Enabling Deep Voltage Scaling in Delay Sensitive L1 Caches

Chao Yan Russ Joseph
Electrical Engineering and Computer Science
Northwestern University
Evanston, Illinois, 60208-0834 USA
Email: {cya410, rjoseph}@eecs.northwestern.edu

Abstract—**Voltage scaling is one of the most effective techniques for providing power savings on a chip-wide basis. However, reducing supply voltage in the presence of process variation introduces significant reliability challenges for large SRAM arrays. In this work, we demonstrate that the emergence of SRAM failures in delay sensitive L1 caches presents significant impediments to voltage scaling. We show that increases in the L1 cache latency would have a detrimental impact on a processor's performance and power consumption at aggressively scaled voltages. We propose techniques for L1 instruction/data caches to enable deep voltage scaling without compromising the L1 cache latency. For the data cache, we employ fault-free windows to adaptively hold the likely accessed data using the fault-free words within each cache line. For the instruction cache, we avoid the addresses that map to defective words by relocating basic blocks. During high voltage operation, both L1 caches have full capability to support high-performance. During low voltage operation, our schemes reduce Vccmin below 400mV. Compared to a conventional cache with a Vccmin of 760mV, we reduce the energy per instruction by 64%.**

I. INTRODUCTION

While dynamic voltage frequency scaling (DVFS) remains an effective technique for trading off energy and performance, its effective scope can be severely limited by SRAM reliability. While CMOS logic within a processor core is capable of operating in the near threshold regime and can provide optimal energy operation, the SRAM cells that comprise cache structures cannot. Due to parameter variations, SRAM reliability degrades rapidly as the supply voltage decreases. This effectively creates a critical operating voltage known as Vccmin, beyond which reliable operation of the SRAM arrays cannot be guaranteed. The implications of Vccmin are widespread. If the logic and L1 caches within a core are confined to a single voltage domain, this places a bound on the lowest operating voltage for the processor and may prevent DVFS from being extended to the optimal voltage level for some workloads. While it may be possible to further reduce core voltage by providing independent voltage domains for the L1 caches, this introduces overhead in the form of additional voltage regulators and possibly level converters [1]. Even still, scaling benefits are confined to the logic.

Previous approaches using spatial/information redundancy or trading off the cache capacity have been applied to enhance the SRAM reliability. While those approaches may be satisfactory in L2 or L3 caches that are not latency sensitive, they are problematic in L1 caches where the access path is part of a critical performance loop for the entire design.

In this work, we introduce novel techniques for managing L1 data and L1 instruction caches that allow deep voltage scaling without compromising access latency. For each of the L1 caches we tailor an approach to suit the predominant nature of the access. For the L1 data cache we take advantage of spatial locality within a logical cache block to remap the most likely accessed data into the fault-free words within each physical frame. For the L1 instruction cache, we take advantage of the sequential nature of fetch and control flow to relocate basic blocks. Both of these techniques offer very high fault coverage without impacting critical paths. Moreover these approaches require little hardware overhead and do not affect high voltage operating modes. Overall, we show that it would be possible to scale the core voltage to 400mV in a 45nm technology node with 64% percent reduction in energy per instruction over a conventional cache with a Vccmin of 760mV.

The remainder of this paper is organized as follows. Section II introduces the impact of process variations on conventional SRAM cells and our model for failure probability. In Section III, we discuss the prior work on reducing the variation impact. Section IV presents the proposed schemes in detail. In Section V, we introduce the experimental methodology. Section VI evaluates our schemes with design analysis of area, latency and leakage power. We also present the simulation based results for performance and energy consumption. Section VII concludes this paper.

II. BACKGROUND: UNDERSTANDING IMPACT OF VARIATION

A. SRAM cell failures

Figure 1 shows a conventional 6T SRAM cell. The cell consists of two cross coupled inverters 'I0' and 'I1' that form a bi-stable circuit allowing the storage of either '0' or '1'. The storage nodes 'n0' and 'n1' are connected with bitlines through two pass transistors 'T0' and 'T1'. A wordline select signal (WL) is applied to the pass transistors during read/write operations.

Read: both bitlines are precharged to VCC. Then the wordline select signal turns on the pass transistors. After which, node 'n0' discharges the bitline 'BL' through transistors 'N0' and 'T0'. A sense amplifier is activated to amplify the small differential voltage across the bitlines. When the time given for discharging the 'BL' is insufficient, the differential voltage developed across the bitlines is too small to be identified by the

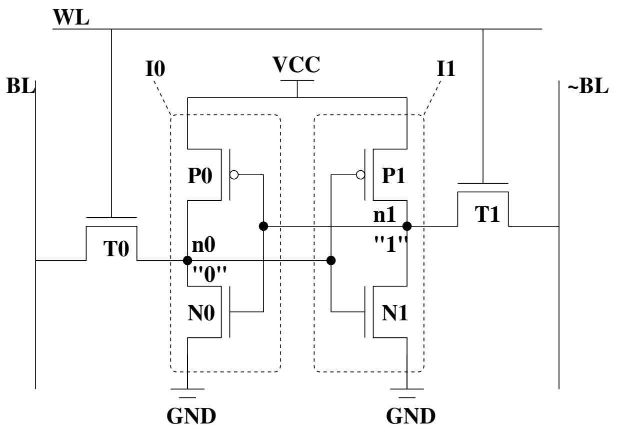


Fig. 1: Conventional 6T SRAM cell.

sense amplifier, leading to an **access time failure**. In addition, when the voltage noise rising on node 'n0' becomes larger than the switching point of the inverter 'I1', the stored value is flipped during a read operation, leading to a **read failure**.

Write: bitlines are driven to the desired value. Then the wordline select signal turns on the pass transistors. After which, the write value drives into the cell and flips the cell state. Since the pass transistor 'T1' must discharge node 'n1' in order to toggle the content of the cell, 'T1' must be overpower the pull-up transistor 'P1'. Failing to toggle the cell content during a write operation leads to a **write failure**.

Standby: a cell may lose its state due to a voltage drop, which leads to a **hold failure**.

B. *Impact of supply voltage and failure modeling*

To guarantee the reliability of SRAM cells, certain noise margins must be met at all process corners [2]. At high supply voltage, the operating margins are high enough to ensure reliable operation. However, reducing supply voltage leads to decreased noise margins. This decrease, coupled with manufacturing induced process variations, introduces significant reliability challenges to SRAM arrays. This presents the biggest impediment to lowering Vccmin. In particular, intra-die Random Dopant Fluctuation (RDF) is primarily responsible for SRAM cell failures. The RDF randomly impacts the number and the location of dopant atoms in transistors, leading to different V_{th} on neighbouring transistors, which are supposed to be matched in the original design. These random variations are modelled as independent random variables with a Gaussian distribution [3]. The failure probability (P_{fail}) is determined as a union of individual failure events.

