CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3906

A survey of techniques for designing and managing CPU register file

Sparsh Mittal*,†

Oak Ridge National Laboratory, Oak Ridge, TN, USA

SUMMARY

Processor register file (RF) is an important microarchitectural component used for storing operands and results of instructions. The design and operation of RF have crucial impact on the performance, energy efficiency, and reliability of the processor, and hence, several techniques have been recently proposed to manage RF in modern processors. In this paper, we present a survey of techniques for architecting and managing CPU register file. We classify the techniques across several parameters to underscore their similarities and differences. We hope that this paper will provide insights to researchers into working of RF and inspire even more efforts towards optimization of RF in next-generation computing systems. Copyright © 2016 John Wiley & Sons, Ltd.

Received 23 February 2016; Revised 8 May 2016; Accepted 4 June 2016

KEY WORDS: review; classification; CPU; register file; heterogeneous bank design; power management; soft-error resilience

1. INTRODUCTION

Modern processors use register file (RF) to store and provide intermediate results of computations, and thus, RF works as the highest level in memory hierarchy. The design of RF is governed by many different and often conflicting requirements. Because register read happens on critical schedule-to-execute path, performing this in a single cycle is desirable, and hence, RF latency limits the processor frequency. In case of large RF latency, RF access needs to be pipelined over multiple cycles, which increases pipeline length and penalties of branch misprediction, impacting complexity and performance.

The capacity of RF should be large to effectively exploit instruction level parallelism (ILP)‡ through large instruction windows. Operands, which cannot be accommodated in RF, need to be spilled to caches/memory [1], which incurs much higher latency. Further, to allow issuing and writing-back multiple instructions in each cycle, RF must also have high bandwidth and large number of ports. The pressure on RF is even higher in multi-threaded processors, because they host larger number of thread contexts. Because RF is frequently accessed, any soft-error in it can quickly propagate to other system components, and hence, protecting RF from soft-errors is important [2]. Finally, to meet power and thermal budget constraints, the energy consumption of RF should also be small.

*Correspondence to: Sparsh Mittal, 1 Bethel Valley Road, Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA.
†E-mail: mittals@ornl.gov.
‡We use the following acronyms frequently in this paper: error correcting code (ECC), floating point (FP), functional unit (FU), hardware (HW), hierarchical RF (HRF), instruction level parallelism (ILP), instruction set architecture (ISA), logical register (LR), physical register (PR), register cache (RegCache), register-renaming scheme (RRS), simultaneous multithreading (SMT), and software (SW). Also, an RF with m-read and n-write ports is shown as an mRnW RF.

Paper organization

Figure 1. Organization of the paper in different sections.

It is clear that a careful optimization of RF design and operation is required for striking a right balance between these diverse requirements. The recent design trends such as move towards multi-core processors and research in low-leakage memory technologies also call for a re-examination and further optimization of RF architecture. Recently, several techniques have been proposed to fulfill these needs.

Contributions: In this paper, we present a survey of techniques for architecting and managing CPU register file. Figure 1 shows the organization of this paper. We first provide a background on RF operation and terminology (§2.1) and discuss the RF organization and capacity in some commercial processors (§2.2). We then summarize the opportunities and obstacles in RF management (§2.3) and classify RF management techniques based on crucial parameters (§2.4). Then, we review innovative RF architectures (§3), such as heterogeneous bank RF designs and hieararchical RF designs. Further, we discuss techniques for improving performance and energy efficiency (§4), which work by increasing register availability and reducing register demand. Then, we discuss techniques for improving soft-error resilience (§5). We conclude this paper with a discussion of future challenges (§6).

Scope: To balance breadth and depth of coverage, we limit the scope of the paper as follows. We discuss RF management techniques for CPUs and not for other processing units such as GPU. We discuss techniques related to performance, energy, and soft-error resilience and not thermal management and process variation. We review system and architecture-level techniques and not circuit-level techniques.

2. BACKGROUND AND OVERVIEW

2.1. A brief note on terminology

We briefly discuss some concepts, which will be useful throughout this paper.

Register-renaming scheme: In dynamically scheduled processors, reuse of instruction set architecture (ISA)-defined registers (i.e., architecturally visible registers) leads to false write-after-write and write-after-read dependencies. To eliminate these, register-renaming scheme (RRS) is used, which assigns different *storage locations* for various instances of the same *register name*. The register name is termed as an logical register (LR), and the actual location used for storing the register value at any time is termed as a physical register (PR). Thus, RRS decouples ISA-defined RF architecture from the physical RF architecture on the processor chip and allows exploiting ILP.

As an example, assume the code sequence shown in the first column of Table I. Here, instructions (d) and (e) are independent of (a)–(c); however, because of the reuse of register R1, a false dependency is created. To avoid such false dependency, registers can be mapped to PRs as shown

Table I. An illustration of RRS.

	Before RRS	After RRS
(a)	`R1 ← Memory[16]`	`P1 ← Memory[16]`
(b)	`R1 ← R1+5`	`P1 ← P1+5`
(c)	`Memory[32] ← R1`	`Memory[32] ← P1`
(d)	`R1 ← Memory[96]`	`P2 ← Memory[96]`
(e)	`Memory[48] ← R1`	`Memory[48] ← P2`

RRS, register-renaming scheme.

in the second column of Table I .

Data bypassing: To avoid read-after-write hazards and reduce RF accesses, data bypassing is used whereby the outputs of functional units (FUs) are forwarded (bypassed) as an input of the same or another FU.

Register window: On a function call/return, the registers are saved in and restored from the stack, which incurs large overhead. To avoid this, register window mechanism is used whereby a private set of registers is allocated to a function when it is called. A runtime scheme renames local variables to the windowed registers. Thus, this mechanism allows passing parameters from one function to another without interacting with the stack.

2.2. Register file design in commercial processors

To show the trends in RF capacity and implementation of some research ideas in commercial products, we now briefly discuss the RF designs in some commercial processors.

Early processors had only few registers; for example, Control Data Corporation (CDC) 7600 [3] had only eight floating-point registers. The ETA-10 architecture [4] had 256 general-purpose registers, whereas Heterogeneous Element Processor (HEP)-1 [5] had several thousands of general-purpose registers [6]. Alpha 21264 processor had 72 FP registers and 80 integer registers [7]. As for recent processors, Intel's 32-nm Itanium 9560 processor has 128 80-bit floating-point registers and 176 64-bit integer registers for each thread [8]. Thus, each thread has 1.25-KB floating-point RF and 1.375-KB integer RF, for a total of 20-KB floating point RF and 22-KB integer RF for 16 threads.

In Alpha 21264 processor [7], the integer RF is split into two banks, which store duplicates of the 80 registers. Both the banks can supply operands to FUs, but only one bank can supply operands to a single FU, and thus, the RF organization is one level. The results are always stored in both the banks, although accessing the non-local bank incurs one-cycle penalty.

CRAY-1 processor [9] used a two-level RF organization with 8 and 64 registers at first and second levels (respectively) for addresses and a similar organization for scalars. Data movement between these levels was performed using explicit software instructions.

2.3. Factors and tradeoffs in register file management

We now discuss several factors which impact the effectiveness of RF management techniques and also determine their optimization targets.

2.3.1. Achieving high register file capacity. The number of PRs required in an OOO processor is roughly the sum of (1) threads multiplied by architected registers and (2) pipeline-depth multiplied by issue-width [10]. Thus, deep and wide pipelines require many PRs for storing the state of ongoing instructions. Conversely, the RF capacity limits the amount of exploitable ILP. However, increasing RF size may not translate into proportionate enhancement in performance, because increased RF latency reduces processor frequency [11]. These factors call for a careful balance of RF capacity with latency/area overheads.

2.3.2. Achieving high register file utilization efficiency. The RF resources are usually over-provisioned to handle the worst case or meet the peak performance demand; for example, an eight-wide issue processor may have 16R8W RF to support the worst case. However, the average utilization of RF remains low because of several reasons [12]. For example, some instructions (e.g., control flow such as branch) do not write to registers; some instructions have only one operand, and some values are obtained via bypass network. Also, some register values will be used by future instructions, which have not yet entered the instruction window. Similarly, registers remain allocated to threads waiting for long-latency misses. This problem is more severe for simultaneous multithreading (SMT) processors, which show higher cache miss-rate because of sharing of cache between threads. Intelligent techniques can alleviate such inefficiencies and can even achieve the performance of a higher-sized RF (e.g., double-sized RF [1, 13]).

2.3.3. Reducing number of register file ports. Increasing RF port-count enhances the bandwidth but also incurs area/power overheads. For example, Naresh *et al.* [14] note that for a 64b 128-entry RF, moving from 2R1W to 8R4W port design leads to 1000% increase in area, 75% increase in leakage power, 50% increase in latency, and 75% increase in read energy. Similarly, Kim *et al.* [15] note that a 16R8W RF has 10X higher area than a 4R2W RF. In fact, Shioya *et al.* [16] note that because RF area increases with square of port-count, a 12R6W *1KB RF* has nearly *same* area as a 1R1W *16KB L1 D-cache*.

2.3.4. Reducing register file power consumption. RF accounts for significant fraction of processor power, and hence, RF power management is vital. For example, in Intel's 32-nm Westmere core, RF accounts for 30% of leakage and 30% of dynamic power [17]. Similarly, RF contributes 42% of data path power and 16% of processor power in Motorola M.CORE architecture [18].

