Reconfigurable STT-NV LUT-based Functional Units to Improve Performance in General-Purpose Processors

Adarsh Reddy Ashammagari[1], Hamid Mahmoodi[2], Tinoosh Mohsenin[3], Houman Homayoun[1]
[1]Dept. of Electrical & Computer Engineering, George Mason University, Fairfax, VA
[2]Dept. of Computer Engineering, San Francisco State University, SF, CA
[3]Computer Science & Electrical Engineering Dept., University of Maryland Baltimore County
E-mail: {aashamma, hhomayou}@gmu.edu, mahmoodi@sfsu.edu, tinoosh@umbc.edu

ABSTRACT
Unavailability of functional units is a major performance bottleneck in general-purpose processors (GPP). In a GPP with limited number of functional units while a functional unit may be heavily utilized at times, creating a performance bottleneck, the other functional units might be under-utilized. We propose a novel idea for adapting functional units in GPP architecture in order to overcome this challenge. For this purpose, a selected set of complex functional units that might be under-utilized such as multiplier and divider, are realized using a programmable look up table-based fabric. This allows for run-time adaptation of functional units to improving performance. The programmable look up tables are realized using magnetic tunnel junction (MTJ) based memories that dissipate near zero leakage and are CMOS compatible. We have applied this idea to a dual issue architecture. The results show that compared to a design with all CMOS functional units a performance improvement of 18%, on average is achieved for standard benchmarks. This comes with 4.1% power increase in integer benchmarks and 2.3% power decrease in floating point benchmarks, compared to a CMOS design.

Categories and Subject Descriptors
C.1.1 [PROCESSOR ARCHITECTURES], Single Data Stream Architectures: Pipeline processors Systems; C.4 [Performance of Systems]

Keywords
STT Technology, Reconfigurable Functional Units, Performance

1 INTRODUCTION
With the current shrinking trend in CMOS technology, larger processing capabilities can be incorporated within the same die footprint. At the same time, the number of functions that are now computationally realizable has also increased in leaps and bounds. Therefore, an efficient allocation of functional resources becomes crucial to the overall performance of any processing unit [3, 4, 5]. Under limited functional resources available to general-purpose processors, major performance bottlenecks arise from functional units unavailability. There are two ways to look into this problem (i) one to increase the number of functional units in a general-purpose processor (ii) transform and adapt the functional units to serve different function needs. The first solution however is not design efficient as will be discussed in Section 2. The next alternative that we have addressed in this paper is adaptability and reconfigurability between functional units. Incorporating adaptable functional units results in better utilization of hardware, which leads to performance improvement. Reconfiguring a unit to multiple functions requires an on-chip programmable fabric. This reconfiguration is performed on a Spin Transfer Torque Random Access Memory based look-up table (STT-NV-LUT) that is a composed of Magnetic Tunnel Junctions (MTJs). The advantages of using STT-NV technology are its zero standby power and thermally robust behavior. Recently use of MTJs has been explored for realizing low power programmable Look Up Tables (LUT) in processor and Field Programmable Gate Arrays (FPGAs) [12, 14, 16]. MTJs have been mainly used to design low power and thermally robust logics [12, 16]. In latest work MTJs has been used to reduce power and temperature in processor architecture [12, 17]. MTJs therefore have computing ability in addition to non-volatile storage property [12, 14, 17]. MTJ based clocking and logic architecture have already been developed in integration to CMOS [16]. In this paper, we utilize the STT-NV based look up tables [12, 16], to build on-chip adaptable functional units. Such look up tables show very little leakage power. Mapping a function to look up tables generally results in lower performance as compared to the custom implementation using standard cell logic gates; however, the ability to reconfigure the function itself in real time can potentially result in system performance improvement when running applications.

In this paper, we have investigated adaptation and reconfiguration from two perspectives: (i) in a static way (ii) in a dynamic way. In the static way, reconfiguration of all idle units is done at the end of a learning phase in the order of their activities. In this process, only one reconfiguration is performed during program execution time. In the dynamic mechanism, functional units are continuously monitored and the reconfiguration decision is made periodically. All of the functional units are reconfigured back to their original functions in the reset mode, before applying the new reconfiguration.

This is the first research paper that explores the opportunity and benefits of deploying adaptable STT-NV logic in general-purpose processors. While in this research we mainly focus on functional units there are several other processor units that will benefit from a reconfigurable and adaptable design. In general, some of the benefits that stem from adaptable logic are (i) activity migration based on thermal profiling of the processor (ii) failure tolerance by segregating faulty units and performing fine grain reconfiguring over good ones. For example, the integer ALU is one of the hottest spot on the processor. Reconfiguring the int/multiply units and applying activity migration can reduce the temperature significantly. Such run-time reconfiguration would help migrate some of the adder functionalities onto the multiply/divide unit and help reduce the overall temperature of all units. Fine grain reconfiguration possible through MTJ based coupling would enable reconfiguration between an adder/multiplier and help segregate the faulty units by reconfiguring some of the idle good ones while maintaining the chip functionality. The novel contributions of this work are statically and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

GLSVLSI '14, May 21 – 23, 2014, Houston, TX, USA.

