Memory Based Computation Using Embedded Cache for Processor Yield and Reliability Improvement

Somnath Paul and Swarup Bhunia
Electrical Engineering and Computer Science Department, Case Western Reserve University, Cleveland, OH.
{sxp190, skb21}@case.edu

Abstract

VLSI systems in the nanometer regime suffer from high defect rates and large parametric variations that lead to yield loss as well as reduced reliability of operation. In this paper, we propose a novel memory-based computation framework that exploits on-chip memory for reliable operation by transferring activity from a defective or unreliable functional unit to the embedded memory. This allows the die to run at a reduced performance level instead of being completely discarded or being throttled (in case of variations). We show that the proposed method improves yield and reliability in a superscalar out-of-order processor by tolerating defective functional units and allowing dynamic thermal management. The simulation results show that it entails only a small loss in performance (average 1.8%) at the cost of 9.5% of area overhead required with hardware duplication.

I. INTRODUCTION

Although technology scaling provides the capability to integrate billions of transistors in a modern processor, it gives rise to important issues such as high defect rate and variability-induced reliability concerns [1]. Increasing defect rate and device parameter variations in sub-90nm technology regime leads to reduced yield [2]. Moreover, increased power density in modern high-performance microprocessors (~100W/cm^2 for 50-nm technology [3]) leads to an overall increase of the processor temperature due to the limited cooling capacity of the package. Moreover, the power density varies across the chip depending upon the functionality of the circuit block. Typically power density of a cache is much less compared to the execution unit (e.g. integer ALU). In order to adapt to both manufacturing defects and Process-Temperature-Voltage (PTV) induced parameter variations, an effective solution is to develop a system which can dynamically detect and correct defect and variation induced failures. For example, thermal management under temperature variation can be addressed using a Dynamic Voltage Frequency Scaling (DVFS) scheme [4], which can dynamically adjust operating conditions (i.e. voltage and frequency) to operate under a temperature and power envelope (thereby achieving high reliability). Such a system, however, incurs high performance loss (due to system-wide voltage/frequency scaling) and large hardware overhead. In this paper, we propose an architecture-level solution for improving processor yield and reliability of operation using memory based computation. The proposed computation framework allows on-demand transfer of activity from functional units of the processor such as ALU to the embedded memory of the processor and thus compensates for hardware defects as well as reduced reliability of operation in a functional unit under parametric variations.

The key idea is to realize the functionality of different execution units in a processor, such as adder or multiplier using on-chip memory, which acts as a reconfigurable computing resource. We demonstrate that on-chip cache in a processor can be used to

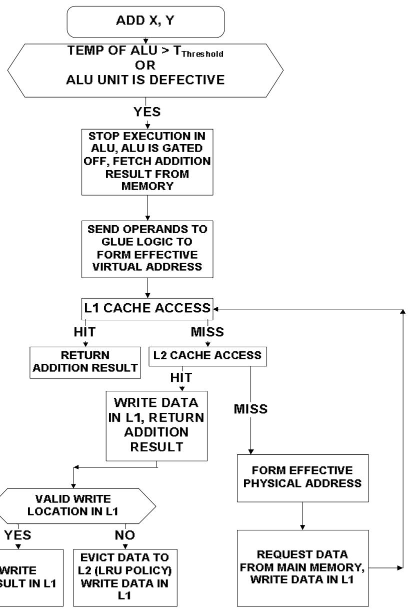
ADD X, Y
TEMP OF ALU > $T_{Threshold}$ OR ALU UNIT IS DEFECTIVE
YES
STOP EXECUTION IN ALU, ALU IS GATED OFF, FETCH ADDITION RESULT FROM MEMORY
SEND OPERANDS TO GLUE LOGIC TO FORM EFFECTIVE VIRTUAL ADDRESS
L1 CACHE ACCESS
HIT
MISS
RETURN ADDITION RESULT
L2 CACHE ACCESS
HIT
MISS
WRITE DATA IN L1, RETURN ADDITION RESULT
FORM EFFECTIVE PHYSICAL ADDRESS
VALID WRITE LOCATION IN L1
YES
NO
WRITE RESULT IN L1
EVICT DATA TO L2 (LRU POLICY) WRITE DATA IN L1
REQUEST DATA FROM MAIN MEMORY, WRITE DATA IN L1

