iPatch: Intelligent Fault Patching to Improve Energy Efficiency

David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
palframan@wisc.edu, nskim3@wisc.edu, mikko@engr.wisc.edu

***Abstract*—Dynamic voltage and frequency scaling can provide substantial energy savings but is limited by SRAM since some cells will fail at very low voltages. Due to process variation effects, a small subset of SRAM cells will be more sensitive to voltage reduction, requiring increased margins and limiting energy savings. Since large arrays like caches are most vulnerable to cell failures, recent proposals suggest disabling failing portions of the cache to enable low voltage operation. Although such approaches save power, energy reduction is limited because reducing the effective cache size increases program runtimes. In this paper, we present iPatch, a solution to regain this lost performance and enable energy savings by exploiting the redundancy inherent in superscalar processors. By relying on existing microarchitectural structures and mechanisms to "patch" the faulty parts of caches, we enable further energy reduction with minimal overhead and complexity. Furthermore, because no critical paths or circuits are affected by our implementation, there is no impact on normal-voltage operation. For high cell failure rates, our results show significant energy savings with iPatch as well as an 18% reduction in energy-delay product compared to prior work.**

I. Introduction

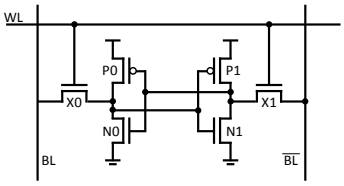
With the increasing prevalence of mobile technology, processor power and energy consumption remains an important concern. When high performance is required, techniques such as aggressive clock and power gating can be employed to reduce power. When performance is not crucial, dynamic voltage and frequency scaling (DVFS) can provide further energy savings, since power scales quadratically with supply voltage. If the voltage is reduced too much, however, storage elements such as SRAM cells may begin to fail. To avoid such failures, V_{min} must be set appropriately high, limiting potential power savings. V_{min} must be further increased due to process variations, since parameter fluctuations cause some cells to fail at higher voltages than others [1].

A number of solutions have been devised to work around this voltage scaling limitation. For instance, ARM's big.LITTLE design takes an architectural approach by incorporating both a wide out-of-order core and a smaller, lower-performing core [2]. Although V_{min} is set high enough for both cores to guarantee that no SRAM cells will malfunction, the smaller core consumes less power because it is less complex. When the DVFS mechanism can no longer reduce the voltage of the big core, work is migrated to the smaller core while power-gating the big core. Although effective, this scheme has some obvious inefficiencies due to the need for a second core and the migration overhead.

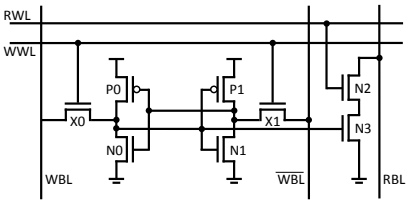
Ideally, similar savings could be achieved with a single non-heterogeneous processor if V_{min} could be further reduced while ensuring functional correctness. Without using robust SRAM cells that are less area and power efficient, such a design must be capable of correctly handling some SRAM cell failures at low voltages. For large structures that are less latency-sensitive such as L2 and L3 caches, stronger (and higher-latency) error correcting codes (ECC) can be deployed, or more complicated word-substitution approaches can be used to consolidate fault-free words into usable lines [3], [4]. A different solution is required for the L1, however, where high-latency error correcting codes or complex schemes are less feasible. Although the simplest L1 solution is to disable faulty cache lines with failing cells, the resulting increase in program execution time would nullify any potential energy savings from lowering the voltage. Recent work proposes fine-grained disabling of subblocks within each cache line, thereby increasing usable cache space [5]. Even with this mechanism, the increase in L2 accesses at aggressively low voltages degrades performance, negating potential energy savings.

In this paper, we propose exploiting the redundancy inherent in standard superscalar structures, allowing us to use existing mechanisms to occlude failing cells. For example, a block with the same address could be simultaneously held by the store queue, L1 data cache, and L2 cache. Standard superscalar mechanisms will forward the data from only one of the structures, however. Our approach, which we call iPatch, intelligently manages special *patch* entries in structures like the micro-op cache, MSHR buffers, and store queue, to proactively avoid requests to faulty L1 subblocks, since these must be forwarded to the L2. This significantly reduces performance degradation due to disabled subblocks and unlocks additional energy savings at low voltages. Because iPatch relies primarily on existing hardware and mechanisms, implementation is non-invasive and requires little departure from modern out-of-order processor designs. This paper includes the following contributions:

- Discussion of an approach to patch failing cells in the instruction cache using micro-op cache entries and MSHRs;
- Presentation of an analogous approach to patch the data cache using store queue entries and MSHRs;
- Analysis of the energy and performance benefits of iPatch compared to prior work.



(a) Standard 6T SRAM cell.



(b) Robust 8T SRAM cell.

Fig. 1: SRAM cell comparision. While less likely to fail due to voltage scaling, the 8T cell requires more area and reduces array density.

The remainder of this paper is organized as follows: Section II discusses the trade-offs of prior proposals for tolerating failing SRAM cells in caches and motivates our proposal. Section III presents the iPatch approach patching failing cells in L1 caches. Section IV details simulation methodology and results showing the energy and performance benefits of iPatch when operating at low voltages. Finally, Section V concludes the paper.

II. Background and Motivation

Modern processors employ a hierarchy of storage structures to keep the most relevant data close to the execution logic so it can be accessed quickly. These structures, including large instruction and data caches, comprise a significant and growing amount of architectural state and die area. Given the large number of storage cells and the increasing impact of process variations as technology scales, it is important to employ techniques that can tolerate or prevent failing cells to enable low voltage operation for energy and power savings.

A. Circuit-Level Solutions

Traditional 6-transistor SRAM cells like the one in Figure 1a are carefully designed to prevent unintended state changes on reads while allowing intentional writes. Due to this delicate balance, process variations can cause cells to fail in two primary ways [6]. To read the value in the cell, the bitlines (BL) are precharged. When the wordline (WL) is activated, one of the bitlines will start to discharge through an access transistor (X0 or X1). If noise on this bitline exceeds the trip point of the inverter (due to V_{th} variation), a *read failure* can occur in which the cell flips. Likewise, a *write failure* occurs if a write operation is unable to toggle the cell when a voltage differential is applied to the bitlines. This can happen if a pull-up transistor (e.g. P0) is stronger than the access transistor (e.g. X0), keeping the node from being discharged. These failures become much more prevalent as the supply voltage is lowered.

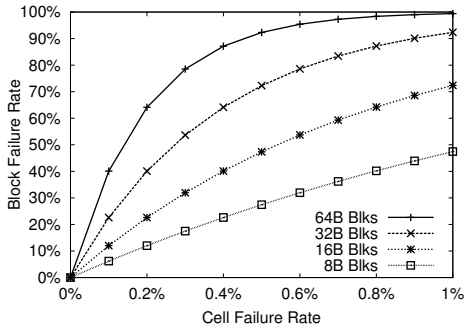


Fig. 2: Relationship between cell failure rate and block failure rate for different block sizes. A block is considered faulty when it has one or more faulty cells.