Copyright 2014 ACM 978-1-4503-2816-6/14/05…$15.00.

http://dx.doi.org/10.1145/2591513.2591535

dynamically adaptive reconfiguration algorithms of functional units and exploiting the STT-NV LUT properties to perform the algorithms. The rest of the paper is organized as follows. Section 2 illustrates the functional unit conflict issue. Section 3 presents LUT based reconfiguration circuitry and the circuit performance and overhead metrics. Section 4 presents the proposed adaptive algorithms. Section 5 discusses the results. Finally, section 6 concludes the work.

2 MOTIVATION

Functional unit unavailability (or alternatively functional unit conflict) is one of the major performance bottlenecks in embedded and high performance processors [1, 2]. Functional unit conflicts occur when the processor pipeline has ready instructions, but not available functional units of particular type (multiplier, for instance) to execute. Note that in spite of high functional unit conflicts, it is not design efficient to increase the number of functional units in processor pipeline, as the complexity of additional functional unit will be significant [6, 13, 15]. As studied in several works, increasing the number of functional units not only increases the power consumption of the processor but also significantly affects the complexity of several back-end pipeline stages including instruction queue, write-back buffers, bypass stage, register file design and could severely affect the processor performance, as the number of write-back ports increase significantly [6]. As the number of functional units decides processor issue width, increasing the total number of functional units (which is equivalent to the maximum issue width) from 2 (which is very common in many embedded processors) to 4, increases the critical path delay and the total power of the processor by 15% and 18% accordingly [6]. The major increase is due to the impact on the wakeup and bypass logic of the processor. In addition, several studies indicated that the utilization varies significantly across various functional units [10, 11]. In Figure 1(a) we report the percentage of execution time each of 4 groups of functional units are idle in our studied architecture. While in some architecture some functional units such as multiplier and adder can be shared in our studied architecture we assume that there is no sharing between functional units. As shown, on average integer multiply and divide unit is idle most of the time. Except from apsi which is idle for 96% of the time, for the rest of the benchmarks this unit is idle more than 99% of time. The idleness is lower for floating point add and floating point multiply and divide with average of 95% and 98% respectively. Integer add is the least idle unit; average 66% of program execution time. Such a large idle time in all functional units provide an opportunity for applying reconfiguration when the functional unit is not being used.

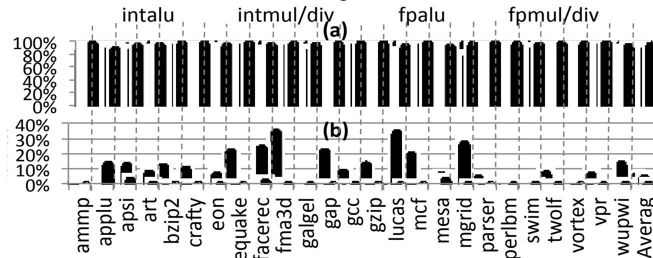

Figure 1. (a) % execution time, for which each group of functional unit is idle (b) % times with functional units conflicts.

Now the question is to which unit the idle unit needs to be reconfigured so that the performance benefit is maximized. To provide more insight in Figure 1(b) we report the percentage of times each group of functional units has been requested but was not available (functional unit conflict) during program execution time.

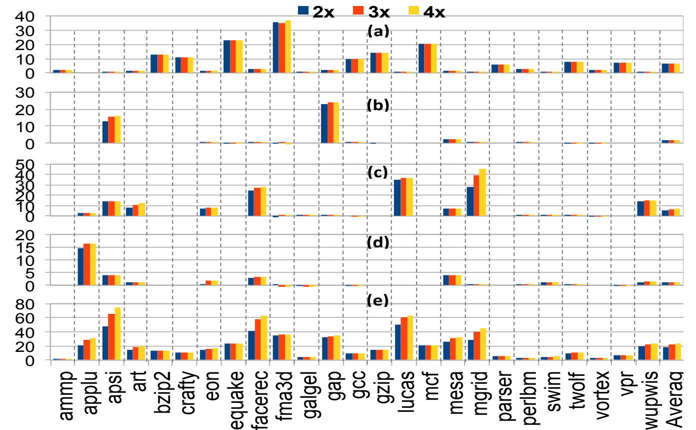

Figure 2. Relative performance improvement when the number of (a) int add (b) int mul/div (c) fp add (d) fp mul/div, and (e) all units increase by 2X, 3X and 4X {vertical bar shows the % of performance improvement}