Figure 1: Flow chart showing memory based computation framework.

perform computation on demand by storing the results of Boolean functions as a look-up table (LUT). The portion of the embedded memory dedicated to LUT implementation of the different functions can be traded off with the performance loss which is however within tolerable limits due to the high locality of reference across different clock cycles. A small hardware overhead (which we call as *glue logic*) is required for forming the effective virtual and physical addresses to access the cache and the main memory respectively.

The paper makes the following contributions:

1. *It proposes to use the on-chip memory of a processor to realize the functionalities of different execution units.*
2. *It ensures correct operation of the processor in case of Process-Temperature-Voltage induced parametric variations by transferring the activity of the functional units to the memory.*
3. *It helps in salvaging the processors with defective functional units by allowing memory based realization of their functionalities and therefore improves the yield in a "go-no-go" situation.*

II. MEMORY BASED COMPUTATION: OVERVIEW

Memory-based look up table implementation is common in case of FPGAs, which essentially consists of two-dimensional array of small LUTs and programmable switching matrix. Memory based computation has also been explored in the context of Digital Signal Processing (DSP) domain as illustrated in [7]. The idea is to circumvent the classic Von-Neumann bottleneck [7] by moving the data to the memory and doing the computation in the memory itself. The advantage of such a technique is some iterative tasks meant for running on the CPU can now be run in the memory itself, thus increasing the throughput. A similar approach has been followed in [9], where the authors have proposed a LUT based implementation of computation intensive DSP and image applications such as DCT and IDCT. The on-chip cache is used for realization of the LUT. However, the method as described in [9] entails a significant design overhead due to incorporation of computing elements and intermediate decoders in the cache memory. Although memory based computation

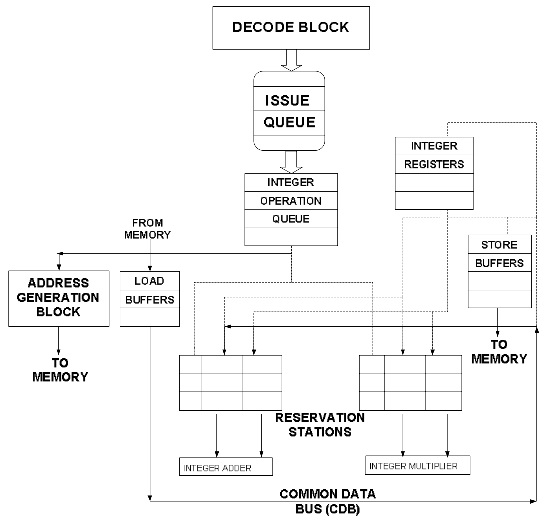

Figure 2: Integration of memory-based computation with dynamic pipeline.

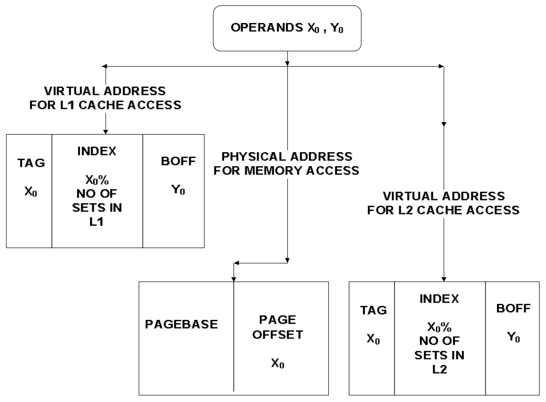
OPERANDS X_0, Y_0
VIRTUAL ADDRESS FOR L1 CACHE ACCESS
TAG X_0
INDEX X_0% NO OF SETS IN L1
BOFF Y_0
PHYSICAL ADDRESS FOR MEMORY ACCESS
VIRTUAL ADDRESS FOR L2 CACHE ACCESS
PAGEBASE
PAGE OFFSET X_0
TAG X_0
INDEX X_0% NO OF SETS IN L2
BOFF Y_0

Figure 3: Formation of virtual and physical addresses.