The P_{fail} is found to be a function of supply voltage, temperature and transistor size. Figure 2 shows an example of P_{fail} over VCC for a 6T SRAM cell in 65nm technology (from [4]). The P_{fail} rises exponentially as VCC reduces in steps with DVFS. It is evident that a die that contains even a single cell failure must be discarded. Hence, the P_{fail} is the key for determining the chip yield. We assume as in [2], [4] that an acceptable manufacturing yield requires 999 out of every 1000 dies to be fault-free. For a 32KB cache, Vccmin must be above 760mV to avoid sacrificing chip yield. Figure

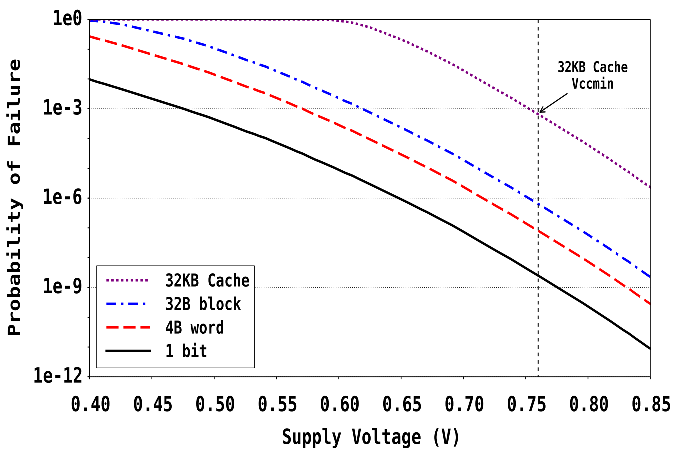


Fig. 2: Probability of failure for SRAM structures at different granularities [4].

2 also shows the P_{fail} of a 4B word and a 32B cache block. Since a cache block consists of many cells and the failure of any one of these cells could render the cache block defective, the P_{fail} of a cache block is significantly higher than that of individual words or bits. Therefore, protecting caches at finer granularity is necessary to handle the rapidly increasing P_{fail} when voltage scales beyond 500mV.

III. Related Work

In order to enhance the SRAM reliability, a cache designer has many alternatives at the circuit, architectural, and software levels. A common objective shared by the existing work is to guarantee architecturally correct execution (ACE) [5]. At the circuit level, the reliability can be improved via building more robust SRAM cells. At the architectural level, a common approach is to disable all of the defective cells at various granularities. At the software level, defective cells can be isolated by carefully controlling the address allocation of useful data. In this section we show that the existing schemes have either limited capability or prohibitive overheads for protecting L1 caches at aggressively scaled voltages. We expect our proposals to overcome those drawbacks and enable voltage scaling beyond 400mV.

A. *SRAM cell enhancement*

At the circuit level, the most intuitive way to improve reliability is to change the basic design of the cell. Upsizing the devices improves the stability of the traditional 6T based SRAM cells. The V_t variations can be reduced by 30% via doubling the channel width [3]. Variations on SRAM cells can also be mitigated using novel topologies like 8T cell [6], 10T cell [7] or ST cell [8]. These designs deliver better low voltage reliability at the cost of a 30% to 100% area overhead. Therefore, the cell enhancement approaches are appropriate only for small array structures like register files. In contrast, our proposals are more area efficient at protecting large array structures like caches, where increasing cell area would lead to either decrease in overall capacity and/or increase in access time.

B. Variation aware architecture

At the architecture level, defective portions of a cache are isolated at the cost of redundant cells or reduced cache capacity.

Redundancy based approaches use redundant space for enhancing the SRAM reliability. Spare rows/columns are the typical redundancy employed for yield saving [9] and reliability enhancement [10]. In these schemes, redundant rows/columns are added to SRAM arrays for replacing defective rows/columns. The cache remains the same capacity, but the amount of available redundancy limits the capability of fault coverage. Thus, the redundant rows/columns are only capable of tolerating a small number of failures.

Error detection/correction code (EDC/ECC) is another form of redundancy for fault tolerance. It has proven to be effective at tolerating infrequent transient faults. However, caches need to be armed with multi-bit ECC to work under high defect density. ECC/EDC requires extra storage for the check bits as well as high latency decoders for error detection/correction. Kim et al. [11] proposed 2D ECC to provide scalable multi-bit error protection for SRAM arrays. While this scheme has a low cost in check bits, it requires updating the column code after each write and multiple cycles for correcting errors. It leads to high power and delay overhead. Recent work attempts to reduce the ECC overhead using variable strength ECC [12] or performing error detection/correction at a finer scale [13]. However, with aggressive voltage scaling, multi-bit errors become increasingly likely and quickly overwhelm the capability of ECC. Unlike the redundancy based approaches, our proposals are capable of tolerating very high P_{fail} ($\geq 1e{-}2$), while having small overhead on area and latency.

Resizing based approaches trade-off cache capacity for better reliability. A typical solution is to disable defective lines/rows/ways using gated-vdd or remapping. The gated-vdd scheme [14] cuts off the supply voltage of the defective way. It also turns off the decoders, precharge circuits and sense amplifiers to eliminate the static power. On the other hand, the remapping schemes ([15], [16]) remap the defective way to a neighbouring fault-free way by altering the way index. Disabling at cache line level gracefully degrades cache capacity and recovers decent proportions of a cache at moderate failure rates ($P_{fail} \leq 1e{-}3$) [17]. However, it is not sufficient at aggressively scaled voltages where almost every cache line is expected to be faulty.

Disabling defective cells at smaller granularities avoids the excessive loss of cache capacity when dealing with high defect densities. Wikerson et al. [4] proposed two word level fault tolerant mechanisms. Word-disable combines two consecutive cache lines to form a fault-free line. Bit-fix uses a quarter of the cache towards repairing defective cells. These two schemes reduce Vccmin to 500mV in 65nm technology, while sacrificing cache capacity by 50% and 25% respectively.

Of fine grained defect-aware caches, the word substitution based techniques are particularly important. A graph based algorithm is used to flexibly group a sacrificial cache line with one or more cache lines. The cache lines within a group are collision free in the respective defective word locations. In this way, defective words of one line can be substituted with fault-free words of the sacrificial line. Examples include ZerehCache [18], Archipelago cache [19] and Macho cache [20]. These techniques attempt to reduce Vccmin to 400mV. However, additional multiplexing and associated logic for Mux-control are required in the critical path. It introduces extra latency to the cache access. Since the hit latency of L1 caches are particularly important for performance, the word substitution schemes are mostly suitable for protecting L2 caches.

Tayyeb et al. [2] introduce a Fault Buffer Array (FBA) that stores defective words in a small word-location-tagged cache. Accesses to the faulty words are redirected to the FBA. The faulty words that miss the FBA are dealt like normal cache misses. These methods are proven to be energy efficient at moderate defect densities ($P_{fail} \leq 1e{-}3$). However, the FBA requires expensive content addressable tag array. It makes the size of FBA a serious limitation at high defeat rate. Similarly, Sasan et al. [21] propose an Inquisitive Defect Cache (IDC) that maps in-use defective words to an auxiliary set-associative cache. Its effectiveness is constrained by the IDC size and the feasible degree of associativity. In this work, the idea of simple word disabling, where defective words that miss the cache are handled like normal cache misses, is extended for an area, delay and energy efficient solution. A new architecture that explores the runtime spatial locality is proposed for L1 data caches.

C. Software solutions

At the software level, compiler techniques are proposed to minimize the hardware overhead of fault tolerance. Meixner and Sorin [22] introduce I-Cache Detouring which keeps the core from touching faulty cache sets by inserting an unconditional jump right before each defective line. This scheme is effective for tolerating hard faults in simple cores where hardware resources are very limited. Nevertheless, a software approach that is able to disable defective cells at word granularity is highly desirable for achieving lower Vccmin. This work propose a novel software based approach to protect the L1 instruction cache by relocating program basic blocks. Compared to I-Cache Detouring, it allows reliable operation below 400mV ($P_{fail} \geq 1e{-}2$) with minimum hardware modifications.