Across most benchmarks mainly a single unit has a high conflict and therefore is the performance bottleneck. Interestingly, this unit is not the same for all benchmarks; i.e. in different benchmarks different functional unit is the performance bottleneck. While in many benchmarks integer add is the high conflict unit, in many others this is the case for floating point adder; examples are apsi, art, eon, facerec, lucas, mgrid, and wupwis. There are also few benchmarks that integer and floating point multiply and divide are the performance bottleneck units. Examples are applu, apsi and gap. Another interesting observation can be seen by comparing the results in Figure 1(a) and (b). For almost all benchmarks the same unit that is the performance bottleneck is also idle for more than 80% of program execution time. For all of these cases such a large conflict in spite of low utilization indicates that in most occasions functional units are accessed in burst. Therefore there is no single unit that is the performance bottleneck across different benchmarks. Thus finding a performance bottleneck unit to reconfigure the idle unit is a challenging problem and requires adaptive technique as we are presenting later in section 4 of this paper.

2.1 Potential for Improving Performance

In spite of large idle time for the functional units, increasing the number of functional units improve performance significantly. In Figure 2 we report the performance improvement in terms of IPC (average number of instruction committed per processor cycle) as the number of functional units increase to 2X, 3X and 4X times.

Figure 2(a) shows that increasing the number of int add improve performance significantly across many benchmarks. Interestingly, in spite of a very high idle time of integer mul/div, floating point add and floating point multiply and divide, increasing the number of these units, improve performance significantly for many benchmarks, as well. For instance in apsi and gap while int mul/div is idle for more than 96% of the time, doubling the number of this unit increase the performance by 13% and 23% respectively. To better understand this we provide the functional unit conflict results in Figure 1(b). Interestingly in these two benchmark the int mul/div unit is the main source of conflict with 11% and 22%, respectively. In fact in these benchmarks the int mul/divide is requested in burst. While the average idle time is almost 95%, there are some intervals that the unit is being accessed very frequently and therefore additional int mul/divide unit during those intervals could reduce the conflict and potentially improve performance.

Also in Figure 2(e) we report the speed up when increasing the number of all functional units at the same time. Doubling the number of functional units improve performance significantly by as

much as 50%. The average speedup is 19%. While there are some benchmarks that tripling and quadrupling the number of functional units improve their performance substantially (applu, art, facerec, lucas, mesa, mgrid), the largest speed up is achieved when doubling the resources. Further gains are seen with increased number of functional unit, but the marginal gains drop off.

3 LUT BASED FUNCTIONAL UNITS

STT-NV technology utilizes Magnetic Tunnel Junctions (MTJ) to realize nonvolatile resistive storage. There have been several attempts to use MTJs for building logic circuits with the hope of exploiting the leakage benefit of MTJs in order to reduce the power [12, 16]. However, due to the significant energy involved in changing the state of an MTJ, circuit styles that rely on changing the state of MTJs in response to input changes do not show any power and performance benefits [16]. An alternative to this approach has been to realize logic in memory by using LUTs that are built based on MTJs [12]. Resistive computation [12] replaces conventional CMOS logic with Magnetic Tunnel Junction (MTJ) based LUTs; it has been proposed for tackling the power wall.

3.1 Estimate of Area, Power, and Performance

To obtain an estimate of area, power, and performance of an LUT based adder as compared to a static CMOS (ASIC) counterpart, we have performed a case study on a 64-bit ripple carry adder and a multiplier implemented in static CMOS, CMOS LUT based, and the STT-NV LUT styles in a 32nm predictive technology node [19]. We used a commercial FPGA tool in order to get a count of LUTs and switch boxes (for routing) needed for each design. For static CMOS design we used design compiler to synthesis functional units (DesignWare) in a commercial 45nm technology and scaled the results to 32nm. Table 1 shows the results of the 64-bit adder and multiplier implemented in both styles. The results indicate that except for the leakage power, the STT-NV design has overhead in other metrics (especially for the adder).