One way to address this problem is through circuit-level approaches that modify the cell design. For instance, a robust 8-transistor SRAM cell like the one in Figure 1b can be used, which adds an explicit read bitline and wordline to decouple read operations from write operations and avoid accidental read upsets [7]. Though stability is enhanced, the modified cell reduces SRAM density due to the extra devices and wires required [8]. To reduce the overhead of using robust cells, some proposals suggest hybrid cache architectures that integrate different cells, though this increases design complexity [9], [10]. For large array structures such as caches where the overhead of 8T cells is expensive, a lower overhead solution is desirable.

B. Architectural Solutions

Because process variation effects determine which cells fail at low voltages, the same cells are always unreliable for a given chip and voltage. Therefore, post-fabrication testing can be used to locate unreliable cells, enabling architectural solutions to work around the failures and guarantee correctness. The most straightforward architectural approach to dealing with faulty cells is to disable all entries in a structure that contain at least one failing cell. In the case of caches, this can mean disabling cache lines and consequentially reducing the number of ways in the affected sets, as proposed in [11]. Although this approach is effective at moderate failure rates, at lower voltages with more failures it can significantly reduce cache capacity and therefore degrade performance and energy efficiency. To better tolerate many failures, strong ECC could be used, but this is not an ideal solution for L1 caches due to increased latency and the read-modify-write operation required for partial writes.

Much prior work aims to extract high performance from partially faulty SRAM arrays by exploiting the fact that the failure rate of larger blocks of cells is much higher than that of smaller blocks. In other words, it is more likely that one or more cells will fail in a 64-byte block than a 16-byte block. Figure 2 shows the block failure rate for different block sizes and cell failure rates. To take advantage of the lower failure rate of smaller block sizes, prior work divides cache lines into smaller subblocks that can be individually disabled. The

remaining valid subblocks can be merged into fault-free lines or used in a way that enables functional correctness.

In the word-disable approach proposed by Wilkerson et al., each word in a line can be individually disabled [6]. The fault-free words from two adjacent lines are then combined to form a single functional line. This approach, while guaranteeing functional correctness, cuts the cache capacity in half. The authors also propose a similar scheme that sacrifices 25% of the cache capacity to hold pointers and values to fix individual bits. These fixed reductions in cache capacity reduce performance and therefore increase energy consumption, even at moderate failure rates. To avoid this high cache space reduction, ZerehCache and its follow-on Archipelago attempt to more efficiently combine faulty lines to form usable lines [4], [12]. The Archipelago approach divides the cache into groups called islands, each of which contains a sacrificial word-line that can be used to repair the other lines in the group.

If too much of the L1 cache is disabled, performance can be degraded. To compensate for this effect, some approaches suggest adding cache assist structures such as buffers or victim caches [11], [13]–[15]. In the RVC approach, cache lines are disabled but replaced with victim cache entries in order to provide performance guarantees in a real-time system [15]. The authors also suggest an alternative in which existing buffers are enlarged, allowing the extra entries to be used as replacements for faulty lines. Although this idea bears some similarity to one of our iPatch mechanisms, iPatch does not require adding buffers or enlarging existing buffers and can support higher failure rates.

The subblock-disable approach is a simple solution that disables bad subblocks without the complexity of merging good subblocks into full lines [5]. Since this approach can degrade performance if many subblocks are disabled, prior work suggests reordering data subblocks before they are written to the cache such that more useful data will not map to disabled subblocks [16]. The authors suggest a hybrid approach using subblock reordering, a fault-free fill buffer, victim caches, and modified prefetching to reduce the performance impact of false hits. These changes add complexity and overhead to the subblock-disable approach, motivating our proposal of iPatch as a simpler and effective alternative.

C. Subblock Disabling

This section discusses the subblock-disable technique in greater detail, since we use it as our baseline [5]. In this technique, the disabled subblocks within a line simply become inaccessible, so a valid copy of this data must be kept in the L2. A fault map containing a disable bit per L1 subblock is required to track whether or not each subblock is disabled (due to unreliable cells) at the current operating voltage. Post-fabrication testing can determine which cells will fail due to process variation effects at each voltage. After a voltage change, the fault map for each L1 line can be updated accordingly.

In subblock-disable, read and write operations are handled normally as long as they access the non-disabled portions of each cache line. Accesses to disabled subblocks are treated similarly to misses and must access the L2, since these requests cannot be serviced by the L1. These accesses are called *false hits*, since there is a tag match in the L1 (just like a hit) but the data must be retrieved from the L2. Since too many disabled subblocks in a hot cache set could produce performance outliers, we implement the performance predictability mechanism from [5] in which the L1 address-to-set mappings are periodically changed (after flushing the L1). This approach significantly reduces the performance variation across chips, with the flushing/remapping happening infrequently enough that performance is not affected.

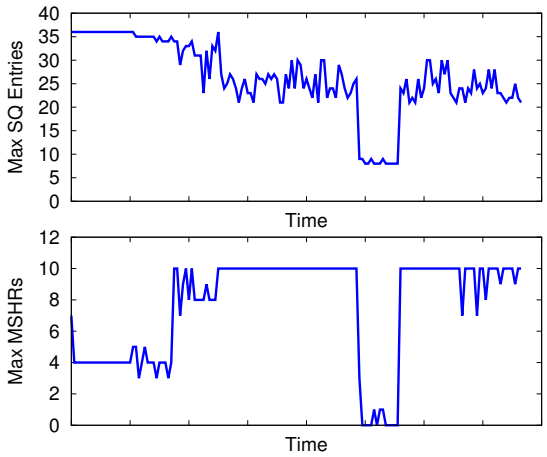


Fig. 3: Maximum store queue and data cache MSHR usage over time when executing bzip2.

One of the main advantages of the subblock-disable approach is its simplicity. It incurs low overhead while making the maximum amount of cache space available, while many of the substitution-based approaches may have "wasted" storage. In addition, modification of the critical L1 datapath is minimal, meaning that there will be few complications in high-performance mode with no failing cells thanks to a higher supply voltage. At lower voltages with higher cell failure rates, however, the increase in the number of disabled subblocks can significantly degrade performance due to additional false hits. This effect imposes a limit on subblock-disable's energy-saving potential, since despite the lower voltage, execution time will increase. With simplicity in mind and to enable a wide range of operating voltages, we show that when building on the subblock-disable approach, our iPatch solution is able to significantly reduce subblock-disable's false hit rate at low voltages by using hardware already common in modern superscalar designs. By improving performance in this manner, we are able to extend energy savings to lower voltages.

