be readily available in MBC caches after the initial load from main memory. Our multi-core dynamic temperature management solution was able to reduce the overall ALU peak temperature on a multi-core processor by up to 11.16 degrees (6.7 degrees for 2-core and 6 degrees for 4-core processors on average) with negligible impact on performance.

Acknowledgment

This work was partially supported by the National Science Foundation (NSF) grant CNS-1526687.

References

[1] S. Borkar, “Designing reliable systems from unreliable components: the challenges of transistor variability and degradation,” *IEEE MICRO*, vol. 25, no. 6, pp. 10–16, 2005.

[2] Y. Huang and P. Mishra, “Vulnerability-aware energy optimization for reconfigurable caches in multitasking systems,” *TCAD*, 2018.

[3] D. Brooks and M. Martonosi, “Dynamic thermal management for high-performance microprocessors,” in *HPCA*, 2001, pp. 171–182.

[4] H. Hajimiri *et al.*, “Dynamic cache tuning for efficient memory based computing in multicore architectures,” in *VLSID*. IEEE, 2013, pp. 49–54.

[5] S. Paul and S. Bhunia, “Dynamic transfer of computation to processor cache for yield and reliability improvement,” *TVLSI*, vol. 19 (8), pp. 1368–1379, 2011.

Table III: Peak temperature (°C) for a two-core processor using multi-core proactive MBC

Task set	**No MBC**		**MBC_A10**		**MBC_A15**		**MBC_A20**		**MBC_A25**	
	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time
mgrid	72.75	1.00	70.97	1.03	64.71	1.06	66.83	1.08	63.64	1.20
lucas	58.49	1.00	56.89	1.07	55.91	1.07	55.35	1.07	53.71	1.08
qsort	59.22	1.00	59.02	1.02	57.13	1.02	56.01	1.02	56.00	1.02
vpr	55.84	1.00	55.63	1.01	55.52	1.02	55.38	1.02	55.38	1.06
toast	62.67	1.00	60.66	1.05	57.39	1.05	57.63	1.10	57.31	1.21
dijkstra	65.16	1.00	61.73	1.10	61.83	1.13	59.28	1.16	59.08	1.17
parser	58.95	1.00	56.37	1.03	56.12	1.04	55.79	1.05	55.42	1.06
toast	62.77	1.00	59.82	1.05	59.28	1.05	58.05	1.10	57.38	1.21
bitcount	79.14	1.00	76.75	1.00	74.39	1.01	72.47	1.01	71.88	1.01
swim	77.75	1.00	76.06	1.00	75.32	1.00	74.05	1.00	73.75	1.01
toast	61.92	1.00	61.29	1.05	60.65	1.06	60.10	1.07	59.98	1.21
mgrid	78.51	1.00	72.21	1.03	72.97	1.07	70.66	1.08	67.35	1.20

Table IV: Peak temperature (°C) for a four-core processor using multi-core proactive MBC

Task set	**No MBC**		**MBC_A10**		**MBC_A15**		**MBC_A20**		**MBC_A25**	
	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time	Peak Temp	Time
toast	62.95	1.00	60.12	1.05	59.81	1.05	58.55	1.10	58.01	1.21
lucas	58.82	1.00	56.99	1.07	55.84	1.07	55.46	1.07	54.90	1.08
vpr	57.05	1.00	56.42	1.01	56.36	1.02	56.25	1.02	56.19	1.02
parser	58.91	1.00	56.39	1.03	56.03	1.03	55.61	1.03	55.52	1.06
qsort	64.09	1.00	63.91	1.02	62.29	1.02	62.07	1.02	61.17	1.02
bitcount	81.20	1.00	76.54	1.01	77.19	1.01	74.35	1.03	73.28	1.03
swim	78.90	1.00	75.71	1.00	75.11	1.00	74.58	1.00	74.20	1.01
lucas	63.93	1.00	61.45	1.00	61.34	1.00	60.19	1.00	59.03	1.07

[6] H. Hajimiri *et al.*, "Proactive thermal management using memory based computing," in *NANOARCH*. IEEE, 2013, pp. 110–115.

[7] W. Wang and P. Mishra, "System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems," *TVLSI*, vol. 20, no. 5, pp. 902–910, 2012.

[8] W. Wang *et al.*, "Energy-aware dynamic reconfiguration algorithms for real-time multitasking systems," *Sustainable Computing: Informatics and Systems*, vol. 1, pp. 35–45, 2011.

[9] W. Wang and P. Mishra, "Dynamic reconfiguration of two-level cache hierarchy in real-time embedded systems," *Journal of Low Power Electronics*, vol. 7, no. 1, pp. 17–28, 2011.

[10] S. Charles *et al.*, "Exploration of memory and cluster modes in directory-based many-core cmps," in *NOCS*, 2018.

[11] Y. Huang and P. Mishra, "Reliability and energy-aware cache reconfiguration for embedded systems," in *ISQED*, 2016, pp. 313–318.

[12] X. Qin *et al.*, "TCEC: Temperature and energy-constrained scheduling in real-time multitasking systems," *TCAD*, vol. 31, no. 8, pp. 1159–1168, 2012.

[13] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, 2018.

[14] H. Jung and M. Pedram, "Stochastic dynamic thermal management: A markovian decision-based approach," in *ICCD*. IEEE, 2007, pp. 452–457.

[15] R. Cochran and S. Reda, "Consistent runtime thermal prediction and control through workload phase detection," in *DAC*, 2010, pp. 62–67.

[16] R. Jayaseelan and T. Mitra, "Dynamic thermal management via architectural adaptation," in *DAC*, 2009, pp. 484–489.

[17] W. Wang and P. Mishra, "PreDVS: Preemptive dynamic voltage scaling for real-time systems using approximation scheme," in *DAC*, 2010, pp. 705–710.

[18] J. Kong *et al.*, "Recent thermal management techniques for microprocessors," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 13, 2012.

[19] A. K. Coskun *et al.*, "Proactive temperature balancing for low cost thermal management in mpsocs," in *ICCAD*. IEEE Press, 2008, pp. 250–257.

[20] H. Hajimiri *et al.*, "Compression-aware dynamic cache reconfiguration for embedded systems," *Sustainable Computing: Informatics and Systems*, vol. 2, pp. 71–80, 2012.

[21] W. Wang *et al.*, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *DAC*, 2011, pp. 948–953.

[22] A. Settle *et al.*, "A dynamically reconfigurable cache for multithreaded processors," *Journal of Embedded Computing*, vol. 2, no. 2, pp. 221–233, 2006.

[23] N. Binkert *et al.*, "The gem5 simulator," *CA News*, 2011.

[24] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009, pp. 469–480.

[25] K. Skadron *et al.*, "Temperature-aware microarchitecture," in *ISCA*, 2003, pp. 2–13.

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

[27] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[28] A. KleinOsowski and D. J. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research," *IEEE Computer Architecture Letters*, vol. 1, no. 1, pp. 7–7, 2002.