CACHE L1/L2				
TAG	INDEX	DATA (ADDITION RESULTS)	VALID BIT	LRU BIT

MAIN MEMORY	
PAGE OFFSET	ADDITION RESULTS

Figure 4: Arrangement of addition results in cache and main memory.

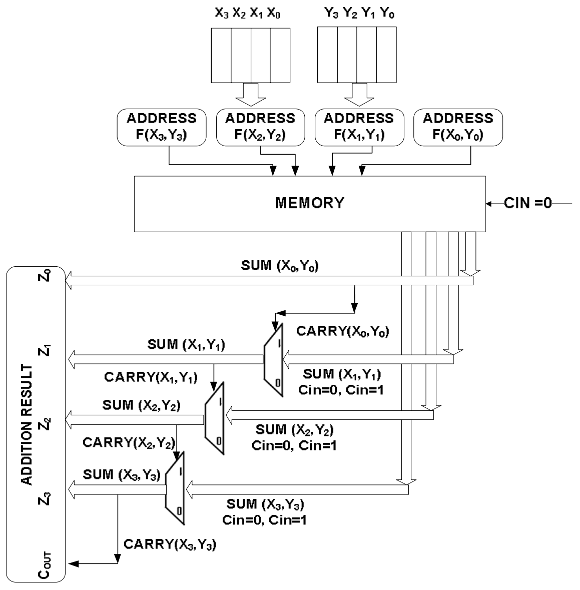

Figure 5: Implementation of memory based addition using carry-select adder.

has been explored before, the proposed computation framework has two major differences from the previous works: a) it tries to perform most common operations in a system (such as addition, multiplication etc.) and not specific iterative tasks; b) it addresses improvement in yield and reliability; c) it preserves the advantage of high device integration density of embedded caches. On-chip memory in modern processors can be utilized to realize logic function apart from being used as a stand-alone storage element for data and/or instructions. In order to explain the operation of the proposed methodology, we will try to answer the following four questions.

a) How can we perform computation in memory?

In this work, we propose implementing logic and arithmetic operations in the on-chip cache, which provides easy dynamic reconfigurability (which means the same cache memory can be used as a look-up table for add and multiplication operations in different cycles). We assume the result of an operation is stored in physical memory and fetched on-demand to the on-chip cache. Clearly, the allocation of data in the table and its intelligent processing is essential to reduce the amount of storage.

b) Which functions should we compute in memory?

Functions which have relatively small number of inputs and outputs are ideally suited for memory based computation. Arithmetic operations such as addition and multiplication often involve large operands. However, we note that such operands can be suitably bit-sliced and the memory based computation can be executed on these bit-sliced operands. This also reduces the amount of storage required for this framework.

c) When do we perform memory based computation?

A memory based computation framework will not be able to replace a logic unit in terms of its performance. However, in the scenario of a permanent defect to a functional unit or when the functional unit operates under thermal stress, the memory may be used to share a workload from the functional unit [6].

b) What modifications we need to incorporate in the memory?

The requirements for the embedded memory to support computation can be divided into two classes. These are: 1) efficient look up and data storage algorithm that integrates seamlessly with the conventional embedded memory addressing scheme 2) the functional requirement for the embedded memory to have low access latencies. Moreover, the algorithm for fetching the data from the main memory should be capable of exploiting the temporal correlation of the data for the same operation across different clock cycles.