Table 1. Comparison of adder and multiplier results in alternative styles

Metric	Unit	STT-NV LUT style	CMOS LUT style	Static CMOS style
Delay	adder	2.89	3.24	1
	multiplier	2	3.73	1
Active mode power	adder	6.46	6.70	1
	multiplier	0.74	1.26	1
Standby mode (leakage) power	adder	0.17	3.87	1
	multiplier	0.23	1.42	1
Area	adder	3.89	4.61	1
	multiplier	0.90	1.83	1

That means the performance of the reconfigurable adder in STT-NV style will be 2.89X lower than that of the static CMOS adder counterpart. Its standby mode power is 0.17X lower, but its active mode power is 6.46X higher. Due to a larger delay of reconfigurable STT-NV multiplier compared to the baseline CMOS style, the STT-NV multiplier implementation needs to be pipelined two times deeper than the original CMOS based implementation. However this has shown to impact performance minimally [12]. Also in spite of the advantage of a static CMOS based multiplier over the STT-NV based design in terms of delay, it still makes a lot of sense to replace it with the STT-NV design due to significant leakage advantage of the STT-NV design. Due to low utilization and high operating temperature of the multiplier, the standby power becomes the major component of the total power. Also as results in the table 1 suggests, the CMOS LUT based style has no obvious advantage over the static CMOS style. While both STT-NV LUT and CMOS LUT are reconfigurable, STT-NV LUT has advantage over CMOS LUT in several metrics, noticeably leakage power. The leakage power of a STT-NV style is at least 6X lower than the CMOS LUT counterpart. Based on the results presented in table 1 we select IntALU to be a non-reconfigurable static CMOS as the power and area increase for a reconfigurable IntALU is significant. Other functional units including multiplier and divider (Int and FP) are implemented with STT-NV LUT reconfigurable style where they do not incur area overhead (the area of STT-NV LUT style is even smaller than the CMOS counterpart).

3.2 Estimate of Reconfiguration Overhead

The reconfiguration energy and performance estimation is performed for configuring a 64X64 multiplier unit to a 64-bit adder unit. This represents the worst-case scenario as reconfiguration between any other pair of functional units takes less energy and delay. Reconfiguring a LUT-based multiplier to an adder unit involves programming the LUTs. We have taken the HDL of the multiplier and adder units and synthesized them using a commercial FPGA (with 6 input LUT) synthesis tool in order to get a count of LUTs needed for each design. We have also taken into account the routing overhead including the switch boxes. The multiplier unit can be realized using 437 4-input LUTs and the adder using 65 such LUTs. Hence, we assume reconfiguring the multiplier unit to the adder or vice versa involves writing to at most 65 LUTs. Therefore, the total number of STT-Non-Volatile (STT-NV) bits to be written is 65 * 16 = 1040 bits or roughly 1 Kbits. The write access time to a single bit STT-NV is estimated to be 25ns [9], which are 25 cycles for 1GHz system clock. If LUTS are written in parallel using a 128-bit wide data bus, the reconfiguration is estimated to take about 8 write operations (i.e. 200 cycles). The configuration bits for the LUTs that are different between the adder and multiplier configuration need to be stored in a ROM. A controller will read the configuration bits from ROM and write to the STT-NV LUTs. For the configuration energy estimate, we have ignored the energy of reading the configuration bits from the ROM, since the configuration energy is expected to be dominated by the energy of writing to the STT-NV cells. Using the NVSIM tool, the write energy per bit cell is estimated to be 7.9 pJ [9]. Hence, the total energy estimated for the reconfiguration of LUTs is 1040 * 7.9 pJ = 8.2 nJ. The above estimates are conservative because we assume all the bits of those 65 LUTS need to be re-written; whereas, in reality some of the bits could be same between the two configurations. In addition to programming LUT we also need to program the router and switchboxes. The routing power overhead is not trivial. We used the results of FPGA synthesis to estimate the routing energy as 3.7nJ.

4 ADAPTIVE RECONFIGURATION

In this section we are presenting the algorithms for the functional unit reconfiguration to improve performance. The adaptive algorithm we are proposing is derived from the observation made from Figure 1 where multiply and divide units (both floating point and integer) are idle for a substantial part of program execution time – more than 95% of time for many applications. Note that in spite of such high underutilization we would still need these types of functional units. However due to the infrequency of multiply and divide operations these functional unit are remaining idle for most of program execution time. Since the adder units (float and integer) are active noticeably compared to multiply and divide units, we only make multiply and divide units reconfigurable – therefore only integer and floating point multiply and divide operation can be reconfigured to either int/fp adder or to each other, for instance a multiplier to a divider. Also we have shown in Figure 2 the speedup when increasing the number of int and fp adder and we found a large performance benefit across most benchmarks. Therefore a

straightforward reconfiguration mechanism is to reconfigure multiplier and divider at run-time to an adder. Note that as shown in Figure 2 increasing the number of adder beyond a certain limit does not improve performance noticeably for many benchmarks. Therefore, it is not much performance beneficial to reconfigure all idle integer and floating point multiplier and divider to an adder.