2.3.5. Relative merits of hardware and software management. Several techniques such as early register deallocation work based on RF usage pattern, which can be estimated by hardware (HW) or software (SW). While HW-based techniques incur smaller overhead and can account for input and runtime variations, they cannot accurately determine the last use of a register. By comparison, compiler has knowledge about control flow, and thus, it can deallocate a register at right time. However, compiler-based techniques may be infeasible for multi-cores, which run arbitrary combinations of workloads. Also, they may increase code-size and require recompilation.

2.3.6. Minimizing additional complexity. Several RF management strategies may increase the complexity of RF design and operation. For example, some techniques may execute a few instructions multiple times [19, 20] or steal registers from one instruction to give to another [20]. Other techniques may increase port-count [21] or RF capacity requirement [22]. A few techniques require auxiliary structures (e.g., buffer) for temporarily storing register values [15, 23, 24] or other instruction state [25]. Also, RF management techniques usually require special arrangements for handling exceptions [1, 23, 24, 26–32]. Lowering these implementation overheads usually come at the cost of reduced performance enhancement [33].

2.3.7. Unique architecture of register file. The properties and operational characteristics of RF are markedly different from other processor components, such as cache and main memory [34, 35]. For example, RF sees much lower degree of temporal reuse than cache, and some PRs may not even see more than one access. RF access lies at critical access path, and hence, it is primarily optimized for speed. By comparison, the latency of caches (especially lower level caches) and main memory can usually be hidden by schemes such as write-buffering and prefetching [36]. Hence, optimization strategies typically used for other components cannot be directly employed for managing RF.

2.4. A classification of register file management techniques

To highlight the similarities and differences between the techniques, Table II classifies RF management techniques based on several parameters, such as their objective, the leakage management approach, and viz. state-destroying (power gating) or state-retentive (drowsy) approach. Table II also mentions novel RF architectures and management approaches. It is noteworthy that some of these

Table II. A classification of research works.

Category	References
	Optimization target
Performance	[1, 10, 11, 13, 20–22, 25, 27, 28, 30, 31, 38–51]
Dynamic energy	[14–16, 22, 24, 26, 28–30, 40, 42, 43, 45, 50, 52–62]
Leakage energy	[14–16, 21, 22, 28, 45, 52, 55, 57, 62–65]
Soft error	[55, 58, 59, 61, 66–71]
NVM lifetime	[32]
	Runtime leakage management
State destroying	[28, 43, 62, 64, 65]
State retentive	[57, 63]
	RF architecture
Heterogeneous banked/port RF	[11, 42, 49, 53, 60]
Hierarchical RF	[6, 11, 30]
Use of RegCache	[16, 38, 45]
Reducing port-count	[1, 15, 16, 22, 30, 48, 56, 60]
	Management approach based on . . .
Late register allocation	[19, 20, 47, 56]
Early register release	[10, 23, 25, 27–29, 31, 33, 40, 41, 46, 72]
Value locality	[33, 44, 50, 65, 73]
Bypass network	[11, 13, 21, 23, 24, 29, 30, 38, 42, 45, 54, 74]
Narrow width values	[10, 39, 40, 42, 48, 60, 73]
Use of compiler	[13, 26, 28, 31, 32, 45, 53–55, 57, 59, 61, 66–68, 70]

NVM, non-volatile memory; RF, register file.

techniques *increase* RF resources (e.g., ports, bandwidth, and capacity) or their availability, by performing RF virtualization, late allocation, early deallocation, etc. By comparison, other techniques *reduce the demand* for RF resources, by leveraging RegCache, value locality, narrow-width values (e.g., a 16-bit value in a 64-bit register [37]), and bypass network. Clearly, they are complementary and inter-related. Detailed discussion and analysis of them are presented in subsequent sections.

As for optimization heuristics, researchers have used integer linear programming [67], graph partitioning [53, 57] and dynamic programming [70]. Some techniques work based on program control flow, for example, function call semantics [31, 61, 64] and nested loop structures [32, 53]. Similarly, researchers have addressed the issues or exploited the properties of specific processor architectures, for example in-order core [63], out-of-order core [31], statically scheduled processor, for example, very large instruction word [13], SMT processor [1, 33], and embedded processors [26, 32, 61, 64, 70]. In general, the techniques focused on saving power are especially important for embedded systems; those focused on reducing latency and improving bandwidth are very important for server-class systems, and those focused on improving soft-error resilience are highly important for mission-critical (e.g., space and health) systems.

In next three sections, we discuss several works by roughly organizing them into multiple groups. The works presented in these sections are deeply intertwined, and although we review a work in a single group, many of them belong to multiple groups.

3. INGENIOUS RF ARCHITECTURES

It is well known that different registers show different properties, such as reuse frequency [53] and data-width [42, 50, 60]. To exploit these features, heterogeneous-bank RF and RegCache (§3.3) designs have been proposed. In heterogeneous-bank designs, the contents of different banks may be *mutually exclusive,* and thus, a register may be mapped to only one bank (§3.1). For example, banks with smaller width or smaller capacity consume smaller access energy and by mapping selected (e.g., narrow or frequently accessed, respectively) registers to them, energy saving can be achieved. In a similar vein, ports with lower width can be used for narrow data values [60]. Second category of heterogeneous-bank designs is hierarchical RF (HRF) where the smaller capacity bank may *cache selected values* of the larger capacity bank (§3.2).

3.1. Heterogeneous-bank register file designs

Nalluri *et al.* [53] note that most RF accesses are concentrated to only a few registers. Based on this, they propose a dual-bank RF design where one bank is smaller than the other. By mapping frequently accessed registers to the smaller bank, overall energy can be saved. They model the problem of mapping registers to two banks as a graph partitioning problem and solve this using a greedy heuristic. The access behavior is obtained using profiling. The partitioning, which minimizes energy consumption is finally selected. To implement this partitioning, they study both HW and SW-based strategies for mapping registers to different banks. They also propose a compiler-based static analysis scheme for finding optimum dual-bank design. Because applications generally spend a large fraction of execution time in inner loops, most frequently accessed basic blocks appear in innermost loops. Based on it, this scheme determines the maximum number of registers needed for allocating variables of loop bodies at maximum nesting depth using Left-Edge algorithm. Based on bank configuration found by this scheme, registers are allocated to banks by using a two-phase algorithm where first local variables and then global variables are allocated to different banks. They further generalize their dual-bank design to a multi-bank design and find an optimal configuration by recursively calling the algorithm for dual-bank design. They show that the static analysis based approach provides similar energy saving as the profile based approach while being orders of magnitude faster in runtime. Also, compared to a dual-bank design, the multi-bank design provides even larger energy saving over monolithic RF.

Guan *et al.* [57] propose dividing the RF into two unequal portions sharing the same address space, where the smaller portion holds more frequently accessed registers. A majority of RF dynamic energy consumption comes from bitline energy which increases linearly with the number of registers on the bitline. Their technique splits the bitline into two segments to lower the dynamic energy consumption. A multiplexer is used to choose the desired output. The size of smaller portion is kept power of two to reduce its access delay; thus, a typical 32-entry RF may be divided as (16,16), (8,24), (4,28), and (2,30). Their technique also transitions the idle RF portion to state-preserving mode to save leakage energy. They model the problem of placing most intensely accessed registers in small portion such that average power is minimized and solves this as a graph partitioning problem. Compared with monolithic RF, their partitioned RF design saves more than half the energy while incurring only small performance loss.

Gonzalez *et al.* [50] propose an RF organization, which is based on the observation that higher bits of many register values may be same. Values similar in lower bits are termed as 'short', and those short values whose upper bits are all zeros, or all ones are termed as 'simple'. Values, which do not have similarity with other values, are termed as long. They replace a K-entry conventional RF with three RFs, which store 'long', 'short', and 'simple' values. The long RF stores 64-bit values, whereas short RF stores certain upper bits of short values. Both these RFs have less than K entries. The simple RF has K entries, and it stores unique lower bits of long, short, and simple values. On a read operation, if a PR is found to have long or simple value, it is accessed from long RF or simple RF, respectively. If it is found to have short value, higher bits from short RF and lower bits from simple RF are read to produce the final value. Similarly, a write operation is performed after ascertaining its type. Their technique reduces area, energy consumption, and access latency of RF.

Wang *et al.* [42] present an asymmetric RF bank architecture. For integer benchmarks in SPEC2000 suite, they observe that nearly $50\%, 60\%$, and 94% register values can be represented with at most 16, 32, and 34 bits, respectively. Based on this, they design 16-, 34-, and 64-bit banks. This reduces the total area and latency of RF. Their technique predicts the data width of the destination register of a decoded instruction and, based on the width, renames the register to a PR in a suitable bank. If no free entry is available in this bank, a PR from a wider (than the predicted width of value) bank may be allocated to a value. Bypassed values are not accessed from RF, which reduces read accesses to it. Also, write port conflicts are reduced using port scheduling scheme from Balasubramonian *et al.* [30]. They show that by serving most accesses from 16- and 34-bit banks, their technique saves significant energy with small performance loss over an ideal 1-cycle monolithic RF.

Aggarwal *et al.* [60] present a heterogeneous port design, which uses narrow and wide (i.e., normal) ports. In *narrow* ports, bitlines for higher bit positions are removed, and hence, they can read/write narrow values only (e.g., 8b and 16b). The *normal* ports can operate on both narrow and wide (e.g., 32b) data. They also provide narrow read and write ports for some FUs, and these FUs can only have narrow inputs/outputs. They study two policies for setting the width (i.e., narrow or wide) of two operands of an instruction, first that sets them only at time of renaming and second that possibly updates these widths on execution of producing instruction. They observe that the first scheme leads to performance penalty because for several instructions, operand widths are not known at dispatch time, and hence, the first scheme conservatively assumes them to be wide. By comparison, the second scheme does not harm performance. They evaluate both partitioned and monolithic heterogeneous-port RF and find that the partitioned design provides larger reduction in dynamic power.