III. ACTIVITY TRANSFER IN A PROCESSOR

The proposed memory based computation framework was applied to realize the functionality of the integer execution unit, which is extremely critical in an Out-of-Order Superscalar processor pipeline. An overview

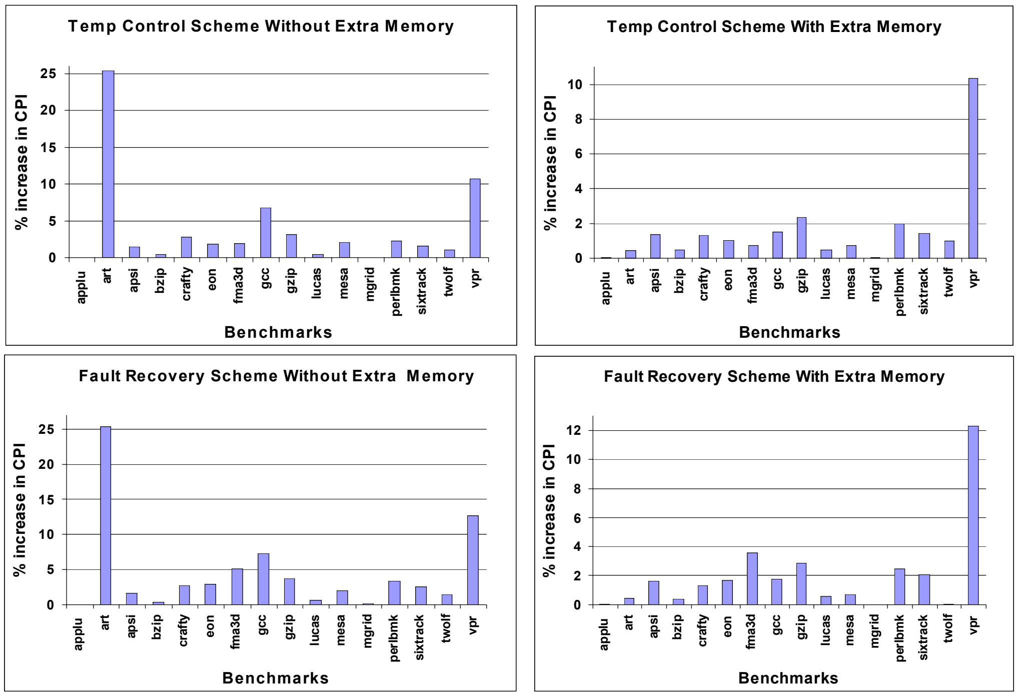

Figure 6: Performance overhead for temperature control and fault tolerance schemes.

of the proposed scheme has been shown in Fig. 1. In case of permanent failure of a functional unit or when the temperature of a particular functional unit such as an ALU adder or a multiplier is above a threshold limit (100°C), the adder or the multiplier of the ALU is bypassed followed by memory based computation of these functions. Once a need for bypassing the normal execution unit has been detected the processor will need to issue an indication for the OS to load the result tables for that particular operation in a section of the main memory. The OS returns the pagebase address where the results for the look up table realization of the operation are being stored in the main memory. This pagebase address will be used for further access to the main memory to load the result tables to the on-chip cache memory. The pagebase address is used to form the physical address required to access the main memory, the on-chip cache being accessed with a virtual address. The instruction after being issued from the issue queue is redirected to the address generation unit (shown by bold arrows in Fig.2) for calculation of effective address, which is used to access the result of the operation.

A. Formation of effective addresses

Two most frequent arithmetic integer operations, addition and multiplication were chosen for realization using memory based computation procedure. The multiplication operation was implemented using repeated additions. The following section discusses the formation of virtual and physical addresses required to retrieve the addition results from the cache or the main memory. The formation of virtual and physical addresses is shown in the Fig. 3. Let us consider the addition of two 32 bit long operands X and Y. Each of the operands can be seen as a regarded as a combination of four 1 byte operands (X_0, Y_0), (X_1, Y1), (X_2, Y_2) and (X_3, Y_3). The addition result for such 1 byte operands can be stored in the memory in the form of tables. For the first access, let us consider that the operands are X_0 and Y_0. As shown in Fig. 3, the virtual address to access the L1 cache is formed with the operand X_0 forming the tag section for the virtual address. The index for the virtual address is formed by 'X_0 *mod* $L1_{set}$' operation where $L1_{set}$ is the number of sets present in the L1 cache. The other operand is used as the block offset section for the virtual address. The proposed addressing scheme thus circumvents the synonym/aliasing problem common for virtually indexed virtually tagged addressing by having one of the operands present in the tag portion of the virtual address. The physical address required to access the main memory tables is formed with one of the operands serving as the page offset. The page base address stored previously from the OS is concatenated with the page offset to form the complete physical address that selects a line of a page in the main memory. The selected line contains the addition results of the operand in the page offset with all possible 1 byte operand. Since we are considering only addition as the fundamental operation, the addition LUTs loaded in the main memory have one page base address. This single pagebase address eliminates the overhead of having a Page Table or Translation Look-Aside Buffer (TLB). In the proposed scheme, a virtually indexed virtually tagged cache removes the need for virtual to physical address translation on each cache access. Such a translation is required only for accessing the main memory.