Also as seen in Figure 2 there are few benchmarks such as applu, apsi and gap, which benefit significantly from increasing the number of multiply and divide units. Based on all of these observations, in this section we propose several algorithms to capture each benchmark behavior and adapt the number of functional unit required to maximize performance accordingly. We categorize these algorithms into static and dynamic algorithms. The goal of these algorithms is to find the idle functional units and reconfigure them to the active units to improve performance. Note that in all of these algorithms when a unit that has been reconfigured is requested and therefore is not available it needs to be reconfigured back to its original function. We refer to this re-reconfiguration as adjustment process. The adjustment process is asynchronous - For example if a multiplier is reconfigured to an adder and later in the program execution a multiply operation request a multiply unit, then the reconfigured adder need to be adjusted back to a multiplier, immediately.

4.1 Static Adaptive Algorithm

In this algorithm the application is being profiled for an initial phase (learning phase) and based on the profiling information the reconfiguration decision is being made for the rest of program execution. During the learning period active and idle functional units are being identified. At the end of the learning period all idle units are reconfigured to active units in the order of their activity. The reconfiguration pseudo-code is shown in Figure 3.

```
For the first 100M cycles:
-Monitor functional units
-Identify the idle units: idle [1, 2, 3, … i]
    (i is the total number of idle units)
-Identify the active units: active [1,2,3, … j]
    (j is the total number of active units)
-Order active units based on their activity: active_order [i]
At the end of 100M cycles:
Loop: for all idle units (i)
-Reconfigure idle units to active units: idle[i] → active_order [i%j]
```

Figure 3. Static Adaptive Algorithm pseudo code.

Note that the reconfiguration decision is made only once and after an initial learning period (after the first 100M cycles which for many benchmarks is larger than the initialization period). Since only one reconfiguration is allowed at the end of the learning phase, at most one adjustment process is performed during program execution time.

4.2 Dynamic Adaptive Algorithms

We present two dynamic algorithms shown in Figure 4. We refer to these algorithms as balanced idle to active and biased idle to most active algorithms. In both of these reconfiguration algorithms the functional units are monitored periodically and the reconfiguration decision is made every N cycles based on the functional unit activity in the previous N interval (due to space limitation we only report the results for N=100K interval). Then, at the beginning of each 100K interval, all idle functional units are reconfigured to the active ones. Such a periodic monitoring and reconfiguring process is based on the fact that many standard programs execute as a series of nonstationary phases. Each phase is very different from the others, while the program behavior within a phase is homogeneous. The goal of these periodic algorithms is to capture program behavior to find the right number of each type of functional unit for each program phase. In both of these algorithms, in the beginning of monitoring interval all units are reconfigured back to their original functions (reset process) before applying the new reconfiguration – for example if a multiplier reconfigured to an adder, then in the beginning of every monitoring interval it should be first reset back to a multiplier. Then, the dynamic reconfiguration based on the monitoring information collected in the previous interval is applied.

Balanced idle to active (Dynamic-BIA): In this algorithm all idle functional units are reconfigured to the active units in a balance way. This is shown in Figure 4(a). The order of reconfiguration is based on the activity results presented in Figure 1– first we reconfigure the idle unit to int add, then the remaining idle units are being reconfigured to fp add, fp multiply, int multiply, int divide and fp divide, respectively. This algorithm implementation is simple – we require a single bit for recording the idle functional unit during every monitoring interval. If during the monitoring intervals the functional unit was busy (even for a single cycle) we set the idle bit to busy, otherwise the functional unit is idle.

Biased idle to most active (Dynamic-BMA): In this algorithm (shown in Figure 4(b)) all idle functional units are reconfigured to the most active units in a biased way and based on their activity. From results reported in Figure 1 we observed that the integer adder unit is always the most active unit therefore it make a lot of sense to reconfigure most of idle units to an integer adder.

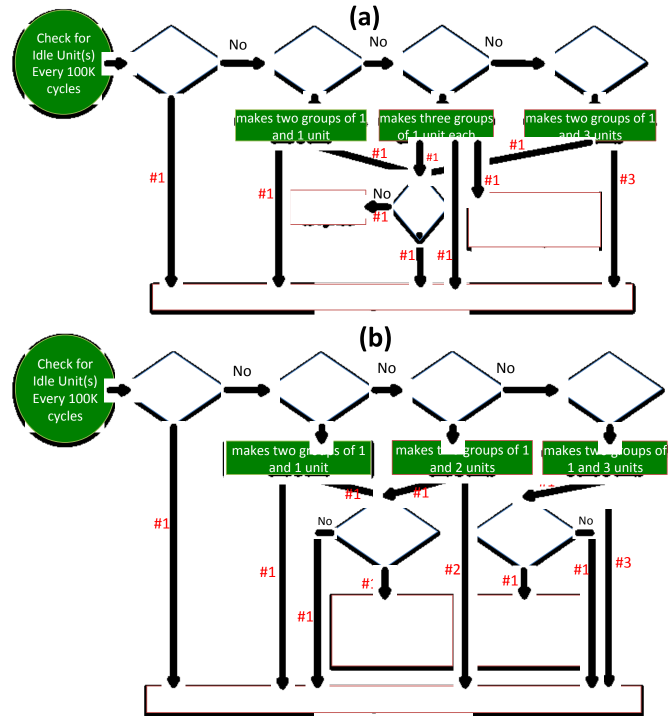

Figure 4. Dynamic Adaptive Algorithms (a) Balanced idle to active, and (b) Biased idle to most active.