III. Fault Occlusion with iPatch

Modern processors rely on a hierarchy of caches and buffers to significantly reduce the data load-to-use latency and ensure functional correctness. Such structures include data and instruction caches, store queues, fill buffers, and micro-op caches. In this work, we observe that data redundancy

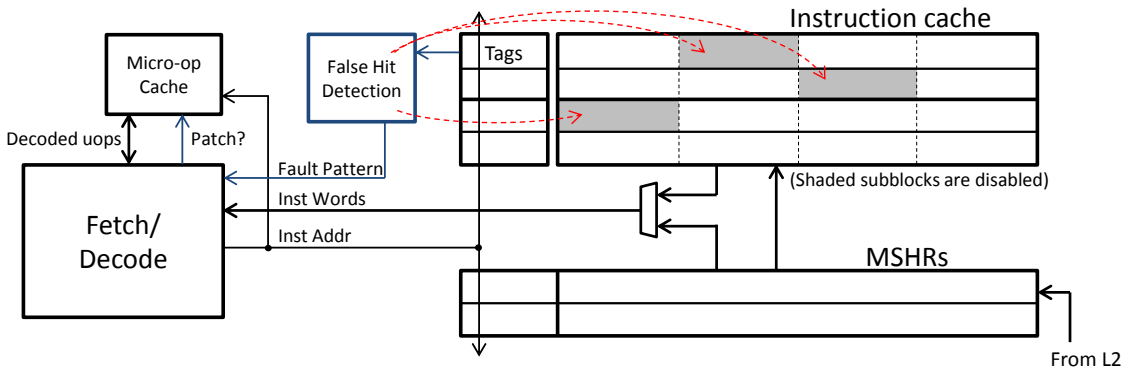


Fig. 4: Processor front-end with fault patching additions. In order of preference, instructions are read in decoded form from the micro-op cache or undecoded from the L1 MSHRs or L1 instruction cache. The false hit logic tracks faulty sections of the cache and forces a miss (false hit) if the front-end tries to access these subblocks.

naturally exists across these units. For example, a block with the same address could be simultaneously held by the store queue, data cache, and L2 cache, though some copies might be dirty. Even if the data is identical across all of these elements, a load need only be serviced by one of them. Consider a case in which there is a single failing bit cell in the data cache. If a load to the associated address is forwarded from the store queue, the faulty cell will not be read and correctness is guaranteed. A similar scenario can occur when the processor front-end reads an instruction from a small micro-op cache instead of accessing the L1 instruction cache. The iPatch approach proposed in this work intelligently exploits this patching mechanism whereby a fault-free unit services the request, thereby occluding an L1 cache fault from the processor core. Buffers such as the store queue (SQ) and miss-status handling registers (MSHRs) that can produce this patching effect are small and therefore inexpensive to protect with robust cells that are less likely to fail at low voltages. With a mechanism to compensate for failing cache cells, large cache arrays can be designed with dense 6T cells while enabling low voltage operation.

For the remainder of this paper, we make a distinction between *patch* and *non-patch* entries in various buffers and structures. A *patch* entry is one in which the subblock in the cache that corresponds to the data address is disabled. A *non-patch* entry's data, on the other hand, is valid in the L1 and can be read without triggering a false hit. Due to the benefits of *patch* entries over *non-patch* entries, iPatch attempts to create and promote *patch* entries when possible. One additional observation inspires our solution. We note that most buffers that can hold *patch* entries are not fully utilized all of the time. As an example of fluctuating resource use over time, Figure 3 shows the maximum number of allocated store queue entries and MSHRs while executing bzip2. During the phases in which resources usage is lower, iPatch can repurpose the unused entries as patches.

The remainder of this section discusses each of the patching mechanisms in detail. We begin by discussing how to patch the instruction cache, since instruction fetch latency is critical to performance. In this case, patching is accomplished using micro-op cache entries and MSHR entries. Similarly, we can patch the d-cache with SQ entries as well as MSHRs. Finally, we discuss how to best combine these approaches.

A. Patching with decoded micro-ops

Many modern processors employ micro-op caches or buffers that are very effective at saving fetch and decode power through clock gating. As shown in Figure 4, decoded micro-ops will be read from the micro-op cache if they are present. On a micro-op cache miss, instructions are read from the i-cache or its associated MSHR buffers. After they are decoded, the instructions are saved to the micro-op cache for later use. For our purposes, if an instruction fetch hits in the micro-op cache, the micro-op cache entry may act as a patch for the instruction cache, since the instruction cache will not be accessed. Since not all parts of the i-cache will always be patched, the false hit detection logic shown in the figure keeps a fault map (bit vector) to track the disabled subblocks in each line and forces a miss if the front-end tries to access one, just as in the subblock-disable approach.

In the iPatch implementation, when the fetch stage reads instructions from the i-cache, the i-cache also provides a fault pattern of the disabled subblocks within the block of instructions read. If the size of the instruction block requested is less than or equal to the i-cache subblock size, the fault pattern is simply a single bit indicating whether the data being provided is mapped to a disabled subblock. If this is the case, the instruction data will have been read from the L2 or an i-cache MSHR buffer. In either case, the cache management logic determines the current or destination way in the i-cache and is able to forward the associated fault pattern. Using this fault pattern information, the decoder can track which micro-ops are derived from disabled i-cache subblocks. When these micro-ops are written into the micro-op cache, the destination entry is then designated as a *patch* entry using an extra per-entry *patch* bit.

With the extra bit to indicate which micro-op cache entries are patches, we can modify the replacement policy to favor keeping these entries. This modified policy must be carefully chosen, since the micro-op cache can provide both energy and

performance benefits. If we only allowed *patch* entries in the micro-op cache, for instance, both of these areas could be negatively affected. As the performance penalty of false hits (and, by proxy, energy) rises at higher cell failure rates, the importance of retaining patches increases. Thus, our approach attempts to find a balance between the number of *patch* entries and *non-patch* entries. We accomplish this using a patch threshold parameter that indicates the target percentage of *patch* entries in each μop cache set. If the number of patch entries falls below the threshold, the replacement decision favors replacing a non-patch entry. If the number of patch entries exceeds the threshold, the replacement decision (e.g. LRU) is made without any modification. The patch threshold can be tuned for optimal performance at each voltage point and its associated cell failure rate.

B. Patching with MSHRs

Each cache is equipped with a number of miss status handling registers (MSHRs), as depicted in Figure 4. MSHRs are used to track outstanding misses that must be serviced from a lower-level cache or memory. They allow non-blocking memory requests by storing the information needed to continue the operation once the data is available. Each MSHR has an associated fill buffer entry to hold the data before it is written into the cache. Since cache line data may not be furnished by the lower-level cache all at once, each MSHR contains valid bits to track which subblocks are currently valid in the associated fill buffer. Once all subblocks are valid, the line is written into the cache from the buffer and the MSHR is freed. For best performance, MSHRs are able to service loads from partially accumulated cache blocks or blocks that have not yet been written to the cache. A load will check its address against the block address stored by the MSHR to see whether the data required is currently valid in the buffer. On a match, the load can be serviced directly from the fill buffer. Otherwise, the load misses, allocating a new MSHR or adding itself to an existing MSHR.

iPatch takes advantage of the ability to service loads from MSHRs in order to use MSHRs as patches for faulty lines in the cache. Unlike the μop cache implementation in which a *patch* entry patches only part of a cache line, an MSHR can be used to patch an entire cache line. This approach is highly efficient, therefore, in the case of cache lines with multiple disabled subblocks.

To allow intelligent patch management decisions, iPatch augments each MSHR with a patch bit and a reference bit, as shown in Figure 5. The patch bit indicates whether or not each entry is a patch. The reference bit is set when a load is serviced from the MSHR, and allows iPatch to find a not-recently-used patch entry to invalidate in case a new MSHR is needed but the buffer is full. The reference bits of all patch entries are reset if all patches have been referenced.

To use MSHRs as patches, we simply keep the entries and data valid after a cache miss/fill is complete. On a cache miss, an MSHR is allocated to track the request status. Once the entire cache line has been accumulated by the MSHR, it is written into the cache, as usual. Instead of being freed, the MSHR entry is retained and its patch bit is set to denote that it is not actively tracking an outstanding request. As depicted in Figure 5, a copy of the same block is now present in both the MSHR and the cache. In this scenario, iPatch requires that a load hit in an MSHR will take precedence over a cache hit so that as many loads as possible will be serviced from MSHR buffers, avoiding disabled cache subblocks.