3.2. Hierarchical register file designs

In HRF, only one level, for example, upper-level RF (R_{L1}) provides values to FUs, and hence, the effective latency of HRF becomes that of R_{L1} (usually one cycle), which is lower than that of an iso-capacity monolithic RF. This allows increasing processor frequency. Also, the bypass logic required in HRF is the same as that in a 1-cycle RF. We now discuss several techniques, which propose HRF designs.

Cruz *et al*. [11] note that while a processor requires many PRs, only a few registers may need to be used at any time for operating the processor at its peak throughput. They present an asymmetric bank architecture where the latency, number of ports, and registers in different banks are different. This can be organized as either a single-level or a multi-level design. In the multi-level design (assuming a 2-level design), R_{L1} caches hot operands from lower-level RF (R_{L2}), which stores all the values. R_{L1} has few registers (e.g., 16) and many ports to achieve 1-cycle latency, whereas R_{L2} has many registers (e.g. 128) and few ports and, thus, has higher latency. They propose two caching schemes for R_{L1}. In the first scheme, only non-bypassed values are stored in R_{L1}. The second scheme caches source-operands of only those unissued instructions, which have all their operands ready. These values will not be forwarded by bypass network and are expected to be used soon. They also explore two schemes for fetching values from R_{L2} to R_{L1}. The first scheme fetches data only when all operands of an instruction are ready. Because this conservative scheme can delay some instructions, their second scheme proactively fetches in R_{L1} the other source-operand of an issued instruction that consumes the result of the existing instruction. Their results show that first caching scheme and second fetching scheme provide better performance than their counterparts. Overall, their technique improves the processor performance significantly.

Balasubramonian *et al*. [30] present a two-level RF design for reducing RF size requirement. On an instruction dispatch, registers are allocated from R_{L1}. When a register is no longer required for any consumer, it is moved to R_{L2} for handling branch misprediction and exceptions. Because R_{L1} needs to store only those values sourced by FUs, its size and, hence, latency can be small. Although the hierarchical design improves performance, it also increases energy consumption because of the use of additional structures for tracking status of registers (e.g., remaining consumers). For improving RF energy efficiency, they note that while a multiple-instruction-issue processor may theoretically require many ports, the actual port requirement remains small (refer §2.3). This allows reducing the number of RF ports, and hence, they present a banked architecture where each bank has only one read port and one write port. Because an instruction that has both the source operands in a single bank cannot read them in a single cycle, their design allows an instruction to read and store one operand and continue competing for the bank to read the second operand. When both operands are read, the instruction begins execution. Compared with a monolithic RF, their banked design is faster, more scalable, and energy efficient.

Sangireddy *et al.* [49] note that there are large periods between allocation, consumption, and release of registers, whereas the duration for which a register supplies values to consumers is a small fraction of its lifetime. They propose an RF design, which exploits this access pattern. They use three heterogeneous RF banks, named RF1, RF2, and RF3. RF1 stores few registers and has adequate number of write and read ports for supporting issue width of the processor. Both RF2 and RF3 store equal number of registers and have a few write and read ports. LR to PR mapping is carried out at dispatch stage using PRs in RF2. Only RF1 supplies data to FUs, and thus, effective RF latency becomes one cycle. Results are always stored in RF2. Only those registers, which are expected to be soon consumed by a ready instruction, are copied from RF2 or RF3 to RF1. When RF3 has free registers, some values are moved from RF2 to RF3, and thus, registers in RF2 are released much sooner than in a traditional monolithic RF. A register in RF3 is freed in the same way as in a traditional RF, that is, on commitment of another instruction writing to same LR. Their design reduces RF latency and provides higher RF bandwidth, which allows exploiting higher ILP.

3.3. Register cache-based register file designs

A register cache (RegCache) allows reducing RF latency or the number of RF ports [16] or both [38, 45]. Some techniques predict the number of uses of a register, and once all its uses have been completed, it can be released [28, 72, 75] or evicted from RegCache [38, 45] or from upper-level RF in an HRF [30]. We now discuss some techniques, which use RegCache.

Butts *et al.* [38] propose insertion and replacement strategies for managing RegCache. Their technique initially predicts number of readers of a register value with the help of a use predictor. Based on this, only registers with non-zero uses are stored in RegCache. Thus, values, which will not be used after being supplied from bypass network, are not inserted in RegCache. On each use of a value, its expected-reuse count is decremented. In the event of replacement, a RegCache value with least number of reuse (generally zero) is replaced and this minimizes miss-rate in RegCache. They also propose a scheme for reducing conflicts in set-associative RegCache. During register renaming, in addition to renaming an LR to a PR, they also assign a RegCache set to an LR using one of the three indexing schemes. The first scheme computes sum of expected reuse of all values assigned to a set, and then, the next instruction is assigned a set with minimum sum. This scheme attempts to uniformly distribute high reuse values in multiple sets to reduce conflicts. The second scheme assigns sets to instructions in round-robin order; thus, it attempts to assign different sets to results of instructions executed together. The third scheme skips those sets in round-robin order which have more than a certain number of high-reuse values assigned to it. They show that their schemes reduce conflicts in RegCache. Overall, their RegCache design and management policies provide significant performance improvement compared with monolithic RF and HRF design.

Jones *et al.* [45] present a compiler-based technique to improve the effectiveness of a RegCache. They execute the application using training input and record the execution frequency of every basic block in the control flow graph. Using this, compiler collects the number of uses in or after each node's most frequently executed successor. For each consumer instruction, the use-count of the register is incremented by one. This information is stored in spare bits in the ISA and is passed to the microarchitecture. When the defining instruction of a register writes back, a decision is taken whether the instruction writes to the RegCache. For this, the number of reads from bypass network is subtracted from the use-count, and if this value is no more than zero, the register is not inserted in RegCache, because it is expected to see no reuse in the future. When a consumer reads a register, its use-count is reduced by one. When RegCache becomes full, the entry with the lowest use-count is evicted. Also, if a PR is deallocated, the corresponding entry in the RegCache is evicted. On a miss in RegCache, the value is read from the main RF. This, however, stalls the dependent instructions, and to avoid this, their technique stops issuing instructions after the one that saw a RegCache miss. Instruction issue begins again at a later cycle. They show that their technique reduces RF accesses and port requirement, which lead to performance and energy gains.

Shioya *et al.* [16] note that although a RegCache reduces effective latency of accessing RF, a RegCache miss requires stall or flush, which disturbs the pipeline and degrades IPC. They propose a design where RegCache is *not* used for reducing latency but only for reducing the number of ports in main RF and simplifying the bypass network. In their design, all instructions go through pipeline stages for reading main RF, irrespective of RegCache hit/miss. Because of this, a RegCache miss does not immediately disturb the instruction pipeline. Thus, their technique always predicts a 'RegCache miss' and, hence, removes the misprediction penalty, which is the main reason for improvement of performance in their design. By comparison, traditional RegCache designs, which predict a 'RegCache hit', incur large penalty on a RegCache miss. Compared with an RF without RegCache, their design reduces number of ports in RF that reduces area and power consumption, while incurring negligible loss in performance.

3.4. Register file designs for low port-count and high bandwidth

To enhance RF bandwidth and reduce port conflicts, some techniques perform register duplication [14, 47], delay register renaming [56], make bypass network more efficient [54], and employ read/write specialization [22]. For instructions with two operands, some techniques fetch data from RF only when both operands are ready, which avoids RF pollution [11], whereas others allow reading one operand and continue waiting for another which reduces *peak* port demand [15, 30].

Naresh *et al.* [14] use ideas from network coding to reduce the number of RF ports. They divide an RF into two interleaved banks with half the number of R/W ports. To increase the bandwidth, they also add a third bank, which stores the XOR of the values written back by these two banks. Assuming the values written by these banks is B1, B2, and B1$\oplus$B2, respectively, the value B2 can be generated as (B1$\oplus$ B2) $\oplus$ B1, and so on. Thus, a single 4R2W RF can be replaced by three 2R1W banks. While requiring limited duplication and additional control logic, their design reduces area, latency, and leakage power of RF and incurs very small loss in application performance.

Duong *et al.* [47] present a scheme for reducing bank conflicts in processors with large RF and smaller frequency. Their scheme maps an LR to two PRs in different banks. On a write operation, both PRs are updated. To reduce bank conflicts during writes, their technique delays PR allocation till the writeback stage [20]. A read operation can be performed in either of the two banks, which provides more opportunity of finding an empty port for reducing bank conflicts and, thus, improves performance.

Kim *et al.* [15] present a technique to reduce the number of RF ports. They use three additional structures, an operand prefetch buffer (OPB), an operand prefetch request queue (OPRQ), and a delayed writeback queue (DWQ). They note that several times, only one of the two operands of an instruction may be ready and, hence, an instruction has to wait in the instruction queue. Because one operand is ready, the address of PR can be ascertained, and thus, instead of waiting for both operands to become ready, the operand can be prefetched in OPB. When there is no free read port or operand prefetch space during dispatch cycle of instruction, the PR address of ready operand can be stored in OPRQ. Using this, prefetch can be issued when resources actually become available. Thus, peak demand for read ports can be reduced by distributing read port access over multiple cycles. Based on the observation that most results from FUs are consumed within a few cycles, DWQ stores recently generated results from FUs. This avoids the need of accessing RF for them and lowers the requirement of ports. By using OPB/OPRQ and DWQ together, port demand can be lowered even further. Their design reduces RF area, access latency, and energy without harming overall performance.