For the rest of the units including fp adder, int/fp multiplier and divider the activity is monitored periodically and if they are busy more than 10K cycles in a 100K cycles monitoring interval they are considered as highly active unit. An idle unit will be then reconfigured to the most active unit out of the highly active units. The rest of idle units are reconfigured to int add. This algorithm is more complex than the Balanced algorithm as it requires constant monitoring of functional units activity, finding the highly active units, and selecting the most active among the highly active units.

5 METHODOLOGY AND RESULTS

In this section we present our simulation methodology and the results demonstrating the performance benefit of a reconfigurable STT- logic when deployed in the functional unit of the processor.

Table 2. Baseline Processor Configuration

Number of cores	4	**Register file**	64 entry
L1 I-cache	8KB, ,4 way, 2 cycles	**Memory**	50 cycles
L1 D-cache	8KB, 4 way, 2 cycles	**Instruction fetch queue**	8
L2-cache	256KB, 15 cycles	**Load/store queue**	16 entry
Pipeline	12 stages	**Complex unit**	2 INT
Processor speed	1 GHz, 1V	**Issue**	dual, out-of-order
Fetch, dispatch	2 wide	**Arithmetic units**	3 integer

As discussed earlier we only replace the integer and floating point multiply and divide CMOS unit with a reconfigurable STT-Logic. The int add and fp add remain unchanged in CMOS technology. Our baseline architecture parameter is shown in Table 2. We model a 4-core chip multiprocessor architecture using gem5 simulator. Each core is a dual issue processor similar in functionality to IBM PowerPC 750 FX architecture. We used SPEC benchmarks suite for evaluation. The benchmarks were simulated for 2 billions instructions after fast forwarding for 2 billions instructions. For the power and delay overhead associated with reconfiguration we used the results reported in sections 3.1 and 3.2.

5.1 Results

In this section we report the performance and power for the five following architecture:

-*Baseline-1X*: All functional units are implemented in CMOS and there is no reconfiguration. We assume that in baseline architecture leakage power is suppressed by power-gating technique reported in [8] with the performance loss below 2%.

-*Baseline-2X*: All functional units are implemented in CMOS and the number of functional units increased by 2X compared to baseline. As discussed in [6] in superscalar processors increasing the number of functional unit impact the processor operating clock frequency. Based on [6] we assume the clock operating frequency in this design is reduced by 15%. Similar to Baseline-1X leakage power in functional unit is being suppressed [8].

Static-Reconfig: Except int add and fp add other functional units are implemented in STT-NV technology and therefore they are reconfigured using static technique.