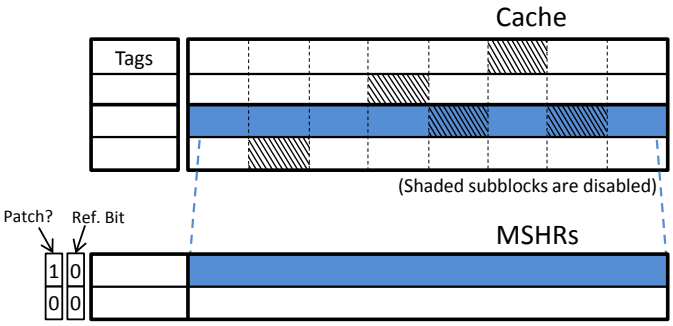


Fig. 5: MSHR patching illustration. For lines with many disabled blocks, a duplicate fault-free copy can be retained in an MSHR to service loads.

Because MSHRs are retained after they would normally be deallocated, with the iPatch approach, more MSHRs are in use at any given time. We carefully select our patch management policy to guarantee that it will not reduce performance or cause deadlock. Because the data from patch entries has already been written into the cache, a patch entry can simply be invalidated if all MSHRs are in use and the cache needs to allocate an MSHR to handle a new miss. In this scenario, the MSHR patch to overwrite is selected using the previously-mentioned reference bits. Once the patch MSHR is overwritten, the copy of the line in the cache is now unpatched, exposing its missing subblocks to the execution core. By taking this approach of keeping entries longer than usual but immediately invalidating them as necessary, iPatch does not reduce the number of MSHRs available to handle misses compared to a system without iPatch. Since the cache never blocks due to patch entries, performance is not reduced. We note that the MSHR patching approach can be applied to any cache, including L1 data and instruction caches, as well as other levels. Furthermore, no performance penalty is incurred in L1 caches, since data can be supplied from the fill buffers with the same latency as the cache.

We also invalidate patch entries when a line with an MSHR patch is written to. Because the L1 cannot write directly into its own MSHRs without an additional write port, the fill buffer data no longer matches the cache after a write, so the patch must be invalidated for correctness. A more aggressive incarnation could also reset the MSHR subblock valid bits depending on which section of the line was written (although we did not implement this). In either case, this policy does not impact the i-cache, since there are no writes to cause this invalidation.

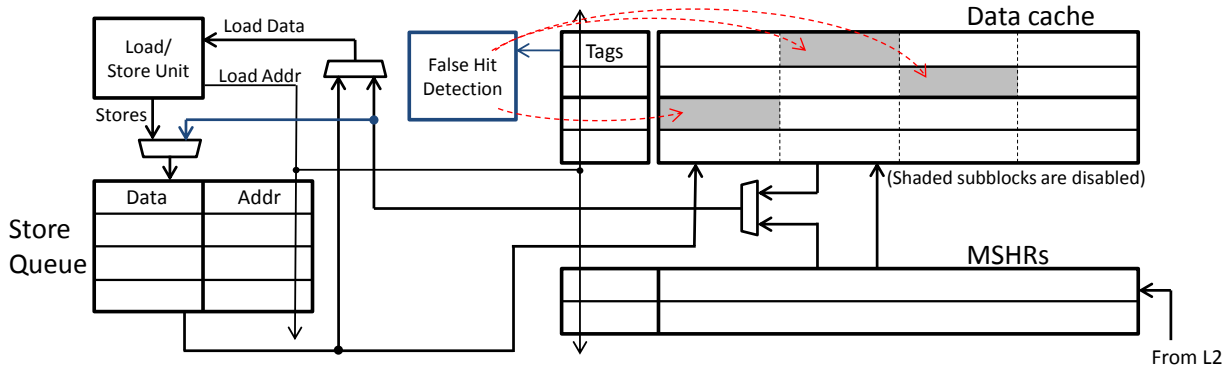


Fig. 6: Load/store hardware with fault patching additions. On an address match, loads are serviced from (in order of preference) either the store queue, the L1 MSHRs, or the L1 data cache. The false hit logic tracks faulty sections of the cache and forces a miss (false hit) if a load tries to access these subblocks.

C. Patching with SQ entries

The load and store queues in an out-of-order processor allow memory instructions to be executed out of order while maintaining program-order dependencies. They are also responsible for managing and squashing speculative memory accesses, if necessary. The store queue holds pending stores that have not yet been written to the data cache. Each load instruction is assigned a store color corresponding to the most recent store instruction (in program order). Using its store color, a load checks the store queue for older stores to the same address. If a match is found, data is forwarded from the matching SQ entry.

Just as we can use μop cache entries to patch the i-cache, we can use store queue entries to patch the d-cache by exploiting the store-to-load forwarding mechanism. Figure 7 demonstrates how a store queue entry can act as a patch for faults in the data cache. The highlighted data exists in both the data cache and the store queue due to a store that has not yet completed. As shown, the cache copy overlaps with a faulty subblock and cannot not be reliably read from the cache without an L2 access. Since loads to that address will be forwarded from the store queue, however, the disabled L1 subblock is no longer a concern.

A store is considered to be completed after its data is written to the d-cache. In most designs, these completed stores are then removed from the SQ. For iPatch, we modify this behavior and keep some completed stores in the queue. Normally, there would not be much utility in keeping such entries after they are written back due to the equivalent latency for store-to-load forwarding and data cache accesses, since the structures are searched in parallel. For iPatch, however, keeping completed stores provides considerable benefit if these entries are patches. Furthermore, allowing completed SQ entries does not degrade performance or cause deadlock, since if a new SQ entry must be allocated, a completed entry can immediately be invalidated. Because allowing completed store queue entries implies having multiple copies of the same data, the store queue is kept coherent with the rest of the cache hierarchy.

By allowing completed stores in the store queue, some parts of the data cache will be patched as a side effect. Relying

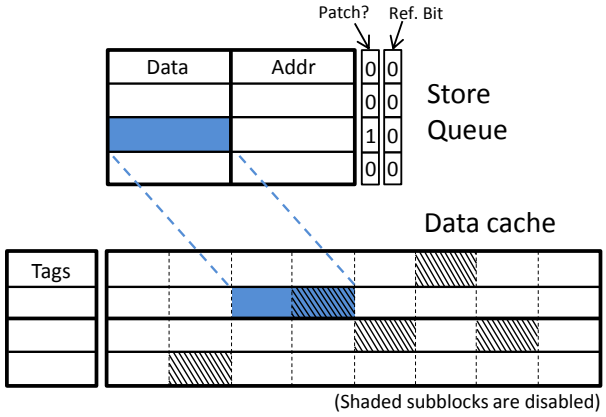


Fig. 7: Illustration of using a store queue entry to patch a faulty cache subblock. Loads to the highlighted block are serviced from the store queue and not the partially faulty copy in the cache.

solely on this natural patching, however, does not provide maximum benefit. Depending on the SQ implementation, iPatch can be more proactive about keeping and creating patch entries. Although the store queue is traditionally implemented as a circular buffer, prior work has developed practical approaches to managing SQ entries in an unordered fashion [17], [18]. Unordered store queue management allows late entry allocation, reducing the pressure on the queue and enabling larger instruction windows. This technique also benefits weakly-ordered ISAs (our evaluation uses ARM) by allowing a unified SQ from which completed stores to non-overlapping addresses can be removed out of order. For optimal efficiency, iPatch can benefit from unordered SQ management by removing completed non-patch entries out of order while allowing patch entries to be persistent. An alternative to a unified store queue that still benefits from out-of-order store write-back could combine a circular store buffer with an additional unordered committed store buffer. In this implementation, iPatch can still perform aggressive d-cache patching by managing completed entries in the committed store buffer.