Seznec *et al.* [22] note that traditional superscalar processors allow any general-purpose FU to access any general-purpose PR. They propose RF write specialization and read specialization techniques for clustered superscalar processors. They divide PRs into subgroups, which are only write-connected or read-connected with subsets of FU outputs or inputs, respectively. In write specialization, an instruction executing on a cluster writes its results only to a subset of RF. In read specialization, the clusters can read their operands from a subset of PRs only, assuming that at least one cluster can execute the instruction. This reduces the requirement of write and read ports on each PR, respectively. Their approach simplifies bypass network and wakeup logic, because only few clusters need to be connected to bypass point and be monitored by the wakeup logic. Further, although requiring larger number of PRs, their approach reduces RF area, latency, and power consumption.

3.5. Non-volatile memory-based register file designs

Yang *et al.* [32] study an RF design where the static random access memory (SRAM)-based RF is augmented with an non-volatile memory (NVM)-based RF. The SRAM RF serves the read/write accesses, and its contents are periodically backed up in NVM-based RF. However, because NVM has high write latency/energy and limited endurance [35, 76], this design incurs performance overhead and has limited lifetime. They propose two techniques to address these issues. They note that only the live variables are required for recovering from an interrupt/exception. Also, the live variables, which do not get updated inside a frequently executed loop, need to be backed up only once before entering the loop. Only those variables, which are live at loop entry and updated inside loop, need to be repeatedly backed up inside loop. Their first technique uses these insights to reduce the number of writes to NVM RF. Because in nested loops, a variable can be backed up at multiple points, this technique also ascertains the best position for performing backup. In case of high failure probability, inner loop is preferred because it allows the computation to progress at fine-granularity. In contrast, in case of low failure probability, outer loop facilitates lower backup frequency, which reduces the backup cost. This technique only analyzes those loops, which account for more than 10% of the execution time. The first technique can lead to non-uniform write distribution, and hence, a few NVM registers can fail much early than the remaining registers. Their second technique addresses this by dynamically rotating register mapping to uniformly distribute the writes to all registers. Because 'live and updated' registers are written to NVM RF at different frequency than the 'live and non-updated', their technique rotates them at different frequencies. Their techniques bring significant improvement in lifetime of NVM RF.

4. IMPROVING PERFORMANCE AND ENERGY EFFICIENCY

Conventional RF management schemes allocate a register early (decode stage in the pipeline) for tracking dependencies, although the register stores a value only when the instruction writebacks. Similarly, these schemes release a PR only at commitment of another instruction with same destination LR, although it can be released early, e.g. on commitment of last consumer of the register. In §4.1 and §4.2, we discuss techniques which alleviate these inefficiencies by on-time (de)allocation of registers.

4.1. Delaying register allocation

Gonzalez *et al.* [19] present an RRS which makes use of 'virtual-physical registers' (VPRs). VPRs are merely names (and not actual storage) used for identifying results of future instructions. VPRs are used for tracking dependencies between instructions for whom a register has not been allocated to their destination operand. In their scheme, at the time of an instruction decode, its destination register is mapped to a VPR. Only on completion of execution of an instruction, a PR is allocated for storing its result. However, in this scheme, instructions, which do not have a PR at the time of completion, need to be re-executed (e.g., by more than three times). To address this, they propose a second scheme which allocates PR at the time of instruction issue. They show that the first scheme is more effective in reducing RF pressure and improving performance, because re-executed instructions generally use idle resources. Overall, their RRS boosts performance compared with conventional RRS by virtue of leveraging higher amount of ILP.

Monreal *et al.* [20] extend VPR-based RRS [19] with a register allocation strategy. The traditional strategy allots PRs in program order and stalls the processor in absence of a free PR. In their strategy, a PR is allotted to an instruction right before writeback to minimize idle registers. If no PR is available at this point, they check whether a register has been allotted to a more recent instruction. If so, this register is stolen from the recent instruction (most recent if multiple such instructions exist) and allotted to older instruction. This allows early commitment of older instruction, which indirectly avoids delay in commitment of recent instruction. With this strategy, some instructions may be executed multiple times, which actually proves to be beneficial in two ways. First, any execution of load prefetches data in the cache and thus leads to cache hit on last execution. Second, on a misprediction for a branch instruction, fetching on right path begins much early than in traditional RRS. For same performance, their strategy reduces RF size requirement, which reduces RF access latency.

4.2. Performing early register deallocation

Quiñones *et al.* [31] present an early register release scheme for out-of-order processors using register windows. They note that function invocation/return mechanism provides opportunity for early release of registers. On commitment of `return` statement of a function, every PR allocated by the function can be deallocated, because the values will not be reused. Also, in case of scarcity of PRs, those assigned to caller function instructions, which are not in-flight, can also be deallocated after saving them to memory because they will be reused. Also, depending on the registers required on a control flow path, the context size can be increased or reduced using compiler-inserted commands, and thus, when the context is reduced, additional registers can be deallocated. Register window scheme maps compiler-defined local variables to architectural registers. To recover this mapping information in the event of exceptions and branch mispredictions, their technique also stores state information of all uncommitted contexts. Their technique reduces register lifetime and brings the requirement of PRs to the minimal number required for ensuring forward progress (i.e., number of architectural register+1) while providing nearly same performance as an infinite RF.

Jones *et al.* [28] present two compiler-based schemes for early release of registers. The first scheme deallocates a register when its last user instruction is committed because at this point, all consuming instructions have read the register. The second scheme deallocates a register when the last user instruction is issued, which is even earlier than the first scheme. In both schemes, before deallocation, the register values are backed up to allow recovering from an interrupt, etc. Compared with the schemes of Ergin *et al.* [75] and Monreal *et al.* [72], their schemes perform deallocation earlier because Ergin *et al.* [75] deallocate a register after commitment of the original defining instruction, entry of the redefining instruction into pipeline, and reading of the value by all consumers. Also, Monreal *et al.* [72] perform deallocation when the redefining instruction becomes non-speculative and every consumer has read the value. They show that their technique can reach a large fraction of the potential of an oracle technique for register occupancy reduction.

While instructions stall for L2 miss on load operations, RF remains under-utilized. Sharkey *et al.* [41] note that most of the instructions stalling for long-latency loads are actually independent of the load operation and although these instructions free the issue queue entries, they release PRs only upon completion of cache miss request. Based on these, they present a scheme for eagerly deallocating PRs assigned to threads, which see L2 cache misses. Even before completion of cache miss, their technique speculatively commits load-independent instructions and deallocates PRs corresponding to the earlier mapping of their destinations. These PRs can be allocated to instructions in the same thread and other threads. Speculatively committed instructions are actually committed on completion of cache miss, and load-dependent instructions are not speculatively committed. They show that their technique improves throughput of an SMT processor.

Alastruey *et al.* [23] present architectural mechanisms for allowing speculative renaming used in the techniques for early release of PRs and for omitting allocation of PRs for bypassed values (§4.4). An early released register is moved from main RF to auxiliary RF, and results not allocated in main RF are also stored in auxiliary RF. To address an operand residing in any of the two RFs, they decouple dependence tracking from PR identification using idea of virtual registers [19, 20]. Although accessing a value from auxiliary RF incurs additional delay, it avoids the need of storing those values in main RF. A value is discarded from auxiliary RF only when it is safe to do so, and this allows supporting precise exceptions. They employ a last-use predictor for guiding both speculative release and omission based techniques and achieve significant performance improvement.

4.3. Leveraging value locality

Several times, multiple registers may store the same value because of various reasons, for example, instructions executed within a dynamic instruction window generally produce same result, and this is especially true for integer results. Register-to-register move operations, trivial computations (e.g., $A * B$ where A or $B = 0$), etc. also lead to value locality. Based on this, some techniques share same PR between multiple LRs to reduce the requirement of PRs [33, 44, 65, 73]. Partial value locality, which refers to the case where only higher-order bits are same, has also been exploited [50]. Because 0 and 1 occur very commonly, some techniques make special provision for identifying them [33, 73]. Some researchers propose register reference counting to properly release the shared registers [51, 65]. Register sharing, however, increases bank conflict, and to avoid this, some techniques map *one LR* to *multiple PRs* [47], which is just opposite of register sharing.

Jourdan *et al.* [44] present a technique which identifies redundant results in register-renaming stage by using a small cache and exploits this value locality for four optimizations. First, multiple LRs with same value can be mapped to the same PR, which allows reducing RF size and number of ports or increasing effective size of active instruction window. It also allows eliminating register-to-register and immediate-to-register move operations. Second, instead of FUs, the register renamer can generate results of instructions, and thus, execution of certain instructions can be avoided. Third, instead of one instruction, an earlier instruction can generate results, which allows early scheduling of dependent instructions. Fourth, memory renaming can be unified with register renaming for extending the previously mentioned three benefits to both RF and memory addresses. It also eliminates store operations, which move the value to be stored from register renamer to the memory renamer. They show that their technique reduces RF size requirement and improves performance.

Balakrishnan *et al.* [73] present techniques to exploit value locality. One of their techniques maps destination LRs of multiple instructions to a single PR if they produce the same result. However, because of requirement of additional structures, this technique incurs high overhead, and hence, they propose two other techniques, which lower the overhead but can avoid duplication of selected values only. Noting that '0' and '1' are the most frequently written values in the registers, their next technique assigns 0 and 1 to some PRs, and on a reference to these registers, actual access need not be performed. The limitation of these two techniques is that they require rebroadcasting the tag after register remapping. To address this, their last technique stores 2 extra bits with each PR. One of these bits stores the $0/1$ 'value' and another bit shows whether this previous bit is valid. By using these extra 2 bits, their technique frees the main PR for storing outcome of some other instruction. They also use a tag naming strategy for referring to PR and its 2 bits together or separately. On using this, the tags of source registers need not be changed or broadcast. They show that their techniques reduce register accesses and improve performance.