IV. TWO FAULT TOLERANCE MECHANISMS

In this section, we describe two fault tolerance mechanisms which permit reliable low voltage operation in L1 data cache and L1 instruction cache respectively. The first mechanism, fault-free window (FFW), identifies defective words and dynamically maps the most likely accesses into fault-free words in each cache line. The second mechanism, basic block relocation (BBR), identifies defective words and maps program basic blocks into contiguous fault-free words. Both mechanisms address SRAM cell failures in cache data arrays. The tag arrays and other memory structures are implemented using robust 8T cells [6] which allow the tag array of a 32KB cache to operate at 400mV. Since the tag array and the extra structures for fault tolerance are much smaller than the data array, using 8T cells would not incur high overhead (see Section 6). The L1 caches are assumed to be addressed at a 32-bit word level. We leverage Built-in Self-test (BIST) ([4], [23]) to identify defective words at all system supported DVFS operating points. BIST checks read/write functionality

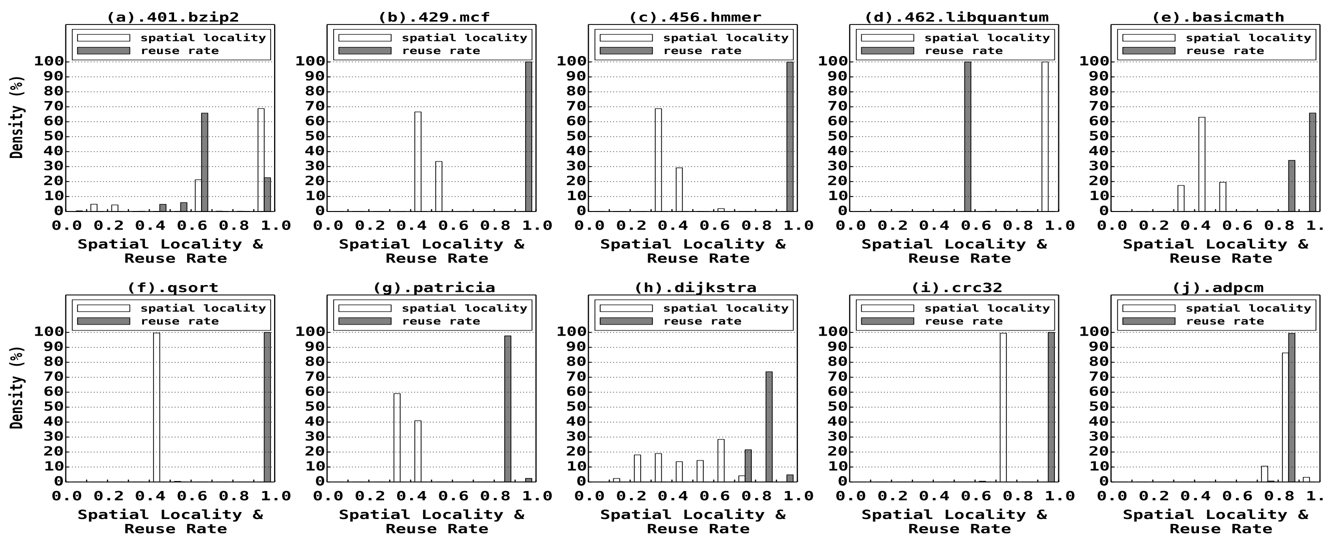


Fig. 3: Normalized histogram of spatial locality and word reuse rate. The results are accumulated from 10000 representative intervals for each application. Most applications exhibit poor spatial locality and/or high reuse rate.

of an SRAM array by writing test patterns and then evaluating the read responses. Defective words are transferred to an off chip storage and are recorded in fault maps ([2], [4]).

A. *Fault-free window in data caches*

1) Spatial locality and word reuse: The FFW based scheme was motivated by the observation that the majority of applications exhibit low spatial locality and/or high word reuse rate. This means that only a portion of the words in a cache block are likely to be accessed repeatedly during a given program interval. The goal of the FFW is to minimize the impact of defective words by allowing cache lines to store those most likely accesses using the available fault-free words.

Figure 3 shows the normalized histogram of the spatial locality and the word reuse rate in data caches. To quantify the spatial locality, we use the method proposed in [24]. Every fixed interval of 10000 instructions during an application's representative 100 million instruction trace is examined. The spatial locality is described as the ratio of data which the application actually uses to the total cache line size. We also show the word reuse rate, which is described as the ratio of the repeated accesses on unique words to the sum of the word accesses.

The results show that most applications exhibit poor spatial locality and/or high reuse rate. For example, 429.mcf, 456.hmmer, basicmath, qsort, patricia and dijkstra have only 30% to 60% of the words being accessed during a program interval and more than 80% of the accesses are repeated. Moreover, 401.bzip2, crc32 and adpcm have over 60% of the words being accessed and more than 60% of accesses are repeated. 462.libquantum is the only exception that has high spatial locality and low word reuse rate.

2) Fault-free window for capturing likely accesses: The FFW is a mechanism that takes advantage of the low spatial locality and high word reuse rate to dynamically capture the active words within each logical cache block. When a physical frame contains defective words, it can still hold a partial cache block using the remaining fault-free words. When a cache block arrives, contiguous words are scattered to fault-free word entries in the physical frame. It logically forms a fault-free window which is able to capture the likely accesses. Assuming a write through cache, where accesses to the missing words can be treated as normal cache misses. When the requested words are not in the cache, it triggers a read request to the next level in the memory hierarchy. After a miss penalty, the corresponding block arrives at the cache. Then the content of the fault-free window can be updated based on the observation that an application is likely to reference the memory locations that are close to the missing words in the near future.

Figure 4 shows the proposed cache architecture. The stored pattern array (StoredPattern) stores valid bits of the cache words. If there is a tag hit, the data arrays column MUX selects the hitting cache way for data transaction. However, the requested word may not be in the cache. In the StoredPattern array, MUX1 selects the stored pattern of the hitting cache way using matched way index. Subsequently, MUX2 selects the valid bit of the requested word using the word offset. A hit signal is generated when the address hits in the tag array and the valid bit is 1. The fault map array (FMAP) stores defect marks of the cache words. When a processor switches to low voltage mode, the fault map which corresponds to the current operating condition is loaded into FMAP. Fault maps can be stored in main memory and loaded into FMAP using special instructions or system calls [2]. The FMAP employs MUX3 to select the fault pattern of the hitting cache way using matched way index. Since contiguous logic words are scattered to fault-free word entries in each physical frame, the word offsets are altered based on the stored pattern and the fault pattern of the hitting cache way. Word remapping logic is employed to calculate the actual word offset before the data array's column MUX selects the requesting word.

In order to minimize cache misses, a fault-free window should move along with the likely accesses in a cache block.

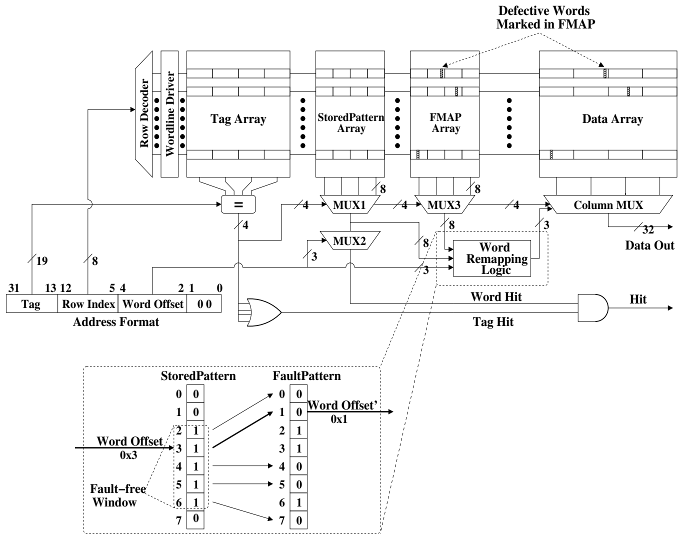


Fig. 4: Fault-free window based cache architecture. We also show an example of computing the actual word offset. The stored pattern '01111100' indicates that the fault-free window in the hitting cache way contains logic word 2 to 6. A word offset '0x3' points to the second word in the fault-free window. It is mapped to the second fault-free word entry '0x1' in the physical frame. Therefore, the real word offset '0x1' is sent to data array's column MUX to select the requesting word.