With an unordered store queue, we can aggressively insert patches into the store queue as lines are loaded into the L1

cache. To enable this mechanism, we add an additional path allowing the data cache MSHRs to write directly into the store queue, as depicted in Figure 6. Cache replacements are performed in the L1 as usual, except that patches are written to the SQ from the fill buffer before the associated MSHR is freed. Once the destination way is determined, the fault pattern stored by the false hit logic is used to determine which SQ entry-sized segments of the line must be inserted into the SQ as patches.

When patch entries are inserted into the store queue, they are marked as completed entries, which are, by definition, the oldest "stores" relative to others in the queue. We add two extra metadata bits to each store queue entry to facilitate resource management. One of the extra bits is a "patch" bit, which indicates that the entry is both a patch and has completed. The second extra bit is a reference bit, which is set when the entry forwards its data to a load. We use these two bits to determine which entry to free (through invalidation) when a new SQ entry needs to be allocated. In order of decreasing preference, we prefer to free untouched/non-patch entries first, followed by touched/non-patch, untouched/patch, and finally touched/patch entries. If all patch entries have their reference bits set when searching for a replacement candidate, all reference bits are reset. Finally, if an older entry was a patch (according to its "patch" bit) a newly-completed store to an overlapping address inherits this status before the older entry removed. If no such overlap exists when a store completes, we cannot know for sure if it is a patch without checking the d-cache. In this case, we avoid this check by simply assuming that the entry is not a patch and removing it from the queue.

D. Putting it all together

We have presented two iPatch techniques that can be combined in both the d-cache and the i-cache. In the i-cache, μop cache patching can be combined with MSHR patching. Likewise, in the d-cache, SQ patching can also be combined with MSHR patching.

One possibility when combining the two mechanisms is to partition the lines in the cache such that a subset are patched with only one mechanism (e.g. μop cache patching) and the remainder are patched only with the other mechanism (e.g. MSHR patching). In our experiments, however, we were unable to find a partitioned configuration that performed significantly better than one in which we applied both mechanisms to all lines. To combine approaches in the i-cache, lines that are MSHR-patched upon insertion still send their fault pattern to the front-end for tagging in the μop cache, even though the data is read from the fault-free MSHR. On the d-cache side, we adopt the same approach when adding patches to the store queue. iPatch provides the maximum benefit when both mechanisms are implemented in the i-cache and d-cache.

IV. Evaluation

A. Fault model

iPatch provides performance and energy benefits when L1 SRAM cells fail at very low supply voltages, requiring portions of the L1 to be disabled. To quantify the benefits of iPatch, we simulate a number of processors with different faulty cell locations according to the following fault model, which is similar to the methodology used by previous work [5], [11]. Process variations randomly cause the threshold voltage to vary across devices. Due to this effect, for a given chip, certain cells will be predisposed to fail at low voltages. Furthermore, for each chip, the same cells will be unreliable at each low voltage point, with more cells failing at lower voltages. These unreliable cells can be located through post-fabrication testing.

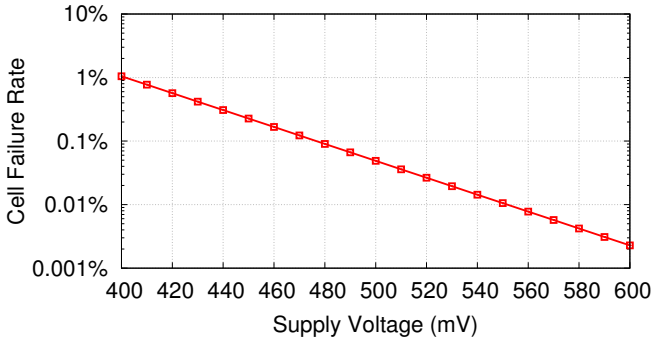


Fig. 8: SRAM cell failure rate as a function of voltage for 32nm technology [19].

To simulate a given operating voltage, we first determined the SRAM failure rate for that voltage using data from [19], as shown in Figure 8. We then performed Monte Carlo simulations in which the failing L1 cells were randomly chosen based on the cell failure rate for the voltage. For each fabricated chip (and voltage) simulated, a random number was generated for every L1 SRAM cell and compared to a threshold to determine if the cell should be modeled as faulty. The faulty cells chosen in this manner were considered to remain unreliable for all benchmarks run on each simulated chip at a given voltage. For functional correctness, our baseline (subblock-disable) and iPatch disable all L1 subblocks containing one or more unreliable cells. Because performance will vary somewhat depending on fault locations, a number of fabricated chips are simulated for each voltage/configuration by using different random seeds to select different faulty cells. We implement the L1 address remapping mechanism discussed in Section II-C to significantly improve the performance predictability of the chips simulated. We assume that other non-cache SRAM elements like tags and buffers (e.g. SQ and MSHRs) are implemented with more robust cells and are therefore fault free. Because these structures are much smaller than caches, switching to more robust 8T cells is relatively inexpensive. In addition, multiported structures like the register file already implement 8T cells to enable the extra ports.

B. Simulation infrastructure

To evaluate the performance and energy benefits of iPatch, we used the gem5 simulator in conjunction with McPAT [20], [21]. With gem5, we modeled a future high-end ARM processor, as detailed in Table I. We carefully sized all relevant buffer structures (i.e. MSHRs and the store queue) to be

TABLE I: Simulator configuration

Category	Configuration
OoO Core	Fetch/commit: 4-wide Issue: 5-wide Reorder buffer: 192 entries Instruction queue: 54 entries Physical registers: 160 INT/144 FP Load queue: 64 entries Store queue: 36 entries
Execution Units	Integer ALUs: 3 (1 cycle, 3 cycle multiply) Memory: 2 (1 cycle AGU) FP adder/multiplier: 2 (5 cycles) FP div/square-root: 1 (10 cycles)
Memory/ Caches	μop Cache: 32 set/8-way L1 instr: 32 KB/8-way/4 MSHR, 3 cycles L1 data: 32 KB/8-way/10 MSHR, 3 cycles L2: 256 KB/8-way, 12 cycles L3: 4 MB/16-way, 30 cycles Memory latency: 30 ns

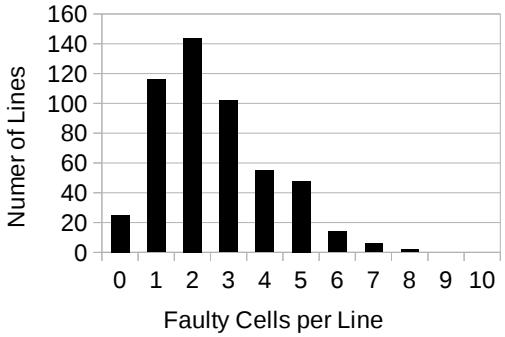


Fig. 9: Distribution of faulty cells across lines for a 32KB cache with 64B lines and a 0.5% cell failure rate.

consistent with modern high-performance architectures such as Intel's Sandy Bridge. A sensitivity study showed nontrivial performance impact from downsizing these buffers. Since current ARM processors employ a decoded loop buffer to save front-end power, we assume that future generations will upgrade this loop buffer to a μop cache, as implemented in many other high end designs from other vendors. We find that our μop cache implementation achieves an 80% hit rate on average. We also modified the store queue implementation to model an unordered store queue that can complete and remove stores out of order, thereby taking advantage of the ISA's weak memory ordering. We set the store-to-load forwarding latency to match the L1 hit latency.