Tran *et al.* [33] present two register sharing schemes, which focus on reducing implementation overhead at the cost of losing opportunities for register sharing. The first scheme uses a single PR for multiple LRs with same value which is restricted to 0 and 1. Thus, unlike others techniques (e.g., [44, 73]), their technique does not use cache to detect value locality and, hence, can detect locality for limited cases only. For trivial computations, which can be pre-decided to produce zero (e.g., $A + B$, when $A = 0$ and $B = 0$), the destination register is remapped to the zero register. Instead of releasing a register the next cycle after writeback, their scheme delays the release until commitment of an instruction, which requires simpler implementation. Also, in every cycle, only one register sharing update is allowed. Their second scheme detects self-overwriting (those that update one of the source register, e.g., $P + Q \rightarrow P$) instructions which are the only consumers of the value in their destination registers. For such instructions, there is no overlap between lifetimes of different versions of logical destination register in PRs and hence, their technique uses same PR for storing multiple versions of the LR. For reducing implementation overhead, sharing is limited to at most three instructions, which appear within the same basic block of the original assignment. Their simple scheme provides nearly half the benefit of a complex scheme while incurring much lower overhead.

Battle *et al.* [65] note that conventional RF management mechanisms work by (de)allocating a register from a free list organized as a circular queue. They propose a mechanism where (de)allocation events *update reference counts* instead of actually *freeing the registers*. Reference counting is implemented using a bit-matrix where each PR has a column, and each in-flight instruction has a row. By doing NOR operation on the columns, a bit-vector free list is created from which PRs are allocated based on priority scheme. Their mechanism includes the conventional mechanism as a special case and also allows versatile optimizations. For example, the conventional mechanism reduces the gains from RF power gating because it scatters the active registers in different banks. By contrast, their mechanism packs registers naturally and, thus, increases the scope of RF power gating. Similarly, in conventional mechanism, register sharing becomes difficult because sharing requires the instruction overwriting the younger sharer, and not the older sharer, to release the shared register. By comparison, their mechanism tracks register usage behavior in a single location and eliminates the need of assigning register releasing responsibility to any specific instruction beforehand. Their mechanism also supports execution-driven register release and speculative retirement more effectively than the conventional mechanism. They show that the techniques enabled by their mechanism provide performance and energy gains, which easily offset the area/energy overhead of reference counting.

4.4. Leveraging bypass network

Many techniques work by avoiding accesses to monolithic RF or HRF [11, 13, 21, 23, 24, 29, 30, 42, 54, 74] or RegCache [38, 45] for operands available from bypass network. We now review some of them.

Park *et al.* [56] propose two techniques, which reduce the requirement of RF read and write ports, respectively. They note that when the result broadcast by producer wakes up source operand of consumer in the issue queue, the producer generally writes back after the consumer has issued. Thus, bypass network can deliver the operand to the consumer except in rare cases, for example, data hazards. Use of wakeup logic helps in distinguishing between the operands, which are woken up by in-flight producers and the operands whose producers have written back. Their first technique does not access RF for bypassed operands, and because data hazards are rare, the accuracy of bypass hints is very high. Their second technique uses banking to provide large write bandwidth. In traditional scheme, banks are allocated to instructions in round-robin fashion for achieving uniform distribution and reducing bank conflicts during read and writeback. However, because of out-of-order execution and varying instruction latencies, instructions renamed in the same cycle may not read/writeback in the same cycle, and hence, they may see bank conflicts. This happens because in traditional RRS, same mapping is used for PRs and dependence tags. Their technique decouples these two and, thus, assigns dependence tags during rename (same as traditional scheme) but assigns PRs between issue and writeback (e.g., just before writeback). This eliminates conflicts between writes. They show that on using both techniques together, large saving in energy-delay product is achieved.

Tseng *et al.* [21] present a banked RF design, along with schemes for managing it in deeply-pipelined superscalar processors. Their scheme issues potentially conflicting instructions in speculative manner, and thus, register banks need not be arbitrated on the critical wakeup-select path. In case of conflicts, the issue window is immediately repaired, and conflicting instructions are reissued. Detection and resolution of conflicts happens in one pipeline stage, which obviates the need of write buffering or stalling the pipeline. RF access is avoided for bypassed values. To check which operands can be obtained via bypass, they store a bypass bit for every instruction window operand and read this along with result of wakeup tag match in each cycle. If an instruction is chosen in the same cycle when it was woken up by a tag match, the bypass bit is set showing that bypass network can supply the operand. If the instruction is not chosen for issue, the bit is reset in the next wakeup phase. Their bypass check scheme, although conservative, is always correct, whereas that of Park *et al.* [56] works by prediction and, hence, may sometimes be inaccurate. Their design reduces RF area, access latency, and energy, while incurring only small loss in overall performance and a small increase in port-count per bank.

Ayoub *et al.* [26] note that writes to RF are not required for bypassed values. They propose a technique, which identifies such short-lived registers using compiler and renames all or most of them to a few 'transient' registers specified at the time of HW design. These registers do not write to RF which saves RF energy. In case of cache misses or exceptions, write operation to RF is not skipped to avoid correctness issue. They also propose renaming algorithms, which take into account whether sufficient number of registers are available to be used as transient registers.

Ponomarev *et al.* [24] present a technique for saving energy of writeback and commit operation. They note that the destination registers of most register values are renamed before these values get written back. Their technique buffers such transient values in a small separate RF, which avoids writing to result repository (in re-order buffer (ROB) or a physical register) or committing them to RF. When a subsequent instruction with the same destination architectural register is committed, its previous instance is not required. At this time, buffering a value is no longer required for recovering from branch mispredictions and recreating correct state for handling an interrupt and hence, the value is evicted from the small RF. Their technique reduces the access energy of ROB and RF.

Balkan *et al.* [29] note that a large fraction of result values can be bypassed to consuming instructions, they are not required for recovering from a branch misprediction, and their consuming instructions are not subject to memory replay trap. They propose a technique that recognizes such values and drops them from the data path immediately after their generation. Thus, these values need not be written to PRs, which allows early deallocation of PRs upon instruction execution. This, however, precludes restoring the precise state from ROB-based mechanisms. To recover the precise state, their technique periodically checkpoints RF such that the transient values are not dropped during checkpoint generation period. They show that their technique improves both performance and energy efficiency.

Yan *et al.* [13] present a technique for statically scheduled architectures, for example, very large instruction word, which do not use any RRS. If a variable is alive for shorter duration than the time it takes between forwarding of a value by the producer and reception by the consumer (i.e., length of the bypassing network), then their technique does not allocate a register to it. For such variables, only a placeholder, called virtual register, is used for identifying data dependencies and delivering the data to the correct instruction. They note that because most variables are alive for only a few instructions, by avoiding allocation of actual registers to them, they can be more effectively used for long-lived variables. Their technique reduces register spills and improves performance without increasing RF size.

Park *et al.* [54] propose an instruction scheduling scheme for increasing the number of operands available via bypass network for reducing RF accesses. Their compiler tracks when instructions bypass their results, which source operands read them, and when the results are stored in RF. Their scheme evaluates all permutations of instructions, which do not violate data dependencies. For each schedule, the number of operands of its instructions , which are available from bypass network, is computed, and the schedule with the largest value is chosen. Such scheduling is performed for every basic block. Their scheme attempts to schedule dependent instructions nearby for allowing them to supply dependent operands via bypass network. They perform evaluations using Intel XScale architecture and show that their schemes reduce RF power consumption significantly.

4.5. Leveraging narrow-width values

Many techniques exploit narrow-width values to improve performance [10, 39, 40, 42, 48, 60, 73] or soft-error resilience [71]. The limitation of these techniques is that they may require speculating the width (e.g., [39, 42, 60]) and if the actual width is larger than the speculated width, or if the application has infrequent narrow values, a PR may not be available for allocation, which can lead to a deadlock.

Lipasti *et al.* [10] note that the instantaneous mapping between the LRs and PRs is stored in a map table and in processors with many PRs, the map table becomes large. In their technique, when a pointer to RF requires more bits than the width of register value, the value is stored in map table itself, which avoids redirection. For example, integer values with less than 8 bits and floating-point values with all 0's or 1's are stored in map table. They show that their technique can store a large fraction of register values in map table, which reduces pressure on RF and improves performance. They also combine their technique with early register release approach and show that it reduces the register lifetime further.

Ergin *et al.* [39] propose two policies for packing multiple narrow results into one PR to reduce RF size requirement. The first policy allots register portions based on speculated result width. If the actual width was found to be larger or smaller than the speculated width, the portions are reallocated or freed up, respectively. They also present strategies for alleviating deadlock when a PR cannot be found for an instruction with wrongly-predicted width. The second policy allots a full register for a result. If this is found to be narrow, it is instead allocated to portions of other registers, and the full register is freed. If no other register can accommodate this result, it remains in the original register, and the portions of this register are made available for other narrow values. They show that generally, the accuracy of width-prediction is high and hence, by virtue of providing more efficient use of register partitions, the first policy provides better performance than the second policy.

Kondo *et al.* [40] present a technique for exploiting narrow register values, which works by dividing a 64b K-entry RF into two 32b K-entry banks. Their technique allocates two PRs, one in each bank, for a value wider than 32b and one PR for a value which is at most 32b wide. This allows reusing 32b portions for storing other values. For different narrow values, one PR may be assigned in different banks to ensure uniform utilization. They show that their technique improves performance by reducing register pressure and saves energy by allowing use of a smaller RF.