Dynamic-BIA-Reconfig: Except int add and fp add other functional units are implemented in STT-NV technology and therefore they are reconfigured using dynamic Balanced idle to active technique. *Dynamic-BMA-Reconfig*: similar to *Dynamic-BIA-Reconfig* except that the reconfiguration algorithm is dynamic Biased idle to most active technique. In Figure 5 we report the performance improvement of the static and dynamic algorithms normalized to the CMOS baseline architecture with no reconfiguration (Baseline-1X). We also report the performance impact of doubling the number of functional units. Since doubling the number of functional unit increases the issue width [6] and therefore impacts the operating clock frequency, for all cases we report the IPC x Clock-Frequency as a performance metric to account for the frequency impacts. In Baseline-2X design, while 2X number of functional units could potentially provide more opportunity to improve performance, in many benchmarks we observe an overall performance loss. The largest performance loss is in ammp, galgel, perl, and vortex; with more than 10% performance degradation. Interestingly these are the benchmarks where increasing functional unit does not improve IPC significantly – therefore when taking into account the impact on frequency (15%) we observe a large loss in terms of IPC x Clock-Frequency. On average a Baseline-2X can only improve performance by 2%. However, in a reconfigurable STT-NV design since there is no impact on the clock frequency we can see a noticeable performance improvement across most benchmarks. In STT-NV reconfigurable architecture, for most integer benchmarks including bzip2, crafty, eon, gap, gcc, gzip, mcf, parser, perlbmk, twolf, vortex and vpr the static technique almost match the more complex dynamic techniques. In fact for these benchmarks we observed a lot of underutilization in the functional unit and the fp units are not used for the entire program execution time. Therefore, by simply monitoring during the learning phase we can identify these idle units and reconfigure them to the heavily utilized units like int adder/multiplier and divider. Since these benchmark behavior remain almost the same after the learning phase, the simple static technique can identify the best reconfiguration and apply it for the entire program execution time. Unlike integer benchmark, for floating point benchmark the static technique cannot capture the program behavior in terms of functional unit utilization by simply monitoring the processor during learning phase. This is particularly the case for applu, mesa, mgrid, and wupwise. For these benchmarks the static algorithm during the learning phase cannot capture the performance bottleneck unit(s) (Figure 2(b)). In fact for these benchmarks the behavior of the program changes significantly after the learning phase. Comparing the two dynamic algorithms show interesting results – in many cases the Dynamic-BIA algorithm is able to capture program behavior at run-time and accordingly reconfigure the idle functional unit to the performance bottleneck ones. However there are few cases that this algorithm also cannot find the performance bottleneck units. Examples are applu, apsi and facerec where the dynamic-BIA algorithm attempts to balance the reconfiguration instead of being biased towards the performance bottleneck unit: fp add, fpmul, and fp add, respectively. On the other hand, the dynamic-BMA is biased towards performance bottleneck functional units by constantly monitoring all functional unit activities. In Figure 6 we present the power dissipation breakdown of functional units in Dynamic-BMA-Reconfig and Baseline-1X designs. To have a better understanding of the power dissipation among several benchmarks, we have separated integer benchmarks (top) from floating point benchmarks (bottom). Note that for Baseline-1X CMOS based design we assumed an state-of-the-art power gating technique has been applied to suppress the leakage power by up to 90% in floating points units and up to 45% in integer units. [6]. In both integer and floating point benchmarks, for IntMUL, IntDIV units the leakage power reduces in STT-NV Reconfigurable based design compared to a CMOS based design (results per unit not presented due to space limitation). In integer benchmarks, for IntALU, the leakage power is lower in Dynamic-BMA-Reconfig compared to CMOS Baseline-1X design. Note that in CMOS based design there is small opportunity to suppress leakage using power-gating techniques, as integer unit is busy most of the times. Overall in integer benchmarks the total leakage power of all functional units increase in Dynamic-BMA-Reconfig design compared to CMOS Baseline-1X design. In fp benchmarks the total leakage power of all functional units in Dynamic-BMA-Reconfig design reduces substantially by up to 51% compared to CMOS Baseline-1X design. The dynamic power increases in both integer and floating point benchmarks in reconfigurable design. This is somewhat expected as STT-NV reconfigurable design attempts to put more functional units into work and therefore they have higher dynamic power dissipation compared to a CMOS based design. In integer benchmark Dynamic-

BMA-Reconfig design has on average 61% higher total power dissipation compared to CMOS+PG design. This is mainly due to significant rise in dynamic power and improving in performance for STT-NV designs compared to a CMOS based design. In floating point benchmark, the total power reduces by 22% in STT-NV design compared to CMOS Baseline-1X design. Using McPAT power simulator [7] we estimated the total processor power dissipation to be increased on average by 4.1% in integer benchmarks and to be reduced on average by 2.3% in floating point benchmarks compared to a CMOS based design.

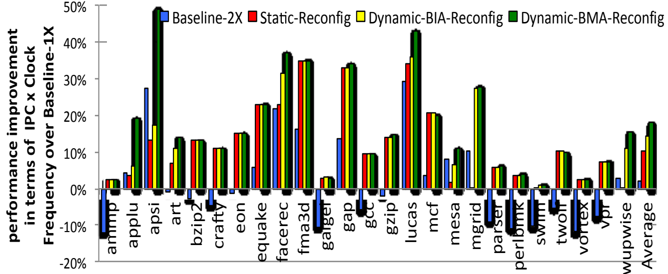

Figure 5. Relative performance improvement of various architecture with and without STT-NV.

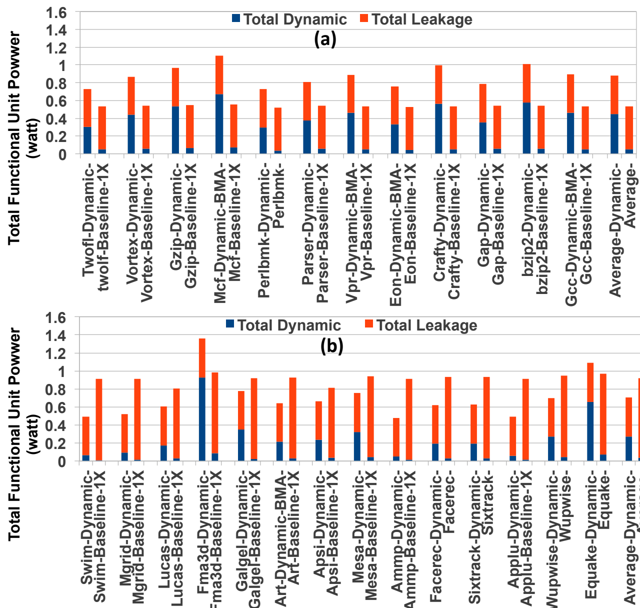

Figure 6. Total power (dynamic and leakage) of functional units for Dynamic-BMA-Reconfig and Baseline-1X in (a) Integer and (b) floating point benchmarks.