B. Cache and Memory Organization

The cache and the memory organization for storage of addition results are shown in Fig. 4. The page loaded in the main memory contains the addition result of all 8 bit operands and the respective carry outs. Each line of the page in the main memory contains the sum of one 8-bit operand (let X_0) with all 8 bit operands (Y_0, Y_1 ..Y_N). Two such sets are present, for input carry is zero and one. The page offset is used to access the addition results for both cases of the input carry. The input carry is then used to select the proper addition results. On a cache miss, the memory line containing the addition results of one of the operands (X_0) is brought into the cache. The second operand (say Y_0) which forms the block offset section of the address is used to access the result of addition for the two operands (X_0 and Y_0). The hierarchy of cache access is shown in the Fig. 1. The L1 cache is accessed with

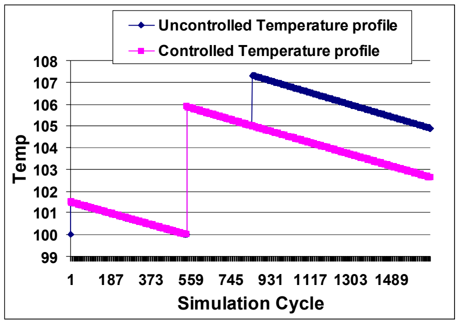

Figure 7: Temperature profile for ALU unit.

Table I: Area overhead for glue logic

	Area (μm^2)	Total Area (μm^2)
32 bit Shifter	11280	19324
32b Priority Encoder	3541	
32 bit Comparator	4503	
Area overhead for duplicated functional unit		
32 bit Adder	10135	204555
16 bit Multiplier	194420	

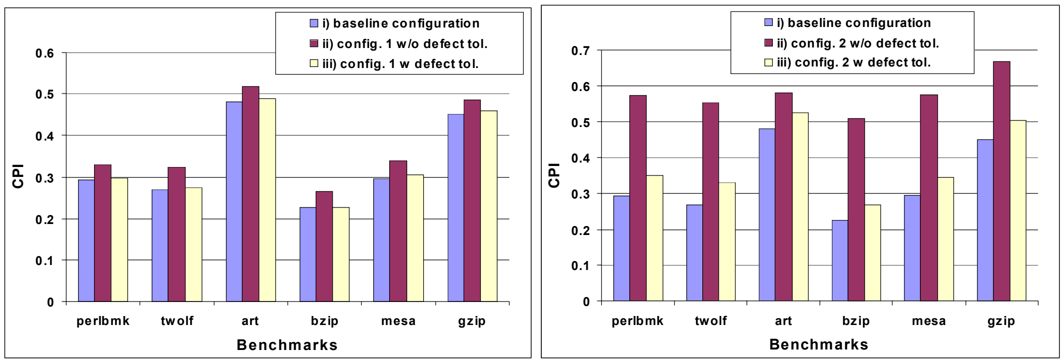

Figure 8: Performance results with and without memory based computation framework in case of a) 2 adders 1 multiplier being faulty and b) 4 adders and 1 multiplier being faulty.

the L1 virtual address; miss on the L1 cache triggers an L2 cache access with the L2 virtual address. A hit on the L2 cache causes the return of data to the reservation station or the integer register that is supposed to receive the result of addition. The data is also written into L1 cache and the valid bit is set on a successful write operation. If a free write location is not found in the L1 cache, then the data present in that location is evicted from L1 and written to L2 cache. The Least Recently Used (LRU) policy is followed as a replacement policy for L1 eviction scheme.