4.6. *Leveraging register file virtualization*

In some techniques, physical RF is treated as a cache for the logical RF to provide illusion of larger RF capacity [1, 74]. Similarly, by temporarily decoupling a register identifier from the stored value, the effective RF capacity can be increased [25]. We now discuss some techniques for RF virtualization.

Oehmke *et al.* [1] note that availability of a large number of LRs can boost performance of multiple thread contexts and function contexts; however, in traditional RF designs, every active context must be fully present in RF, which increases the requirement of physical RF size and imposes huge area/power costs. To address this, they propose virtualizing the LR contexts. Their architecture maps LRs storing local variables to a large memory address space, and the most recently used values are stored in the physical RF, which works as a cache. Thus, by virtue of using memory as a backing store, physical RF size becomes decoupled from number of LRs. Only the source operands and the destination register of an instruction need to be present in the RF for execution. An LR identifier is translated to a memory address and then mapped to a PR using a rename map table. Their architecture saves dormant register values in memory and restores them to RF on-demand. Hence, unlike in traditional RRS, in their design, an LR may not be mapped to a PR, and in such case, the value is retrieved from memory. And the LR is mapped to a free PR. If no free PR is found, the oldest register is replaced. While incurring the average access latency of a traditionally-sized RF, their design appears to provide much larger number of active contexts. Also, it reduces accesses to data cache and provides nearly optimal implementation of register windows.

Balkan *et al.* [25] note that even after deallocation of a PR, it still holds its value, which can be read by the dependent instructions before the PR is overwritten. Based on this, use of the produced register value and partial execution of the instruction, to which the same register is reassigned, can be overlapped. They propose two schemes, which use this insight, and, thus, work by temporarily decoupling a register value from its identifier. The first scheme deallocates a PR immediately after the result has been written to it and it has been redefined. This PR can be allocated to another instruction. This second instruction needs to stall at dispatch stage until the previous value stored in the PR has not been read by all its consumers. Till this time, the instructions that may get stalled are stored in a buffer, and then, they are moved to the issue queue. Use of the buffer avoids dispatch stall because the following independent instructions can execute in a normal manner. Their second scheme deallocates a PR when the next instruction writing to the same architectural register is renamed, irrespective of whether the value is generated or not. Thus, this scheme performs more aggressive deallocation. The instruction, which acquires the early-deallocated PR, progresses to writeback stage even if the consumers of the prior value have not issued. By this time, a normally deallocated PR may become available, and in this case, the early-deallocated PR is remapped to a normally deallocated one. They show that the second scheme provides larger performance improvement than the first scheme.

4.7. *Managing register file power consumption*

We now discuss RF power management techniques. Some of these techniques take an action *while an L1/L2 cache miss is being serviced*, for example, transitioning the RF to state-retentive mode [63], increasing RF size to avoid stall [43], and freeing the PRs allocated to corresponding instructions [41]. Also, separate RF banks can be reserved for load operations, and remaining banks can be power-gated during this time [62].

Roy *et al.* [63] note that in an in-order core, a miss event (e.g., cache miss) seen by an instruction prevents execution of any further instruction from that thread for the duration of the stall. They propose transitioning the RF of the thread in state-retentive mode [34] during the stall. They use two sleep-states, viz. sleep1, and sleep2 that consume 64% and 48% of normal leakage power, respectively, and have wakeup latencies of 3 and 5 cycles, respectively. On an L1 cache miss, the RF transitions to sleep1 and continues in this state in case of L2 cache hit. In case of L2 miss, the RF moves to sleep2 for the duration of L2 miss, and then the RF comes back to sleep1. On completion of L1 miss, RF returns to active state. The wakeup latencies are hidden by overlapping them with L1 fill latency and/or by use of multithreading (MT). They propose different techniques for coarse- grained MT, fine-grained MT and simultaneous MT architectures, which differ in actions taken on instruction fetch miss and data load miss. Their technique provides large leakage energy saving for coarse-grained ($\sim$40%) and smaller leakage energy saving ($\sim$8%) for both fine-grained and simultaneous MT architectures.

Homayoun *et al.* [43] note that an L2 cache miss stalls the processor since subsequent instructions cannot be committed before the completion of the miss. To reduce such stall periods, the size of RF, issue queue and reorder buffer needs to be increased; however, this increases their access latency and reduces the realizable processor frequency. To address this, their technique uses smaller RF size when there is no pending L2 miss and increases RF size while an L2 miss is being serviced. To still access the large RF in a single cycle, during L2 miss, the clock frequency is reduced, which incurs near-zero transition overhead. Their circuit design for performing RF size scaling incurs smaller overhead than clustering or banking schemes. Their technique improves performance and reduces energy-delay product.

Battle *et al.* [62] note that only a fraction of registers in processor RF are actually used and even these registers store valid value for a fraction of time. Based on these, they propose three register allocation strategies and three power-gating scheme to save RF leakage energy. While the conventional allocation strategy allocates registers in the order they become free, their first allocation strategy maintains a history of register allocation and allocates a register from the most recently used bank. On a cache miss, the register remains dormant until the the completion of miss access. Based on this, the second strategy reserves a part of RF for load operations, and thus, remaining banks need not be kept enabled for this duration. The third strategy begins allocating the register from the most-occupied bank. Their results show that both conventional and first allocation strategy perform poorly as they scatter the active register in different banks, leaving little scope of power gating. The second strategy also performs poorly because it can only identify head of a long latency–dependency chain while neglecting dependent instructions, which also use RF. The third strategy is effective in disabling banks and saving leakage energy.

As for power-gating schemes, the first scheme disables a bank immediately when it becomes empty. The second scheme records the number of active banks in last 8 cycles and finds the highest of those eight values. Based on this, all enabled yet empty banks in excess are disabled. The third scheme disables banks based on ROB occupancy. When ROB entries are committed or squashed, empty banks are disabled and with increasing ILP, more banks are enabled. Because the first and the third schemes work aggressively and track RF usage pattern accurately, they disable the largest number of banks. By comparison, the second scheme disables smaller number of banks because it is slow in tracking RF usage pattern. They also study two schemes, which enable a bank 1 cycle and 5 cycles, respectively, after allocation of a register from it, because a register is written 6 cycles after allocation in writeback stage. They find that the second scheme reduces the overhead of power gating and saves larger amount of leakage energy.

Tabkhi *et al.* [64] propose a technique that power-gates passive registers at function granularity for saving static power. Using application binary analysis, their technique identifies all possible registers that may be active in each function. The remaining registers are candidates for power gating. Function-invocation semantics classify registers into scratch, call-preserved, and dedicated, of which their technique attempts power gating of only the first two categories. The information about power gating is instrumented in the binary, and power gating is applied during runtime. For recursive function calls, the instrumentation and power-gating commands found for the original function are reused, which avoids the overhead of redundant search. They show that their technique saves large energy with negligible performance loss.

5. IMPROVING SOFT-ERROR RESILIENCE

We now discuss techniques for protecting RF from soft-errors.

5.1. Duplicating register values

For recovering from soft-errors and unusual events (e.g., interrupts and power failure), some techniques backup registers in main memory (or protected storage) [27–29, 31, 46, 65, 67] or NVM registers [32]. Some techniques duplicate used registers into unused registers [58, 61, 69] or exploit existing duplicate copies [58]. Narrow values can be copied within a register itself [71]. Thus, if an error is detected by if ECC or parity-bit, the duplicate copy can immediately provide correct value [2], because simultaneous error in both copies is extremely rare. We now review techniques, which perform duplication for improving soft-error resilience.

Memik *et al.* [69] note that most applications do not actively use all the PRs. They present two techniques for duplicating actively used PRs in idle PRs. The first technique duplicates a PR only when an idle PR is found; thus, no duplication is performed for phases with high register requirement. The second technique flags a PR, which is unused for long time as dead and duplicates an active PR in it. They show that in error-free execution phases, the first technique does not harm performance, whereas the second technique incurs small (e.g., $< 0.5\%$) loss in performance. However, the fraction of accesses directed to reliable (i.e., those with duplicates) PRs is larger in second technique than in first technique, and thus, second technique is more effective in improving reliability.

Several embedded systems use both general-purpose and special-purpose registers. Tabkhi *et al.* [61] observe that many applications use general purpose registers intensely whereas special purpose registers remain idle over significant part of the execution. This makes general-purpose registers vulnerable and, hence, they propose duplicating them in idle registers for reducing RF vulnerability. Duplication is performed at function granularity. For similar reduction in RF vulnerability, their technique consumes much lower power than a partially-ECC protected RF design.

Hu *et al.* [71] propose duplicating narrow values within the register itself, for example, a value requiring at most 32 bits can be duplicated in a 64 bit register. For both 32-bit halves, a parity bit is used for error detection. Their approach incurs lower overhead (in terms of bandwidth, register renaming, etc.) than use of copy registers; however, it can only protect narrow values.

Eker *et al.* [58] note that because of value locality (§4.3), a large fraction (e.g., more than 60%) of registers may already have a replica within RF. Their reliability improvement technique tracks PRs with same value by comparing input registers and result value of anarithmetic logic unit operation. In addition, their technique duplicates live PRs into idle PRs for increasing the fraction of PRs with replicas. This reduces RF vulnerability.

5.2. Protecting selected registers

To avoid the overhead of full-protection, some techniques protect only selected or most vulnerable registers/variables [32, 55, 66–68]. We now discuss some techniques, which use this idea.