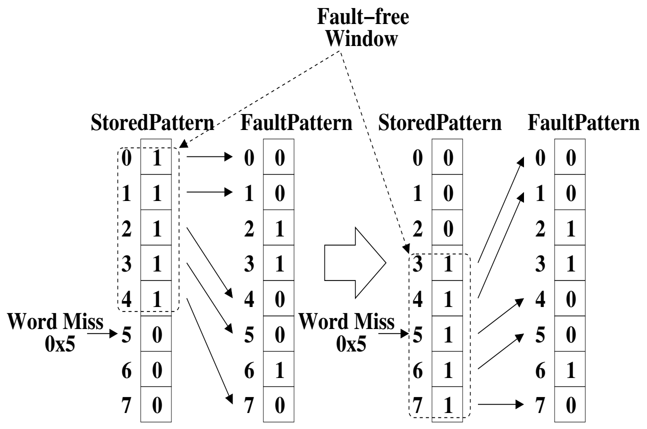


Fig. 5: Updating fault-free windows.

The application's spatial locality shows that if an application references a word that misses the cache, it is likely to reference the memory locations that are close to the missing word very soon. Thus, a fault-free window could satisfy an application's access pattern by storing the words that are close to the missing word. Figure 5 shows an example of updating the fault-free window. When a cache block arrives at the cache for the first time, it stores a default pattern. In the example the default pattern is the first five contiguous words. There is no cache miss until the application references word 5, 6 or 7. Suppose referencing word 5 causes the first cache miss. After a miss penalty the corresponding cache block comes from the lower level cache, the fault-free window changes its stored pattern and moves towards the missing word. In our design, we let the missing word stand in the middle of the new fault-free window. Updating the fault-free window is on the cache miss path, which is not critical compared to the hit path. Moreover, the missing word can be forwarded to CPU before updating the fault free window. Therefore, it requires no extra access latency.

B. Basic block relocation in instruction caches

1) Distribution of effective capacity: The BBR based scheme was motivated by the observation that the remaining fault-free words in the instruction cache are often sufficient to capture the application's working set during each program interval. Note that this may not be true for applications with large instruction working sets like commercial workloads, but this is a valid assumption for the embedded benchmarks evaluated in this paper. The goal of the BBR is to prevent the core from touching defective words by remapping each basic block to a group of contiguous fault-free words (known as a fault-free chunk), which is capable of holding that basic block.

Figure 6 (a) plots the worst case distribution of the effective capacity of a 32KB instruction cache when executing 'basicmath' at 400mV. Suppose that the basic blocks can move as needed and settle in the first available fault-free chunk. We examine the distribution of the cache capacity for every fixed interval of 1 million instructions in the application's representative 100 million instruction trace.

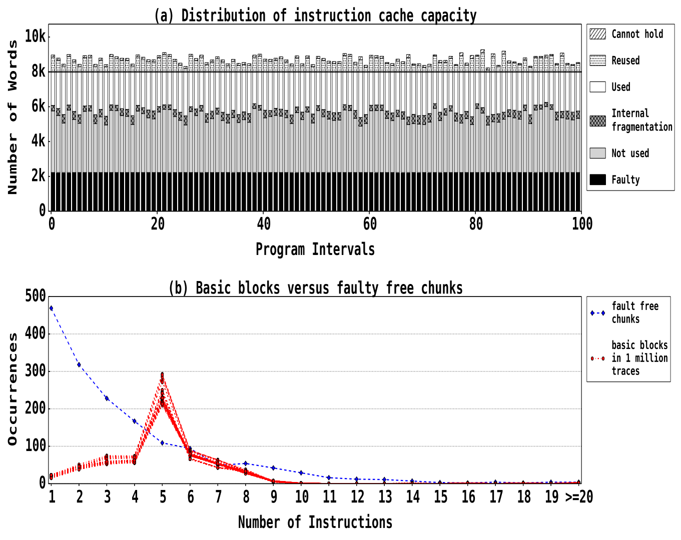


Fig. 6: Worst case distribution of the instruction cache capacity. Despite the presence of defective words, the instruction cache still has enough fault-free words to capture the application's working set during each program interval.

Although the defective words make a significant portion of the cache unavailable, the cache is still not fully utilized mostly due to the application's small memory footprint during each interval. Certain fault-free chunks are being shared by multiple basic blocks that are too large to be relocated to any of the unused fault-free chunks. This sharing may introduce more conflicts, but it only contributes to a small portion of the cache accesses. Figure 6 (b) explains the degree of sharing by comparing the distribution of the basic block size and the fault-free chunk size. The basic blocks that contain 5 instructions are the major causes of the sharing. Previous studies have reported an average basic block size of 5 to 6 instructions for most CPU intensive benchmarks [25], [26]. The basic blocks of this typical size are not problematic since they could be moved to larger fault-free chunks. After all, the available fault-free chunks in the instruction cache are sufficient to capture the application working set.

2) Basic block relocation for isolating defective words: The key to the BBR is to pad the code such that a basic block can be relocated to a proper fault free chunk. The padding can be performed in many contexts — via static compilation, Just-in-time (JIT) compilation or binary translation. In our implementation of BBR, it is performed by the linker, which avoids addresses that map to defective words when placing instructions. Since software driven relocation needs direct control over the location where instructions are placed in the instruction cache, a direct mapped cache is required in low voltage mode. In this work, we change the basic cache design to support dynamic switching of cache operation between set-associative and direct-mapped modes. The instruction cache can be configured as set-associative during high voltage operation to permit high performance, and as direct-mapped during low voltage operation to support BBR. Therefore, the hardware has no overhead when high voltages preclude defects.

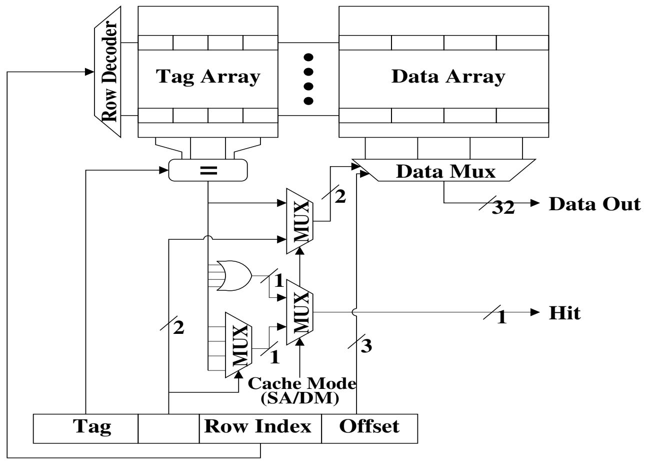
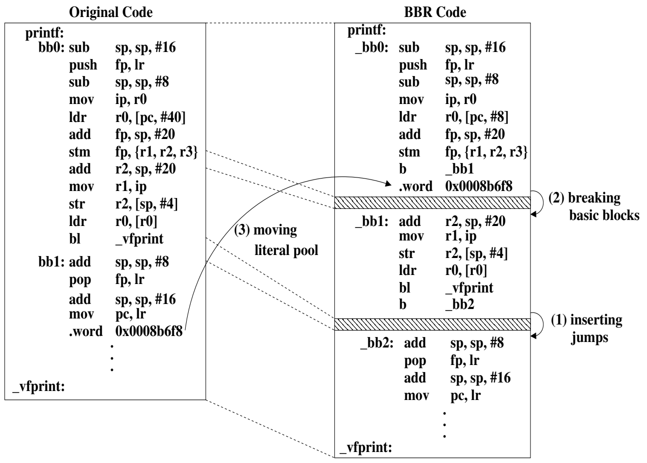


Fig. 7: Cache architecture that supports dynamic switching between set-associative mode (SA) and direct-mapped mode (DM), similar to [27].