6 CONCLUSION

This paper proposes the novel concept of adaptive functional units for improving performance in general-purpose processor. Unavailability of functional units is a main source of performance bottleneck in general-purpose architectures. With functional unit adaptation we overcome this challenge. A selected set of complex functional units that might be under-utilized such as multiplier, divider, etc. are replaced with a programmable STT-NV based look up table fabric. This allows for run-time reconfiguration of such functional units to the functional units that might be creating performance bottleneck, and hence improving performance via functional redundancy and parallel computation. The results show significant performance improvement across standard benchmark. In addition to performance benefit, the new STT-NV based design and architecture is more power-efficient in floating point benchmarks compared to the a CMOS based design. Our future work will study how STT-NV reconfigurable logic can be deployed in other performance/power/temperature bottlenecks in processor architecture to improve efficiency.

7 REFERENCES

[1] Kejariwal. A., et al., “Comparative Architectural Characterization of SPEC 2000 and 2006 Benchmarks on the Intel Core Processor,” SAMOS, 2008.

[2] Folegnani, D., and A. González. "Energy-effective issue logic.". Proceedings. 28th Annual International Symposium on. IEEE, 2001.

[3] A. Kulkarni, H. Homayoun and T. Mohsenin “A Parallel and Reconfigurable Architecture for Efficient OMP Compressive Sensing Reconstruction”, The 24’th Annual Great Lakes Symposium on VLSI (GLSVLSI’2014).

[4] Kulkarni and T. Mohsenin "Parallel and Reconfigurable architectures for OMP compressive sensing reconstruction algorithm", In proceedings of The SPIE Sensing Technology and Applications Conference, May 2014.

[5] Adam Page and Tinoosh Mohsenin, “An Efficient & Reconfigurable FPGA and ASIC Implementation of a Spectral Doppler Ultrasound Imaging System”, IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 24), June 2013.

[6] Palacharla, Subbarao, Norman P. Jouppi, and James E. Smith. Complexity-effective superscalar processors. Vol. 25. No. 2. ACM, 1997.

[7] Sheng Li, et al. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures”, in Micro 2009

[8] Anita Lungu, Pradip Bose, et al, “Dynamic Power Gating with Quality Guarantees” ISLPED, 2009.

[9] X. Dong, et al. "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory." TCAD, 2012.

[10] Homayoun, Houman, and Amirali Baniasadi. "Reducing execution unit leakage power in embedded processors." Embedded Computer Systems: Architectures, Modeling, and Simulation (2006).

[11] Homayoun, Houman, Kin F. Li, and Setareh Rafatirad. "Functional units power gating in SMT processors." Communications, Computers and signal Processing. 2005 IEEE Pacific Rim Conference on. IEEE PACRIM

[12] Guo, X., et al., 2010: Resistive computation: Avoiding the power wall with low-leakage, stt-mram based computing. Power, 371–382.

[13] J. Bisasky, H. Homayoun, F. Yazdani and T. Mohsenin, “A 64-core platform for biomedical signal processing”, In proceedings of The International Symposium on Quality Electronic Design (ISQED), 2013.

[14] H. Mahmoodi, S. Lakshmipuram, M. Arora, Y. Asgarieh, H. Homayoun, B. Lin, D. Tullsen, Resistive Computation: A Critique, IEEE Computer Architecture Letter, 2013.

[15] J. Stanislaus and T. Mohsenin, “Low-complexity FPGA implementation of compressive sensing reconstruction,” IEEE International Conference on Computing, Networking and Communications, (ICNC’13), January 2013.

[16] F. Ren and D. Markovic. True energy-performance analysis of the mtj-based logic-in-memory architecture (1-bit full adder). Electron Devices, IEEE Transactions on, 57(5):2010.

[17] Adarsh Reddy Ashammagari, Hamid Mahmoodi, Houman Homayoun, Exploiting STT-NV Technology for Reconfigurable, High Performance, Low Power, and Low Temperature Functional Unit Design, DATE 2014.

[18] J. Bisasky, J. Chander, and T. Mohsenin, “A many-core platform implemented for multi-channel seizure detection,” In proceedings of The IEEE International Symposium on Circuits and Systems (ISCAS’12), 2012.

[19] Predictive technology models. http://ptm.asu.edu/.