In addition, we modified the simulator to model a write-through L1 cache. This configuration is useful in energy-conscious designs, since it allows the core to quickly enter a low-power state without flushing dirty data from the L1. Likewise, when we update the L1 address mappings for the performance predictability mechanism, we do not have to write back dirty L1 data. We invalidate the L1 every 500,000 cycles so data can be remapped to different L1 sets using a simple hash. This is infrequent enough to have negligible performance impact. We also find that for the benchmarks studied, using a write-through L1 has minimal performance impact when compared to a write-back cache. Note that iPatch would require little modification to work with a write-back cache, as the subblock-disable implementation described in [5] uses a write-back L1.

In our simulations, we modeled 8 subblocks per cache line that can be individually disabled. No special false hit management is required for writes, since all writes are sent to the L2 by default. A read, however, can trigger a false hit if it attempts to read data from a disabled subblock, requiring data to be fetched from the L2. When the data is returned from the L2, the line is invalidated in the L1 and written to a new way selected by the replacement policy, as proposed in [5]. Relocating the data to a line with a different fault pattern reduces successive false hits due to repeated accesses.

To simulate voltage and frequency scaling, we generated $f(V)$, a frequency scaling factor, by measuring frequency of a 24-stage FO4 inverter chain across a range of V values while simulating 32nm devices in HSPICE. We assumed a nominal frequency of 3.3GHz for our simulated processor and scaled this frequency using the trend observed in the HSPICE experiment to obtain a voltage/frequency curve for DVFS. McPAT was used to compute power and energy consumption [21]. The execution core was configured to use 32nm low operating power technology, while the L2 and L3 were set to use low static power devices. To simulate DVFS, power was first computed for each trial with McPAT configured for nominal voltage and frequency. Dynamic power was scaled down from the values reported by McPAT using the previously computed $f(V)$ scaling factor. We used a dummy circuit modeled in HSPICE to more accurately compute a $l(V)$, a leakage scaling factor, than is possible with McPAT alone. The dummy circuit for leakage current modeling consists of a large number gates (INV: 50%, NAND: 30% and NOR: 20% effective widths) where randomly selected input states are applied to each gate with 1-4 inputs to measure the leakage power, as in [22].

We simulated a representative set of integer and floating-point SPEC2006 benchmarks [23]. The SimPoint tool was used to select a section of 100 million instructions from each benchmark when using the train input set [24]. For various operating voltages, we simulated the subblock-disable scheme as a baseline as well as various combinations of the iPatch techniques. For the μop cache patching scheme, the patch threshold was tuned for each operating voltage. For each configuration and voltage, we simulated a total of 50 chips, with each chip having different faulty cell locations. For all of our results, we report the mean performance and energy across these 50 chips. Thanks to the performance predictability mechanism, we find that the true mean lies within $\pm$2% of the reported mean in the worst case.

C. Results

To evaluate the performance benefits of each iPatch technique, we simulated a relatively high cell failure rate of 0.5%. Figure 9 is example data from one of our Monte Carlo experiments showing the number of failing cells per line for a 32KB cache with 64-byte lines. At this failure rate,

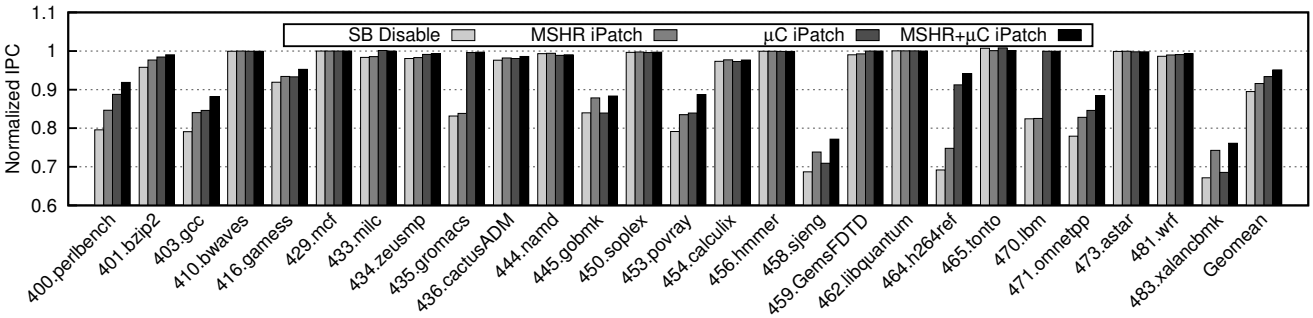


Fig. 10: Performance benefits of different iPatch techniques with a 0.5% cell failure rate in the L1 instruction cache. IPC is normalized to the fault-free case.

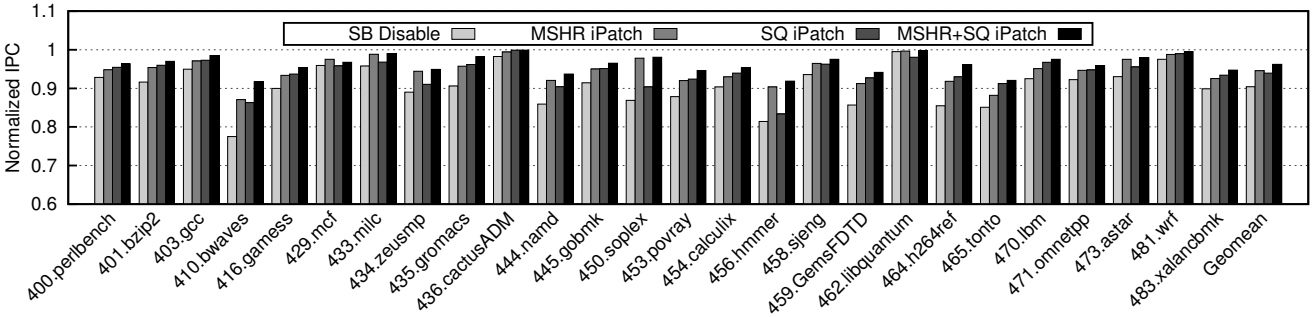


Fig. 11: Performance benefits of different iPatch techniques with a 0.5% cell failure rate in the L1 data cache. IPC is normalized to the fault-free case.

enough subblocks must be disabled that performance using the subblock-disable approach is significantly degraded.

Figure 10 shows the performance of the subblock-disable scheme and various iPatch configurations when a 0.5% failure rate is simulated in the i-cache. All results shown are normalized to the performance of the ideal case with no failing cells. As shown, false hits reduce the performance of the subblock-disable scheme by 11% percent on average. Note that this performance degradation is significantly lower than what it would be without the filtering provided by the μop cache. A number of the benchmarks shown have only negligible performance degradation with subblock-disable due to their high μop cache hit rates, with performance degradation for the others around 20-30%. Many of these fault-sensitive benchmarks derive significant benefit from micro-op cache (μC) patching. On its own, MSHR patching provides less benefit than μC patching, since the i-cache does not have many MSHRs. As shown, the combination of approaches reduces the performance degradation to under 5% on average.