To implement direct-mapped accesses on top of a set associative cache, least significant bits of the tag are used to explicitly select a way in the selected set. This approach is first proposed in the Dynamic Associative Cache (DAC) [27], which uses shadow tags to track hypothetical cache performance and dynamically switch between direct-mapped and set-associative to reduce the power consumption. Unlike DAC, the cache proposed here does not require shadow tags for monitoring performance. When the processor switches to low voltage mode, all cache contents are invalidated and the cache is configured as direct-mapped. Since the cache stays direct-mapped throughout the low voltage mode, the performance impact of mode switch is ignorable. Figure 7 shows the architecture of a 4-way set associative cache which supports dynamic mode switching. The the least significant tag bits and the index bits are combined to select the desired cache line in the direct-mapped mode.

With hardware support, the instruction cache circuitry is under direct software control. The linker is able to decide where instructions are placed in the instruction cache by managing the instruction addresses. Nevertheless, the program code needs to be transformed before the basic blocks can move freely. In our implementation, this job is done by the compiler, which starts with the program source code and generates object files. In this work, we evaluate an ARM system, but in principle BBR can be applied to any ISA given appropriate modifications to the binary.



```
printf:
  bb0: sub    sp, sp, #16
       push   fp, lr
       sub    sp, sp, #8
       mov    ip, r0
       ldr    r0, [pc, #40]
       add    fp, sp, #20
       stm    fp, {r1, r2, r3}
       add    r2, sp, #20
       mov    r1, ip
       str    r2, [sp, #4]
       ldr    r0, [r0]
       bl     _vfprint
  bb1: add    sp, sp, #8
       pop    fp, lr
       add    sp, sp, #16
       mov    pc, lr
       .word  0x0008b6f8
           .
           .
           .
_vfprint:
```

```
printf:
  _bb0: sub    sp, sp, #16
        push   fp, lr
        sub    sp, sp, #8
        mov    ip, r0
        ldr    r0, [pc, #8]
        add    fp, sp, #20
        stm    fp, {r1, r2, r3}
        b      _bb1
        .word  0x0008b6f8
  _bb1: add    r2, sp, #20
        mov    r1, ip
        str    r2, [sp, #4]
        ldr    r0, [r0]
        bl     _vfprint
        b      _bb2
  _bb2: add    sp, sp, #8
        pop    fp, lr
        add    sp, sp, #16
        mov    pc, lr
            .
            .
            .
_vfprint:
```

Fig. 8: Code transformation that supports basic block relocation.

Figure 8 shows an example of the code transformation on function 'printf' from the standard library. The compiler makes three types of transformation to support BBR.

(1) Inserting jumps: the basic block 'bb0' is followed by a succeeding basic block 'bb1'. The program control flow may transfer from 'bb0' to 'bb1' (fall through). If the compiler inserts an unconditional jump to the end of basic block 'bb0', the linker is able to relocate 'bb0' and maintains the program control flow by changing the target address of the new jump instruction.

(2) **Breaking basic blocks**: the basic block 'bb0' may be too large to be relocated to any of the fault free chunks in the instruction cache. If the compiler breaks 'bb0' into two smaller basic blocks '_bb0' and '_bb1' by inserting an unconditional jump, the instruction cache can hold these two basic blocks using smaller fault free chunks.

(3) **Moving literal pool**: a load instruction with PC-relative address is employed to read constant values from a literal pool. The load instruction and the literal pool are required to be within a memory page (4KB). To relocate a basic block that contains such load instructions, the compiler needs to move the corresponding literal pool to the end of the basic block.

The code transformation is applied to all of the program components including the program code, standard libraries and run time libraries.

After the code transformation, the linker reads the object files and manages the instruction addresses according to fault maps. It treats each basic block as a relocatable section and controls their starting addresses by inserting gaps among basic blocks. Moreover, it changes the target address of the jump instruction at the end of each basic block to maintain the program control flow. We use a simple and fast algorithm to match basic blocks to fault free chunks. It maintains a global pointer, which points to the current position in the fault map. For a given basic block, it scans the fault map starting from the current position and matches the basic block to the first appropriate fault free chunk. Then it moves the pointer to the end of that basic block and proceed to match the next basic block. The matching algorithm loops around the instruction cache and tries to utilize all of the fault free chunks in an effort to hold as many basic blocks as possible. In such a way, all of the defective words in the instruction cache are avoided. Algorithm 1 shows pseudocode of the matching process. Note that the BBR is valid only for an instance of the instruction cache at a specific DVFS operating point. The caches using BBR must be flushed when converting to a lower supply voltage and hence higher P_{fail}.

Algorithm 1 Matching basic blocks to fault-free chunks.

```
Input:
    BB: basic blocks
    FMAP: fault map of the cache    ▷ one bit per word
    memAddr: starting memory address
    csize: number of words in the cache
Output:
    bbMap: a map of basic block starting addresses

 1: procedure MATCH($BB$, $FMAP$, $memAddr$, $csize$)
 2:     initialize $bbMap$ ← empty map
 3:     foreach $bb \in BB$ do
 4:         $cacheAddr \leftarrow memAddr$ mod $csize$
            ▷ start scanning FMAP.
 5:         while the fault-free chunk starting at cacheAddr is smaller than $bb$ size do
 6:             $memAddr \leftarrow$ next word address in memory
 7:             $cacheAddr \leftarrow memAddr$ mod $csize$
 8:         end while
            ▷ Assume each bb can find a fault-free chunk, the while loop would finish.
 9:         add mapping $(bb, bbAddr)$ to $bbMap$
10:         $memAddr \leftarrow memAddr + bb$ size
11:     end for
12:     return $bbMap$
13: end procedure
```

V. METHODOLOGY

In order to evaluate our proposals, we model a microprocessor system using the gem5 simulation infrastructure [28]. The simulated processor configuration is depicted in Table I. The processor features a 2 way superscalar pipeline and 32KB dedicated L1 caches with 32B blocks. It models embedded microprocessors like ARM Cortex A9. The workload includes 4 SPEC2006 [29] benchmarks and 6 Mibench [30] benchmarks compiled for ARM ISA. In the experiments, we mainly focus on evaluating embedded applications. While applications with greater live footprints are also included. A comprehensive study of the limit of application live footprints is a part of our future work.

DVFS is only applied to the core logic and L1 caches, whereas the L2 cache is supplied with a separate fixed voltage. To permit synchronized operation, frequency scaling is applied to the L2 cache. Table II shows the DVFS configuration. Our experiments are based on the P_{fail} in 45nm technology (reported in [2]). The core frequencies are estimated assuming 20 FO4 delays per cycle. The FO4 delay is measured with Hspice simulation on a FO4 inverter chain. The region of interest lies between 560mV and 400mV, where P_{fail} rises exponentially from $1e^{-4}$ to $1e^{-2}$. In this region, fault tolerance becomes quite challenging since most of the coarse-grained techniques cannot guarantee reliable operation.