C. Memory based addition procedure

A scheme based on carry-select addition of two 32 bit operands using memory based computation is shown in the Fig. 5. The 32 bit operands are divided into 8 bit operands and for each set, the sum is looked up from the cache using the input carry as select signal. Thus the entire addition procedure is completed in two steps, a memory look up and subsequent carry select addition using the 8 bit operand addition results.

D. Implementation of the multiplier

The proposed scheme for memory based computation can be extended to integer multiplication as well. The fundamental operation for implementing an integer multiplier is addition and the simplest implementation for the multiplication of two integers can be realized by checking the multiplier bits for zero or one. The shifted multiplicand is added to the partial product to obtain the new value of partial product. The shifted value of the multiplicand is obtained by a combinational shifter network. The addition of the partial product and the shifted multiplicand is however realized by memory based addition operation. Some other implementations for the multiplier have also been investigated such as the Booth Radix 2 and 4 as well as using the look-up table based implementation described in [8]. However, the simplest multiplication algorithm has been found to offer comparable performance at a lower design overhead.

IV. TEST SETUP & RESULTS

i) Test Setup*:* The proposed scheme has been validated on an Out-of-Order Superscalar processor architecture using the Simplescalar Tool Set Version 3.0 [10]. The baseline configuration includes an 8-way issue processor with 6 integer ALUs and 2 integer mul/div units. The virtually addressed cache memory includes 2 way 8KB L1 cache with 1 cycle latency and a 4-way 64KB L2 cache with 6 cycle latency, both of which have LRU replacement policy. The main memory page allocated for storing the results of addition

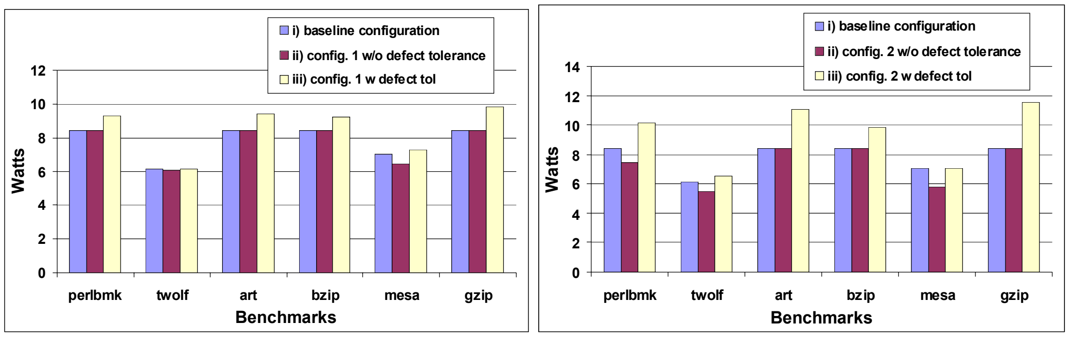

Figure 9: Power results with and without memory based computation framework in case of a) 2 adders 1 multiplier being faulty and b) 4 adders and 1 multiplier being faulty.

instruction is of size 64KB. The latency for accessing the main memory is taken to be 100 cycles for the first access and 4 cycles for inter chunk access. We have considered two different scenarios, i) when the memory based computation is used for improving the reliability of operation under temperature variation, ii) when the proposed computational framework is used to completely replace few functional units that have been rendered inoperative. Simplescalar was modified to incorporate the proposed activity transfer scheme and the above scenarios were then simulated for different Spec2000 Benchmarks [5]. For each scenario mentioned above, the performance loss was estimated when i) an extra on-chip memory is used for LUT implementation ii) when the existing caches are used.