Montesinos *et al.* [55] note that only a few registers are susceptible to soft-errors and the vulnerable period of registers are much smaller compared with their lifetimes. Based on this, their technique stores and checks ECC for only the most vulnerable registers (e.g., those with large lifetimes) while they store critical data. Also, their technique reuses ECC hardware for generating and verifying 1-bit parity for all registers. Use of parity bit allows their technique to detect all single bit errors. They show that their technique has no performance penalty and consumes much smaller power compared with using ECC for all the registers.

Lee *et al.* [67] propose a compiler-based scheme for reducing RF vulnerability by backing up live registers in ECC-protected cache or memory. They use integer linear programming for finding code locations where backup/restore commands can be inserted to minimize RF vulnerability for a given performance bound and with minimal code alteration. Their scheme finds large vulnerable periods and the variables with large live-range and then chooses the best ones using a cost-benefit analysis. They also employ several optimizations for reducing code alteration overhead. Their technique brings large reduction in RF vulnerability.

5.3. Altering instruction scheduling

Some techniques alter instruction scheduling to reduce register vulnerable periods [66, 70] and increase effectiveness of bypass network [54]. We now review some techniques, which work by performing intelligent instruction scheduling.

Yan *et al.* [66] present two compiler techniques to reduce RF vulnerability to soft-errors. Because a register is vulnerable to soft-errors between a write and a read operation, their first technique schedules write operations as late as possible and read operations as early as possible. The performance loss is avoided by exploiting scheduling slack inferred from program dependence graph. For cases where only a fraction of registers are protected by ECC, their second technique controls register allocation such that the most vulnerable register values (found from profiling) are always mapped to PRs with ECC. This improves RF reliability significantly.

Xu *et al.* [70] present a compiler-based instruction rescheduling technique for reducing RF vulnerability. Their basic block scheduling scheme uses dynamic programming algorithm while accounting for instruction dependency constraint. Their technique rearranges the program control flow with a view to reduce the vulnerable periods of registers, for example, it moves the consumer instruction of an operand closer to its producer instruction. They also use an RRS for facilitating the scheduling by avoiding write-after-read dependencies. They show that their technique is effective in improving RF reliability.

6. CONCLUSION AND FUTURE OUTLOOK

In this paper, we presented a survey and classification of techniques for designing and managing register file in CPUs. We classified the techniques based on several parameters to bring out their key features. We conclude this paper with a brief mention of possible future work and challenges.

Many of the techniques presented here can be combined to bring their best together. For example, because write operations are a bottleneck for NVM, the techniques exploiting bypass network (§4.4) and narrow-width values (§4.5) can be used to reduce writes on NVM registers (§3.5). Further, RegCache reduces the accesses to RF and also reduces the impact of RF latency on processor performance (§3.3). This allows designing RF using soft-error immune circuits because their higher latency/power overheads can be easily tolerated because of the use of RegCache. Thus, RegCache-based designs can be used for improving soft-error resilience (§5), and similar idea also applies to HRF designs (§3.2) which allow designing R_{L2} with soft-error immune circuits. Similarly, instruction rescheduling, which has been used for improving soft-error resilience (§5.3), can also be used for reducing RF power consumption. For this purpose, the slack between generation and consumption of an operand can be reduced, which allows aggressively power gating the registers for saving leakage energy (§4.7).

It is also important to note that many of the techniques discussed here have conflicting optimization goals/approaches (also see §2.3.6 and §4.3). For example, allocating PRs from a single or few bank(s) allows power-gating banks for saving energy (§4.7); however, this increases contention for those banks and increases the requirement of ports. Thus, from performance perspective, allocating PRs from different banks is more beneficial. Similarly, duplicating register values for soft-error protection (§5.1) can harm performance and energy efficiency. Clearly, using multiple RF management techniques will require exercising a right trade-off between their costs and benefits.

We believe that given the tall performance and energy efficiency targets of next-generation computing systems, the challenges of RF optimization need to be simultaneously addressed at multiple abstraction levels. At fabrication level, low-leakage memory devices and radiation-hardened circuits can lower energy consumption and soft-error vulnerability. At architecture level, schemes such as data compression and low-voltage operation can be synergistically integrated with those discussed in this paper to further optimize bandwidth and energy. Finally, at SW and application level, altering memory layout and intelligent compiler optimizations can boost the efficacy of runtime techniques.

Massively multi-threaded processors such as GPUs use much larger RF than CPU, for example, Fermi GPU has 2-MB RF, which is even larger than its last level cache [12]. because of this, RF has much larger impact on performance and energy profile of GPUs. Because the architecture and design philosophy in CPUs and GPUs are vastly different, porting existing CPU RF management techniques to and designing novel techniques for GPUs will be an interesting research challenge.

Several applications are inherently error-tolerant and this, along with perceptual limitations of humans, provides scope for *approximate computing* whereby the computation accuracy can be traded to gain performance/energy [77]. This is in contrast to conventional *error-intolerant computing* approach, which can only offer limited gains because of the requirement of accurate computation. In near future, using approximation strategies in RF will be a promising approach. For example, scaling the precision of floating-point registers, storing approximate values in registers to avoid writeback operations, and aggressively scaling RF voltage supply to selectively allow some errors can provide large benefits in performance and energy efficiency with only minor and acceptable quality loss.

ACKNOWLEDGEMENTS

This material is based upon work supported by US Department of Energy, Office of Science, Advanced Scientific Computing Research.

REFERENCES

1. Oehmke DW, Binkert NL, Mudge T, Reinhardt SK. How to fake 1000 registers. *International Symposium on Microarchitecture* 2005;7–18. DOI: 10.1109/MICRO.2005.21.
2. Mittal S, Vetter J. A Survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems* 2016; **27**(4):1226–1238.
3. Control Data 7600 Computer System: Preliminary Reference Manual. *Technical Report*, Control Data Corporation, 1970.
4. Mainframe Subsystem Equipment Specification. *Technical Report 003106*, ETA Systems, 1987.
5. HEP Hardware Reference Manual. *Technical Report 9000003*, Denelcor, Inc., 1982.
6. Swensen JA, Patt YN. Hierarchical registers for scientific computers. *Acm international Conference on Supercomputing*, Saint Malo, France, 1988; 346–354.
7. Kessler RE. The alpha 21264 microprocessor. *IEEE Micro* 1999; **19**(2):24–36.
8. Intel. *Intel Itanium Processor 9500 Series Reference Manual*, 2012.
9. Russell RM. The CRAY-1 computer system. *Communications of the ACM* 1978; **21**(1):63–72.
10. Lipasti MH, Mestan BR, Gunadi E. Physical register inlining. *Acm sigarch Computer Architecture News, Vol. 32*, 2004; 325.
11. Cruz J-L, González A, Valero M, Topham NP. Multiple-banked register file architectures. *International Symposium on Computer Architecture* 2000;316–325. DOI: 10.1145/339647.339708.
12. Mittal S. A survey of techniques for architecting and managing GPU register file. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016. DOI: 10.1109/TPDS.2016.2546249.
13. Yan J, Zhang W. Exploiting virtual registers to reduce pressure on real registers. *ACM Transactions on Architecture and Code Optimization* 2008; **4**(4):3.
14. Naresh VRK, Palframan DJ, Lipasti MH. CRAM: coded registers for amplified multiporting. *International Symposium on Microarchitecture* 2011;196–205. DOI: 10.1145/2155620.2155643.
15. Kim NS, Mudge T. Reducing register ports using delayed write-back queues and operand pre-fetch. *International Conference on Supercomputing*, San Francisco, USA, 2003; 172–182.
16. Shioya R, Horio K, Goshima M, Sakai S. Register cache system not for latency reduction purpose. *International Symposium on Microarchitecture* 2010; 301–312. DOI: 10.1109/MICRO.2010.43.
17. Donkoh E, Ong TS, Too YN, Chiang P. Register file write data gating techniques and break-even analysis model. *International Symposium on Low Power Electronics and Design (ISLPED)* 2012:149–154. DOI: 10.1145/2333660.2333700.
18. Gonzales DR. Micro-RISC architecture for the wireless market. *IEEE Micro* 1999; **19**(4):30–37.