(a) Core Configuration	
Microarchitecture	2-way superscalar (gem5 arm-detailed)
Clock Speed	1.9GHz
Functional units	2 INT ALUs, 1 FP ALU, 1 INT MULT, 1 FP MULT
Physical Registers	128 INT, 128 FP
Reorder buffer	128 entries
Load/store queue	64 entries
Branch history table	4096 entries
Branch target buffer	512 entries, 8-way

(b) Memory hierarchy	
L1 instruction cache	32kB, 4-way, 32B blocks, LRU, 2 cycles
L1 data cache	32kb, 4-way, 32B blocks, LRU, 2cycles, write through
Unified L2 cache	512kB, 8-way, 32B blocks, LRU, 10 cycles, write back

TABLE I: Processor configuration

DVFS configuration		
Core voltage (mV)	Core frequency (MHz)	P_{fail} (from [2])
760	1607	0
560	1089	$1e^{-4.0}$
520	958	$1e^{-3.5}$
480	818	$1e^{-3.0}$
440	638	$1e^{-2.5}$
400	475	$1e^{-2.0}$

TABLE II: DVFS configuration

For a 32KB cache, the supply voltage can be decreased to 760mV without sacrificing the 99.9% target chip yield. Therefore, we simulate a baseline cache with a Vccmin of 760mV to quantify the relative energy savings. In order to understand the relative performance loss due to the fault tolerance overheads on cache latency and capacity, we also simulate an unrealistic baseline cache that has defect-free operation without latency overhead and capacity loss at each DVFS operating point.

Due to the random nature of SRAM failures, the experiments are based on the Monte Carlo method. By repeating simulations on random samples, the results represent common cache accesses. To have statistically meaningful results, we generate up to 1000 faultmaps for both instruction cache and data cache at each DVFS operating point. The instruction cache faultmaps are used to decide basic block placement at link time. On the other hand, the data cache faultmaps are used to guide the update of fault-free windows during the simulation. For each simulation, we execute each program with reference inputs by running to completion. With these faultmaps the results achieve 95% confidence interval and 5% margin of error.

We implemented the code transformation in LLVM 3.7.0 [31]. Except for the program code, it is also applied to the standard library (libc) and the runtime library (compiler_rt), which are linked to every compiled program. The basic block relocation is implemented in the linker. After code transformation, the linker manages basic block placements based on the matching algorithms. The basic block relocation is controlled using command line flags during the compiler invocation. It has no impact on code generation unless it is explicitly enabled.

Since FFW and BBR manage the data cache and the instruction cache independently, combining these approaches does not require extra effort. The combined FFW and BBR approach is compared with a robust 8T-based cache (8T) [6] and recently proposed architectural approaches including Simple Word Disable (Simple-wdis) [2], Wilkerson's word disable (Wilkerson) [4], Fault Buffer Array (FBA) [2] and Inquisitive Defect Cache (IDC) [21]. For all those schemes, the data arrays are based on conventional 6T cells. The tag arrays and other memory structures are implemented using robust 8T cells. Finally, we leverage CACTI 6.5 [32] and MCPAT [33] to estimate area, latency, static power and dynamic power in 45nm technology.

VI. RESULTS AND DISCUSSIONS

In this section, we present the experimental results and the comparison of fault tolerance techniques for L1 caches. The ultimate goal of the evaluation is to determine the effectiveness of our approaches for promoting energy reduction and maintaining acceptable performance. We analyze our scheme and compare the results to a robust 8T-based cache and recently proposed fine-grained techniques. We conduct both design analysis on latency, area overhead, leakage power and simulation based analysis on performance and energy reduction.

A. Static comparison

In this section, we present the static comparison of area, latency and static power. The area overhead is important for choosing the fault tolerance approaches, especially for area constrained cache designs as in the case of embedded processors. On the other hand, latency and static power have a direct impact on performance and energy consumption. That impact is explained in the following two sections. Table III summarizes the cache area, latency overhead and the static power of each scheme in low voltage mode. We normalize the area and static power to the conventional 6T based cache, while showing the latency overhead as number of extra cycles.

Scheme	Normalized Area	Normalized Static Power	Latency overhead
8T cache	128.0%	100.2%	1 cycle
FFW (dcache)	105.2%	106.4%	0 cycle
BBR (icache)	101.1%	100.1%	0 cycle
FBA (64 entries)	112.0%	106.1%	1 cycle
Wilkerson	103.4%	104.5%	1 cycle
IDC (64 entries)	113.7%	105.9%	1 cycle
Simple wdis	103.3%	103.6%	0 cycle

TABLE III: Static overheads

1) Area overhead: The area overhead of the FFW based data cache comes from the extension of the tag array to accommodate FMAP and StoredPattern (Figure 4). Since the FFW only protects cache data arrays, tag arrays with FMAP and StoredPattern are implemented using robust 8T cells. Compared to conventional 6T cells, 8T cells increase the memory cell area by 30% [34]. However, the area of the remaining cache components like decoder, sense amplifier and inter-bank wires may change along with the optimal cache organization. We modify CACTI [32] to estimate the area. The FFW increases the data cache area by 5.2% (1% tag, 4.2% FMAP and StoredPattern). On the other hand, the area overhead of the BBR based instruction cache comes from the implementation of the direct mapped mode on top of a set associative cache (Figure 7). This requires extra multiplexers to explicitly select the hit signal as well as the column index. Moreover, the 8T-based cache tag array increases the memory cell area. CACTI shows that the BBR increases the instruction cache area by 1.1% (1% tag, 0.1% multiplexers)

2) Static power: In order to measure the static power, we extend cache tag arrays with the extra structures for fault tolerance. We also add a static power model for 8T cells. The 8T cell has slightly higher leakage power because it has one more leakage current path than the conventional 6T cell. However, the two transistors on this extra leakage current path have a stack effect that reduces the sub-threshold leakage a bit [34]. As a result, the overall difference for leakage power is relatively small (0.2%). As shown in Table III, the FFW based data cache increases the static power by 6.4% compared to the conventional 6T based cache. Although this value is slightly higher than the other architectural approaches, the FFW allows deeper voltage scaling and significantly higher energy reduction for the entire processor (See Figure 12). Assuming that static power scales linearly with the supply voltage, the values scale uniformly and hold for all cache organizations at each DVFS operating point. On the other hand, the static power of the BBR based instruction cache depends on the cache access mode. In direct-mapped mode, access to all but one hitting cache way can be disabled. On the other hand, in set-associative mode, regular access to all cache ways is performed. In order to reduce the design complexity, we abandon the logic for disabling the missing cache ways in our design and use the static power of the set-associative mode for the BBR based instruction cache. After all, the BBR based instruction cache increases the static power by 0.1% compared to the conventional 6T based caches.

3) Access latency: We evaluate the access latency of the 32KB FFW based data cache in 45nm technology. Figure 9 shows the timeline of each critical path in the data cache (see Figure 4 for architectural organization). To normalize values across technology nodes, the delays are given in FO4. The data array, tag array, stored pattern and fault pattern are accessed in parallel. Since the data array is much bigger than the remaining three components, the cache access time is decided by data array. In addition, because stored pattern and fault pattern are the longest critical paths next to data array and their delays (39.4) are smaller than the row address to column MUX delay of the data array (42.2), the overall access latency remains the same as a conventional 6T based cache. On the other hand, the BBR based instruction cache only adds one MUX delay to the tag array (Figure 7). Since the tag array and the data array are accessed in parallel, there is no latency overhead compared to the baseline 6T based cache.