***ii) Results**:* From the performance results presented in Fig. 6, we note that when the proposed computational framework is used for temperature management with an extra on-chip memory being used for LUT implementation, the average increase in Cycles-Per-Instruction (CPI) is only 1.58%. When existing on-chip memory was used, the performance degraded more (3.89%), since it led to more capacity misses. It is important to note that in DVFS scheme [4], the performance penalty is over 10%. Thus, the proposed scheme has significant savings in performance over DVFS for temperature management. Figure 7 shows the temperature profiles of the ALU unit for a given benchmark (*perlbmk*). As seen in Fig. 7, the uncontrolled temperature profile tends to increase more even after it has crossed the temperature threshold. However, in the proposed activity transfer scheme the ALU operation is bypassed, thus allowing the ALU temperature to decrease. We have also validated the effectiveness of the proposed scheme for improving yield assuming hard defects in the execution units. The results in Fig. 6 correspond to the case where 2 integer adders and 1 integer multiplier are considered to be defective and their activity is migrated. The performance overhead is only 2% in case of a separate flexible memory and 4.47% in case of using the existing cache for LUT implementation. Configuration 1 and 2 in Fig. 8 (a) and (b) correspond to the case when i) 2 integer adders and 1 integer multiplier are defective and ii) 4 integer adders and 1 integer multiplier are defective respectively. As observed from Fig. 8(a), the performance overhead for a faulty processor with configuration 1 is 13.27%. In case of configuration 2, the processor suffers a higher performance overhead (81.2%) due to more number of defective units. However, when the proposed scheme is incorporated in the processor architecture, the loss in performance is minimal (1.71% and 16.4% respectively). The smaller performance overhead for the proposed computation framework is due to the correlation that exists among the operands thereby reducing the number of L2 and main memory accesses. The proposed activity transfer scheme requires additional hardware which involves a 32 bit comparator for comparing the operands before memory based addition and multiplication procedures. Additional hardware required for multiplication includes a 32 bit priority encoder and a 32 bit shifter for obtaining a multiplicand of proper weight. Table I compares the hardware overhead for the proposed scheme against the complete duplication of the entire functional units. The additional hardware requirement for memory based computation is only 9.5% of the duplicated functional units. We have also noted the power overhead due to the proposed framework. The results were collected using Wattch Tool Set Version 1.0 shown in Fig. 9(a) and (b). From the results we note that the proposed scheme incurs a power overhead of 10.74% and 27.15% for configuration 1 and 2 respectively.

V. CONCLUSION

We have presented a novel memory based computation framework that will enable modern processors for on-demand transfer of computation from functional unit to memory. The principal idea is to use embedded cache of a processor as LUT based computing resource. The proposed scheme can be effectively used to improve manufacturing yield and temporarily bypass the activity in functional units under time-dependent local variations, thus providing an efficient solution to dynamic thermal management. We show that the benefits of memory-based computation come at the expense of small loss in performance and low hardware overhead.

REFERENCES

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation", *IEEE Micro*, 2005.

[2] A. Agarwal et al, "A Process-Tolerant Cache Architecture for Improved Yield in Nanoscale Technologies", *IEEE TVLSI*, 2005.

[3] C. Minsik, "TACO: temperature aware clock-tree optimization", *ICCAD* 2005, Pages: 582 – 587.

[4] R. McGowen et al, "Power and temperature control on a 90-nm Itanium family processor", *IEEE JSSC* 2006.

[5] Spec 2000 benchmarks. [online] *http://www.spec.org/cpu/*

[6] K. Asanovic et al, "Reducing power density through activity migration", Pages: 217 – 222.

[7] B. I. Pawate et al, "Memory based digital signal processing",*ICASSP-90*,pp.941-944.

[8] H. Ling, "An approach to implementing multiplication with small tables", *IEEE Computers*, 1990, pp: 717 – 718.

[9] H. Kim et al., "A reconfigurable multifunction computing cache architecture", *IEEE TVLSI*, Vol. 9(4), 2001.

[10] Simplescalar tool set. [online] *www.simplescalar.com.*