19. Gonzalez A, Gonzalez J, Valero M. Virtual-physical registers. *International Symposium on High-Performance Computer Architecture* 1998;175–184. DOI: 10.1109/HPCA.1998.650557.
20. Monreal T, González A, Valero M, González J, Viñals V. Delaying physical register allocation through virtual-physical registers. *International Symposium on Microarchitecture* 1999;186–192. DOI: 10.1109/MICRO.1999.809456.
21. Tseng JH, Asanović K. Banked multiported register files for high-frequency superscalar microprocessors. *ACM SIGARCH Computer Architecture News* 2003; **31**(2):62–71.
22. Seznec A, Toullec E, Rochecouste O. Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors. *International Symposium on Microarchitecture* 2002; 383–394.
23. Alastruey J, Monreal T, Viňals V, Valero M. Microarchitectural support for speculative register renaming. *International Parallel and Distributed Processing Symposium* 2007;1–10. DOI: 10.1109/IPDPS.2007.370237.
24. Ponomarev D, Kucuk G, Ergin O, Ghose K. Reducing datapath energy through the isolation of short-lived operands. *International Conference on Parallel Architectures and Compilation Techniques.* IEEE, New Orleans, Louisiana, USA, 2003; 258–268.
25. Balkan D, Sharkey J, Ponomarev D, Aggarwal A. Address-value decoupling for early register deallocation. *International Conference on Parallel Processing.* IEEE, Columbus, Ohio, USA, 2006; 337–346.
26. Ayoub Raid, Orailoglu Alex. Power efficient register file update approach for embedded processors. *International Conference on Computer Design.* IEEE, Lake Tahoe, CA, USA, 2007; 431–437.
27. Martínez JF, Renau J, Huang MC, Prvulovic M. Cherry: checkpointed early resource recycling in out-of-order microprocessors. *International Symposium on Microarchitecture* 2002; 3–14. DOI:10.1109/MICRO.2002.1176234.
28. Jones TM, O'Boyle MFP, Abella J, González A, Ergin O. Exploring the limits of early register release: Exploiting compiler analysis. *ACM Transactions on Architecture and Code Optimization* 2009; **6**(3):12.
29. Balkan D, Sharkey J, Ponomarev D, Ghose K. Selective writeback: reducing register file pressure and energy consumption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2008; **16**(6):650–661.
30. Balasubramonian R, Dwarkadas S, Albonesi DH. Reducing the complexity of the register file in dynamic superscalar processors. *International Symposium on Microarchitecture* 2001;237–248. DOI: 10.1109/MICRO.2001.991122.
31. Quiñones E, Parcerisa J-M, González A. Leveraging register windows to reduce physical registers to the bare minimum. *IEEE Transactions on Computers* 2010; **59**(12):1598–1610.
32. Yang C, Ruiz Varela M. Qualifying non-volatile register files for embedded systems through compiler-directed write minimization and balancing. *International Conference on Very Large Scale Integration (VLSI-SOC)*, IEEE, Daejeon, South Korea, 2015; 86–91.
33. Tran L, Nelson N, Ngai F, Dropsho S, Huang M. Dynamically reducing pressure on the physical register file through simple register sharing. *International Symposium on Performance Analysis of Systems and Software* 2004;78–87. DOI: 10.1109/ISPASS.2004.1291358.
34. Mittal S. A survey of architectural techniques for improving cache power efficiency. *Elsevier Sustainable Computing: Informatics and Systems* 2014; **4**(1):33–43.
35. Mittal S, Vetter JS. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016; **27**(5):1537–1550.
36. Mittal S. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys* 2016.
37. Mittal S, Vetter J. A survey of architectural approaches for data compression in cache and main memory systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016; **27**(5):1524–1536.
38. Butts JA, Sohi GS. Use-based register caching with decoupled indexing. *International Symposium on Computer Architecture (ISCA)*, Munich, Germany, 2004; 302.
39. Ergin O, Balkan D, Ghose K, Ponomarev D. Register packing: exploiting narrow-width operands for reducing register file pressure. *International Symposium on Microarchitecture* 2004;304–315. DOI: 10.1109/MICRO.2004.29.
40. Kondo M, Nakamura H. A small, fast and low-power register file by bit-partitioning. *International Symposium on High-Performance Computer Architecture* 2005;40–49. DOI: 10.1109/HPCA.2005.3.
41. Sharkey J, Ponomarev D. An L2-miss-driven early register deallocation for SMT processors. *International Conference on Supercomputing*, Seattle, WA, USA, 2007; 138–147.
42. Wang S, Yang H, Hu J, Ziavras SG. Asymmetrically banked value-aware register files. *IEEE Computer Society Annual Symposium on VLSI* 2007:363–368. DOI: 10.1109/ISVLSI.2007.27.
43. Homayoun H, Pasricha S, Makhzan M, Veidenbaum A. Dynamic register file resizing and frequency scaling to improve embedded processor performance and energy-delay efficiency. *Design Automation Conference*, Anaheim, CA, USA, 2008; 68–71.
44. Jourdan S, Ronen R, Bekerman M, Shomar B, Yoaz A. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. *International Symposium on Microarchitecture* 1998;216–225.
45. Jones TM, O'Boyle MFP, Abella J, González A, Ergin O. Energy-efficient register caching with compiler assistance. *ACM Transactions on Architecture and Code Optimization* 2009; **6**(4):13.
46. Ergin O, Balkan D, Ponomarev D, Ghose K. Early register deallocation mechanisms using checkpointed register files. *IEEE Transactions on Computers* 2006; **55**(9):1153–1166.
47. Duong N, Kumar R. Register multimapping: a technique for reducing register bank conflicts in processors with large register files. *Symposium on Application Specific Processors (SASP)* 2009; 50–53. DOI: 10.1109/SASP.2009.5226335.

48. Sirsi S, Aggarwal A. Exploring the limits of port reduction in centralized register files. *International Conference on VLSI Design*, New Delhi, India, 2009; 535–540.
49. Sangireddy R, Somani AK. Exploiting quiescent states in register lifetime. *International Conference on Computer Design*, San Jose, California, USA, 2004; 368–374.
50. González R, Cristal A, Ortega D, Veidenbaum A, Valero M. A content aware integer register file organization. *International Symposium on Computer Architecture* 2004; 314–324. DOI: 10.1109/ISCA.2004.1310784.
51. Perais A, Seznec A. Cost effective physical register sharing. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016; 694–706.
52. Abella J, González A. On reducing register pressure and energy in multiple-banked register files. *International Conference on Computer Design*, San Jose, California, USA, 2003; 14–20.
53. Nalluri R, Garg R, Panda PR. Customization of register file banking architecture for low power. *International Conference on VLSI Design*, Bangalore, India, 2007; 239–244.
54. Park S, Shrivastava A, Dutt N, Nicolau A, Paek Y, Earlie E. Bypass aware instruction scheduling for register file power reduction. *ACM SIGPLAN Notices,* Vol. 41, Ottawa, Canada, 2006; 173–181.
55. Montesinos P, Liu W, Torrellas J. Using register lifetime predictions to protect register files against soft errors. *International Conference on Dependable Systems and Networks*, Edinburgh, Scotland, 2007; 286–296.
56. Park I, Powell MD, Vijaykumar T. Reducing register ports for higher speed and lower energy. *International Symposium on Microarchitecture* 2002;171–182. DOI: 10.1109/MICRO.2002.1176248.
57. Guan X, Fei Y. Register file partitioning and compiler support for reducing embedded processor power consumption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2010; **18**(8):1248–1252.
58. Eker A, Ergin O. Exploiting existing copies in register file for soft error correction. *Computer Architecture Letters* 2016; **15**(1):17–20.
59. Lee J, Shrivastava A. A compiler-microarchitecture hybrid approach to soft error reduction for register files. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2010; **29**(7):1018–1027.
60. Aggarwal A, Franklin M. Energy efficient asymmetrically ported register files. *International Conference on Computer Design*, San Jose, CA, USA, 2003; 2–7.
61. Tabkhi H, Schirner G. A joint SW/HW approach for reducing register file vulnerability. *ACM Transactions on Architecture and Code Optimization (TACO)* 2015; **12**(2):9.
62. Battle SJ, Hempstead M. Register allocation and VDD-gating algorithms for out-of-order architectures. *International Conference on Computer Design (ICCD)*, Asheville, NC, USA, 2013; 108–114.
63. Roy S, Ranganathan N, Katkoori S. State-retentive power gating of register files in multicore processors featuring multithreaded in-order cores. *IEEE Transactions on Computers* 2011; **60**(11):1547–1560.
64. Tabkhi H, Schirner G. Application-guided power gating reducing register file static power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2014; **22**(12):2513–2526.
65. Battle S, Hilton AD, Hempstead M, Roth A. Flexible register management using reference counting. *International Symposium on High Performance Computer Architecture* 2012;1–12. DOI: 10.1109/HPCA.2012.6169033.
66. Yan J, Zhang W. Compiler-guided register reliability improvement against soft errors. *ACM international conference on Embedded Software*, Jersey City, NJ, USA, 2005; 203–209.
67. Lee J, Shrivastava A. A compiler optimization to reduce soft errors in register files. *ACM sigplan notices,*Vol. 44, 2009; 41–49.
68. Lee J, Shrivastava A. Static analysis of register file vulnerability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2011; **30**(4):607–616.
69. Memik G, Kandemir MT, Ozturk O. Increasing register file immunity to transient errors. *Design, Automation and Test in Europe (DATE)* 2005;586–591. DOI: 10.1109/DATE.2005.181.
70. Xu J, Tan Q, Zhou H. Scheduling instructions for soft errors in register files. *International Conference on Dependable, Autonomic and Secure Computing (DASC)*, Sydney, NSW, Australia, 2011; 305–312.
71. Hu Jie, Wang Shuai, Ziavras Sotirios G. On the exploitation of narrow-width values for improving register file reliability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2009; **17**(7):953–963.
72. Monreal T, Viñals V, González A, Valero M. Hardware schemes for early register release. *International Conference on Parallel Processing (ICPP)*, Vancouver, British Columbia, Canada, 2002; 5–13.
73. Balakrishnan S, Sohi GS. Exploiting value locality in physical register files. *International Symposium on Microarchitecture* 2003;265–276. DOI: 10.1109/MICRO.2003.1253201.
74. Postiff M, Greene D, Raasch S, Mudge T. Integrating superscalar processor components to implement register caching. *International Conference on Supercomputing*, Sorrento, Naples, Italy, 2001; 348–357.
75. Ergin O, Balkan D, Ponomarev D, Ghose K. Increasing processor performance through early register release. *International Conference on Computer Design*, San Jose, CA, USA, 2004; 480–487.
76. Mittal S, Vetter JS, Li D. A survey Of architectural approaches for managing embedded DRAM and Non-volatile On-chip Caches. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 2016; **26**(6):1524–1537.
77. Mittal S. A survey of techniques for approximate computing. *ACM Computing Surveys* 2016; **48**(4):62:1–62:33.