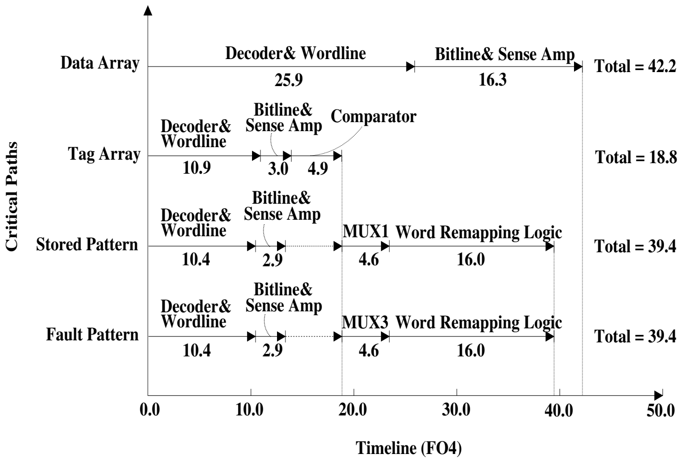


Fig. 9: Timeline of each critical path in the FFW based data cache.

B. Performance

In this section, we compare the performance of our scheme to other techniques at low voltages ranging from 560mV to 400mV. At each DVFS operating point, we measure the program runtime for each simulation and calculate the average runtime of all the simulations. We then normalize the average runtime to the defect-free baseline cache. We also divide the runtime values into three components using the measurement approach proposed in [35].

Figure 10 shows the normalized runtime of different fault tolerance mechanisms at interesting DVFS operating points. While Wilkerson's word disable cannot achieve 99.9% chip yield below 480mV, we give it the benefit of the doubt by applying simple word disable as a supplementary technique (called $Wilkerson^+$). Moreover, we evaluate optimistic implementations of FBA and IDC by granting them 1024 entries (called FBA^+ and IDC^+ respectively). We also give the 8T

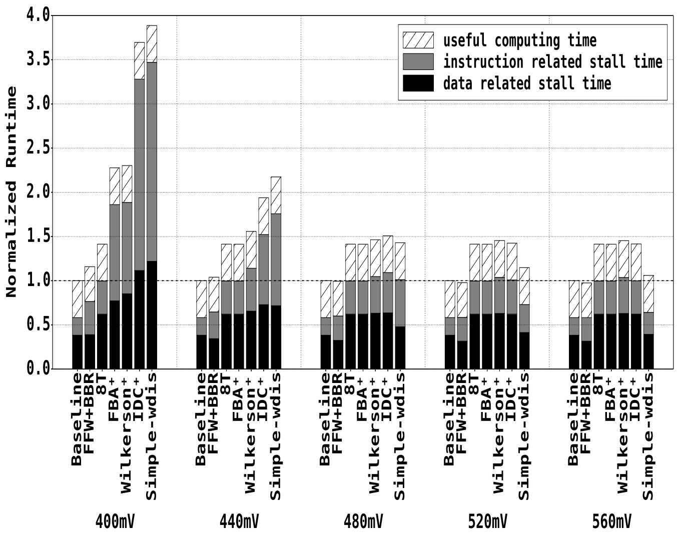


Fig. 10: Normalized overall runtime.

based cache 1 extra cycle assuming that the 28% increased area may cause the latency overhead in wire delay dominated cache structures.

Before 480mV ($P_{fail} = 1e^{-3}$), the performance is very sensitive to the L1 latency. For example, IDC^+, $Wilkerson^+$, FBA^+ and 8T suffer more than 40% performance loss at 560mV ($P_{fail} = 1e^{-4}$) mostly due to the 1 cycle extra latency on L1 caches. On the other hand, Simple-wdis incurs only 5.9% performance loss since it has no latency overhead. The small performance loss is because of the slightly increased L2 accesses that are caused by defective words. Note that our approach incurs slightly higher performance since the BBR changes the default basic block placements. The L1 latency continues to dominate the performance until the increased L2 cache accesses become a bigger problem.

After 480mV, as defective words become increasingly overwhelming, the increased L2 cache accesses start dominating the performance. Simple-wdis bears the brunt of the impact and suffers severe performance loss. On the other hand, FBA^+ and IDC^+ start to achieve better performance than Simple-wdis by dynamically reusing their substitution words. Since the FBA^+ and the IDC^+ have 1024 entries, they are able to achieve fair performance recovery in this region. In the real design, the number of substitution words is much smaller than 1024 and may become a limitation at low voltage. Our scheme achieves greater than 50% performance compared to FBA^+, which is the best among other architectural approaches. Nevertheless, our efficiency comes at zero latency overhead and lowest area overhead compared to other schemes.

Figure 11 shows the number of L2 cache accesses. Our approach is the only architectural solution that maintains acceptable increases in the l2 cache accesses at 400mV where $P_{fail} \geq 1e^{-2}$. It further demonstrates why our schemes could achieve the best performance — via effectively capturing the most likely accesses in the data cache and proper basic block placement in the instruction cache.

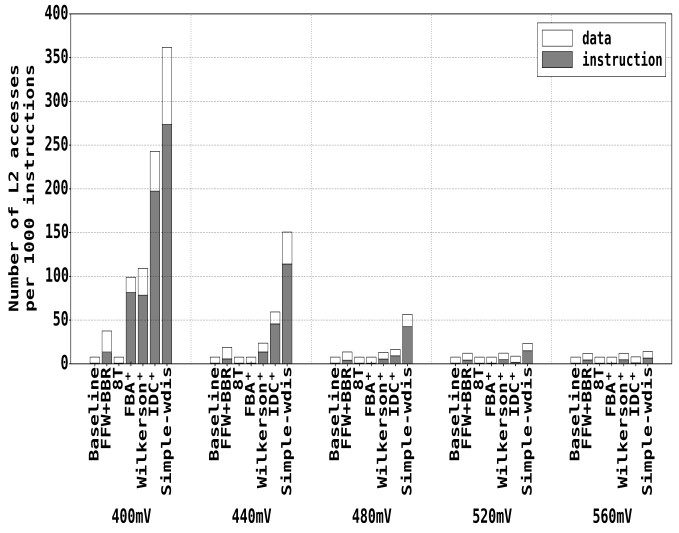


Fig. 11: Number of L2 accesses per 1000 instructions.

C. Energy reduction

In this section, we compare the processor energy consumption between our scheme and other fault tolerance approaches. We use energy per instruction (EPI) as the energy metric. The EPI results are the geometric mean of EPI for all simulations at each DVFS operating point. We normalized the EPI values to the baseline cache with a Vccmin of 760mV.

Figure 12 show the normalized EPI of different fault tolerance schemes at the interested DVFS operating points. In order to calculate the energy consumption, we assume that dynamic power scales quadratically with supply voltage and linearly with frequency. We also assume that static power scales linearly with supply voltage. We give an advantage to FBA^{+} and IDC^{+} in our energy calculation by ignoring the energy overhead of their 1024 entries.

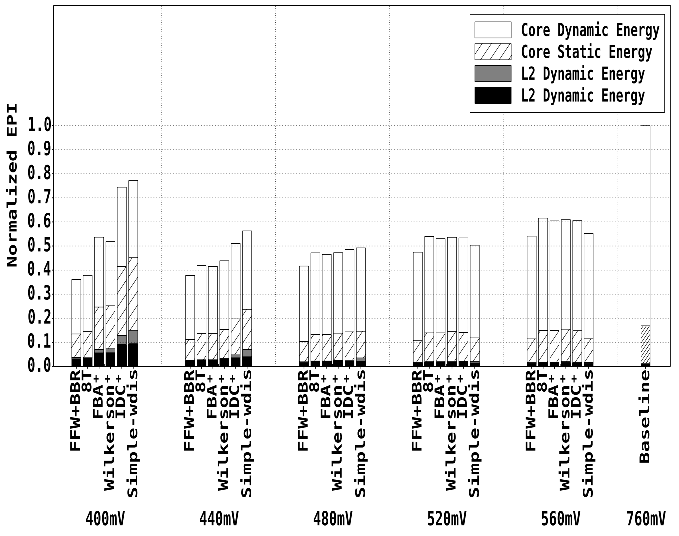


Fig. 12: Normalized overall EPI.