Figure 11 shows the analogous results for the d-cache. Unlike the μop cache on the instruction side, the store queue provides very little "built-in" patching for the d-cache when simulating the subblock-disable scheme. This is because it is a smaller structure and completed stores are removed by default. As in the case of the i-cache, some benchmarks like GemsFDTD and h264ref perform well with SQ patching alone, while others like hmmer and soplex prefer the MSHR approach. In all cases, the combined iPatch approach performs best, though the benefit is not additive.

Figure 12 shows the results of simulating failing cells in both the i-cache and d-cache. We compare the performance of the subblock-disable scheme with a configuration that combines all iPatch techniques. As shown, iPatch improves performance over the subblock-disable approach by 11% on average.

Figure 13 shows the energy of subblock-disable compared to iPatch (all techniques) when failing cells are simulated in both the i-cache and d-cache. Results for two different operating voltages (and cell failure rates) are shown. At the lower cell failure rate, subblock-disable performs well, and iPatch is not needed, as shown. As the voltage is reduced and the cell failure rate increases, the desired outcome is a reduction in both power and energy. Power is reduced, but as shown, energy consumption actually increases when using subblock-disable due to the performance degradation caused by false hits. When iPatch is enabled, however, this performance degradation is significantly reduced, enabling energy savings despite the higher cell failure rate.

Figure 14 details the voltage vs. energy curves for selected benchmarks. The two curves for each application show the energy consumption with subblock-disable and with iPatch. As shown, iPatch extends the energy curve to allow continued savings at lower voltages. The subblock-disable curve, however, turns upwards due to the extra energy consumption incurred by longer execution times. Finally, Figure 15 shows the energy-delay product for iPatch with a 0.5% cell failure rate. Since iPatch provides both lower energy and execution time than subblock-disable, EDP is reduced by 18% on average.

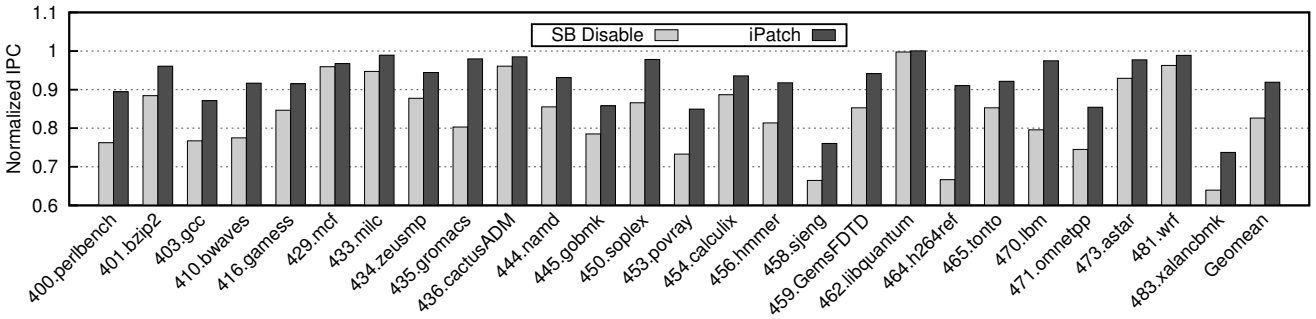


Fig. 12: Performance comparison of the subblock-disable approach and iPatch when simulating a 0.5% cell failure rate in the i-cache and d-cache.

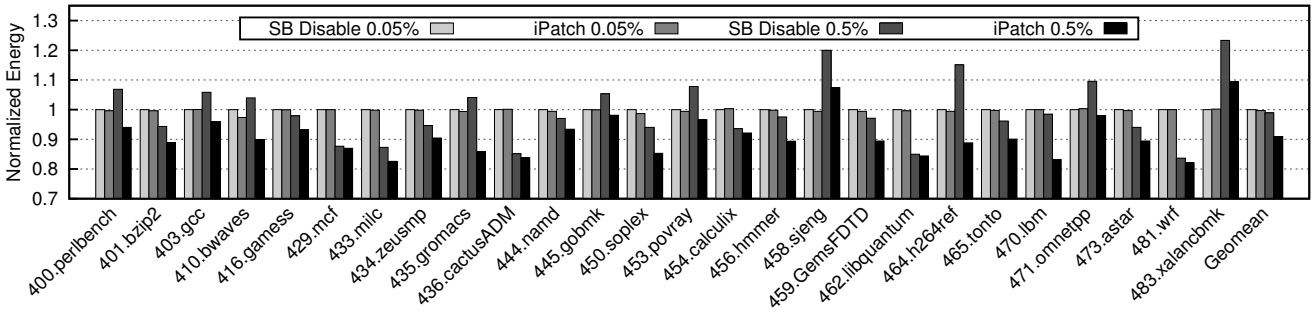


Fig. 13: Energy consumption with subblock-disable and iPatch at different operating points. Subblock-disable performs well with a 0.05% failure rate, but energy is not saved at the lower-voltage 0.5% point without iPatch.

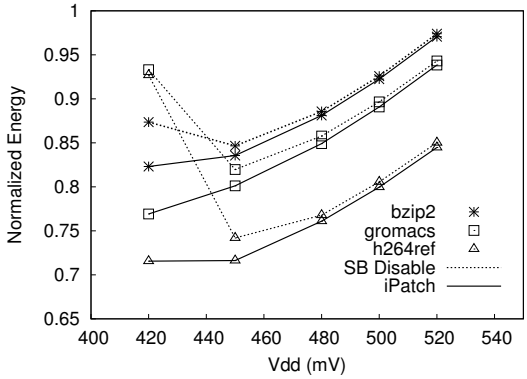


Fig. 14: Energy scaling curves for three benchmarks comparing subblock-disable and iPatch. iPatch allows further scaling.

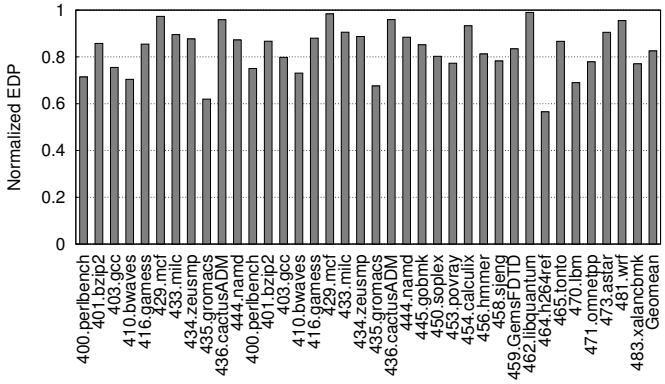


Fig. 15: Energy-delay product for iPatch normalized to the subblock-disable approach for a 0.5% failure rate. On average, iPatch reduces EDP by 18%.

One additional benefit of iPatch is a reduction in performance variation across chips. Table II shows the performance variation and margin of error across all simulated chips for each configuration at a 0.5% failure rate. To represent the performance variation, we show the 3σ value as a percentage deviation from the reported mean, where σ is the sample standard deviation. For a normal distribution, this 3σ range comprises 99.7% of all values. We also include the margin of error for the 95% confidence interval. For all configurations and benchmarks, we therefore have 95% confidence that the true population mean is within $\pm 2\%$ of the reported mean. Despite the reduction in performance deviation provided by the L1 address remapping mechanism, some benchmarks have higher performance variation across chips, with gromacs having the highest at 16.5%. iPatch is able to reduce the performance variation across chips by reducing the number of false hits, since false hits are the cause of the variation. As shown, with iPatch, the variation range for gromacs drops to 1.6%, which is a 90% improvement.