The FFW+BBR is the only architectural approach that achieves sustained energy reduction as voltage is scaled all the way down to 400mV. This is achieved due to the effective reduction of the additional L2 accesses and the short execution time which is due to the zero latency overhead. On one hand, fewer L2 accesses expend less dyanmic energy on the L2 cache. Also, the core wastes less energy on waiting for useful data. On the other hand, the short execution time saves the static energy for both core and L2 cache.

At 400mV ($P_{fail} = 1e^{-2}$), our schemes achieve better energy reduction than any other architectural approaches. Our L2 cache energy is very close to the 8T based cache, which means our scheme can effectively minimize the impact of defective words while tolerating the defect density as high as $1e^{-2}$. Compared to a conventional 6T based cache with a Vccmin of 760mV, our schemes reduce the EPI by 64%, which is better than the energy reduction of 8T based cache (62%). However, our energy reduction comes at significantly lower area overhead (5.2% for data cache and 1.1% for instruction cache) than the 8T based cache (28%).

VII. Conclusions

In this paper, we demonstrate that the L1 cache latency is a critical parameter that affects microprocessor performance and energy consumption at low voltage. We proposed a hardware technique and a software technique respectively for the data cache and the instruction cache. The Fault-free-window scheme minimizes the impact of SRAM failures in data caches by capturing the most likely accesses using the available fault-free words. The Basic-block-relocation scheme isolates defective words in instruction caches by managing the basic block placement. We show that the combined Fault-free window and Basic-block-relocation scheme allows reliable cache operation and achieve sustained energy reduction beyond 400mV. Compared to an conventional 6T based cache with Vccmin of 760mV, our scheme achieves 64% reduction in energy per instruction. This energy reduction comes at zero latency overhead and only 5.2% area overhead on data caches and 1.1% area overhead on instruction caches.

Acknowledgement

This work is supported by the National Science Foundation under grant CCF-1116610.

References

[1] G. Magklis, G. Semeraro, D. Albonesi, S. Dropsho, S. Dwarkadas, and M. Scott, "Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor," *Micro, IEEE*, vol. 23, no. 6, pp. 62–68, Nov 2003.

[2] T. Mahmood and S. Kim, "Realizing near-true voltage scaling in variation-sensitive l1 caches via fault buffers," in *Compilers, Architectures and Synthesis for Embedded Systems (CASES), 2011 Proceedings of the 14th International Conference on*, Oct 2011, pp. 85–94.

[3] S. Mukhopadhyay, H. Mahmoodi, and K. Roy, "Modeling of failure probability and statistical design of sram array for yield enhancement in nanoscaled cmos," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 24, no. 12, pp. 1859–1880, Dec 2005.

[4] C. Wilkerson, H. Gao, A. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, June 2008, pp. 203–214.

[5] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–. [Online]. Available: http://dl.acm.org/citation.cfm?id=956417.956570

[6] L. Chang, R. Montoye, Y. Nakamura, K. Batson, R. Eickemeyer, R. Dennard, W. Haensch, and D. Jamsek, "An 8t-sram for variability tolerance and low-voltage operation in high-performance caches," *Solid-State Circuits, IEEE Journal of*, vol. 43, no. 4, pp. 956–963, April 2008.

[7] S. Jain, K. Santhosh, M. Pattanaik, and B. Raj, "A 10-t sram cell with inbuilt charge sharing for dynamic power reduction," in *Advances in Technology and Engineering (ICATE), 2013 International Conference on*, Jan 2013, pp. 1–6.

[8] J. Kulkarni, K. Kim, and K. Roy, "A 160 mv, fully differential, robust schmitt trigger based sub-threshold sram," in *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, Aug 2007, pp. 171–176.

[9] S. Schuster, "Multiple word/bit line redundancy for semiconductor memories," *Solid-State Circuits, IEEE Journal of*, 1978.

[10] F. Bower, P. Shealy, S. Ozev, and D. Sorin, "Tolerating hard faults in microprocessor array structures," in *Dependable Systems and Networks, 2004 International Conference on*, June 2004, pp. 51–60.

[11] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, "Multi-bit error tolerant caches using two-dimensional error coding," in *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, Dec 2007, pp. 197–209.

[12] A. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu, "Energy-efficient cache design using variable-strength error-correcting codes," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, June 2011, pp. 461–471.

[13] Z. Chishti, A. Alameldeen, C. Wilkerson, W. Wu, and S.-L. Lu, "Improving cache lifetime reliability at ultra-low voltages," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 89–99.

[14] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou, "Yield-aware cache architectures," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, Dec 2006, pp. 15–25.

[15] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 1, pp. 27–38, Jan 2005.

[16] H. Lee, S. Cho, and B. Childers, "Performance of graceful degradation for cache faults," in *VLSI, 2007. ISVLSI '07. IEEE Computer Society Annual Symposium on*, March 2007, pp. 409–415.

[17] T. Mahmood and S. Kim, "Fine-grained fault tolerance for process variation-aware caches," in *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, July 2010, pp. 46–51.

[18] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "Zerehcache: Armoring cache architectures in high defect density technologies," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, Dec 2009, pp. 100–110.

[19] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Archipelago: A polymorphic cache design for enabling robust near-threshold operation," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, Feb 2011, pp. 539–550.

[20] T. Mahmood, S. Kim, and S. Hong, "Macho: A failure model-oriented adaptive cache architecture to enable near-threshold voltage scaling," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Feb 2013, pp. 532–541.

[21] A. Sasan, H. Homayoun, A. Eltawil, and F. Kurdahi, "Inquisitive defect cache: A means of combating manufacturing induced process variation," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 9, pp. 1597–1609, Sept 2011.

[22] A. Meixner and D. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, June 2008, pp. 80–89.

[23] M. Nicolaidis, "Theory of transparent bist for rams," *Computers, IEEE Transactions on*, vol. 45, no. 10, pp. 1141–1156, Oct 1996.

[24] R. Murphy and P. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 937–945, July 2007.

[25] J. Huang and D. Lilja, "Exploiting basic block value locality with block reuse," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, Jan 1999, pp. 106–114.

[26] J. E. Miller and A. Agarwal, "Software-based instruction caching for embedded processors," *SIGPLAN Not.*, vol. 41, no. 11, pp. 293–302, Oct. 2006. [Online]. Available: http://doi.acm.org/10.1145/1168918.1168894

[27] K. Dayalan, M. Ozsoy, and D. Ponomarev, "Dynamic associative caches: Reducing dynamic energy of first level caches," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, Oct 2014, pp. 118–124.

[28] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[29] C. D. Spradling, "Spec cpu2006 benchmark tools," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1241601.1241625

[30] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[31] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[32] N. Muralimanohar and R. Balasubramonian, "Cacti 6.0: A tool to model large caches." Tech. Rep: HP Laboratories, Apr 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[34] Y. B. Kim, Y.-B. Kim, F. Lombardi, and Y. J. Lee, "A low power 8t sram cell design technique for cnfet," in *SoC Design Conference, 2008. ISOCC '08. International*, vol. 01, Nov 2008, pp. I–176–I–179.

[35] S. Eggers, J. Emer, H. Leby, J. Lo, R. Stamm, and D. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *Micro, IEEE*, vol. 17, no. 5, pp. 12–19, Sep 1997.