V. Conclusion

Voltage scaling to save power and energy in modern processors is mainly limited by SRAM cells that fail at lower voltages. Existing proposals to guarantee correctness at low voltages can degrade performance, negating any energy savings. In this paper we propose iPatch, a noninvasive and low overhead technique to mitigate the performance impact of subblock disabling and thereby unlock additional energy savings. The technique works by exploiting the natural information redundancy between L1 caches and structures like MSHRs, store queues, and micro-op caches to avoid expensive

L2 accesses that hurt performance and energy. By relying on existing components in the processor's memory datapath, iPatch requires no changes to performance- or latency-critical structures or circuits. Instead, modifications to control and replacement policies that are implemented off the critical path are used to place memory subblocks in these existing structures in order to occlude faulty blocks in the instruction and data caches. Results show that iPatch enables energy savings at high cell failure rates as well as an 18% average reduction in EDP compared to prior work when 0.5% of SRAM cells are failing.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants (CCF-1116450, CCF-1318298, and CCF-1016262), DARPA under grant (HR0011-12-2-0019), and a gift from Qualcomm Research. Nam Sung Kim has a financial interest in AMD and Samsung Electronics.

REFERENCES

[1] A. Bhavnagarwala, X. Tang, and J. Meindl, "The impact of intrinsic device fluctuations on CMOS SRAM cell stability," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 4, pp. 658–665, Apr 2001.

[2] P. Greenhalgh, "big.LITTLE processing with ARM Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.

[3] 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 *Proc. 38th Annual Int. Symp. on Computer Architecture*, June 2011, pp. 461–471.

[4] A. Ansari, S. Feng, S. Gupta, and S. Mahlke, "Archipelago: A polymorphic cache design for enabling robust near-threshold operation," in *Proc. 17th Int. Symp. on High Performance Computer Architecture*, Feb 2011, pp. 539–550.

[5] J. Abella, J. Carretero, P. Chaparro, X. Vera, and A. Gonzalez, "Low Vcc-min fault-tolerant cache with highly predictable performance," in *Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009, pp. 111–121.

[6] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S.-L. Lu, "Trading off cache capacity for reliability to enable low voltage operation," in *Proc. 35th Annual Int. Symp. on Computer Architecture*, 2008, pp. 203–214.

[7] 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," *IEEE Jounal of Solid-State Circuits*, vol. 43, no. 4, pp. 956–963, 2008.

[8] G. Chen, D. Blaauw, T. Mudge, D. Sylvester, and N. S. Kim, "Yield-driven near-threshold SRAM design," in *Proc. Int. Conf. on Computer-Aided Design*, Nov 2007, pp. 660–666.

[9] B. Maric, J. Abella, and M. Valero, "ADAM: An efficient data management mechanism for hybrid high and ultra-low voltage operation caches," in *Proc. Great Lakes Symp. on VLSI*, 2012, pp. 245–250.

[10] ——, "Efficient cache architectures for reliable hybrid voltage operation using EDC codes," in *Proc. Conf. on Design, Automation and Test in Europe*, 2013, pp. 917–920.

[11] N. Ladas, Y. Sazeides, and V. Desmet, "Performance-effective operation below vcc-min," in *Proc. 2010 IEEE Int. Symp. on Performance Analysis of Systems Software*, March 2010, pp. 223–234.

[12] A. Ansari, S. Gupta, S. Feng, and S. Mahlke, "ZerehCache: Armoring cache architectures in high defect density technologies," in *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, Dec 2009, pp. 100–110.

[13] B. Maric, J. Abella, and M. Valero, "APPLE: Adaptive performance-predictable low-energy caches for reliable hybrid voltage operation," in *Proc. 50th Annual Design Automation Conference*, 2013, pp. 84:1–84:8.

[14] T. Mahmood and S. Kim, "Realizing near-true voltage scaling in variation-sensitive L1 caches via fault buffers," in *Proc. 14th Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, Oct 2011, pp. 85–94.

TABLE II: Performance Variation and Margin of Error

	SB-Disable		iPatch	
	3σ (%)	ME (%)	3σ (%)	ME (%)
400.perlbench	4.04	0.37	1.28	0.12
401.bzip2	2.98	0.28	0.86	0.08
403.gcc	5.45	0.50	3.12	0.29
410.bwaves	5.97	0.55	1.52	0.14
416.gamess	2.74	0.25	1.84	0.17
429.mcf	0.56	0.05	0.34	0.03
433.milc	1.10	0.10	0.55	0.05
434.zeusmp	1.60	0.15	1.12	0.10
435.gromacs	16.53	1.53	1.61	0.15
436.cactusADM	1.53	0.14	0.34	0.03
444.namd	2.62	0.24	1.20	0.11
445.gobmk	2.04	0.19	1.17	0.11
450.soplex	2.41	0.22	1.12	0.10
453.povray	3.79	0.35	2.01	0.19
454.calculix	2.41	0.22	0.67	0.06
456.hmmer	3.85	0.36	2.77	0.26
458.sjeng	3.83	0.35	2.70	0.25
459.GemsFDTD	8.92	0.82	3.42	0.32
462.libquantum	0.61	0.06	0.57	0.05
464.h264ref	12.51	1.16	3.33	0.31
465.tonto	1.77	0.16	1.77	0.16
470.lbm	11.35	1.05	0.91	0.08
471.omnetpp	5.04	0.47	3.54	0.33
473.astar	0.94	0.09	0.23	0.02
481.wrf	0.62	0.06	0.15	0.01
483.xalancbmk	4.36	0.40	3.40	0.31

[15] J. Abella, E. Quiñones, F. J. Cazorla, Y. Sazeides, and M. Valero, "RVC: A mechanism for time-analyzable real-time processors with faulty caches," in *Proc. 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*, 2011, pp. 97–106.

[16] Y. Choi, S. Yoo, S. Lee, J. H. Ahn, and K. Lee, "MAEPER: Matching access and error patterns with error-free resource for low Vcc L1 cache," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1013–1026, 2013.

[17] E. Gunadi and M. H. Lipasti, "A position-insensitive finished store buffer," in *Proc. 25th Int. Conf. on Computer Design*, Oct 2007, pp. 105–112.

[18] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-binding: Enabling unordered load-store queues," in *Proc. 34th Annual Int. Symp. on Computer Architecture*, 2007, pp. 347–357.

[19] J. Kulkarni, K. Kim, and K. Roy, "A 160 mV robust schmitt trigger based subthreshold SRAM," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 10, pp. 2303–2313, Oct 2007.

[20] 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.

[21] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, Dec 2009, pp. 469–480.

[22] A. Sinkar and N. S. Kim, "Analyzing and minimizing effects of temperature variation and bti on active leakage power of power-gated circuits," in *Proc. IEEE Int. Symp. on Quality Electronic Design*, March 2010, pp. 491–796.

[23] A. Phansalkar, A. Joshi, and L. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 412–423.

[24] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1. ACM, 2003, pp. 318–319.