PresCount: Effective Register Allocation for Bank Conflict Reduction

Xiaofeng Guan[†‡], Hao Zhou[‡], Guoqing Bao[‡], Handong Li[†], Liang Zhu[†], and Jianguo Yao[†‡]

[†]*Shainghai Jiao Tong University, Shanghai, China*
[‡]*Shanghai Enflame Technology Co. Ltd, Shanghai, China*
[†](guanxiaof,handong-lee,liang.z,jianguo.yao)@sjtu.edu.cn
[‡](vincent.zhou,guoqing.bao)@enflame-tech.com

***Abstract*—Modern processors with large multi-banked register files often rely on hardware solutions to resolve bank conflicts efficiently. However, these hardware-based methods, while flexible, can incur runtime penalties and restrict the exploration of optimized hardware designs. In contrast, compiler-based methods for register bank assignments avoid runtime overhead. However, incorporating bank assignment into the complex register allocation process presents significant challenges, leading existing methods to adopt conservative approaches to avoid potential side effects.**

This paper introduces the novel register allocation method *PresCount*, which enhances the coloring strategy for the Register Conflict Graph (RCG) and incorporates a bank pressure tracking mechanism to improve performance. The integrated register bank assigner in *PresCount* effectively reduces bank conflicts, achieving remarkable reductions of 43.28% and 27.76%, respectively, compared to existing methods on platforms with rich register banks and limited register budgets, as demonstrated by SPECfp and CNN-KERNEL benchmarks.

Furthermore, a subgroup splitting technique is introduced to facilitate register allocation under the bank-subgroup register file design, specifically our Domain-Specific Architecture (DSA) for AI computing. This technique demonstrates an impressive 99.85% reduction in bank conflicts for domain-specific kernel functions.

By addressing the challenges of bank conflicts in register allocation, the proposed *PresCount* method showcases significant improvements in performance and efficiency for platforms with different register configurations and domain-specific workloads, allowing for more flexible exploration of optimized hardware designs.

***Index Terms*—compiler, register allocation, bank conflict**

I. Introduction

Modern processors, including CPU systems, increasingly rely on the large register file to implement vector registers for Instruction Set Architecture (ISA). Together with Single Instruction Multiple Data (SIMD), CPU throughput can be improved significantly. Similarly, GPU systems utilize the large register file to support the Single Instruction Multiple Thread (SIMT) executions, thus enhancing Thread-Level Parallelism (TLP). Meanwhile, given that Artificial Intelligence (AI) computation typically requires a larger data block, it is possible to incorporate the large register file to improve performance.

The large register file is often implemented as multi-banked on-chip storage [1]–[6], but comes with side effects of bank conflict especially during simultaneous access to a single register bank. For instance, executing an instruction with two input register operands from the same bank can result in a bank conflict if the register bank has only a single read port. Therefore, hardware and/or software must handle the conflicts properly to ensure the functionalities of the computing system.

Different hardware and/or software methods have been proposed to resolve the conflicts. For example, hardware-based methods serialize conflicting register accesses through buffering the acquired data to meet the computational demand of the Arithmetical Logic Unit (ALU); GPU utilizes arbitrator and operand-collector [1], [6] to achieve the intended result. However, an additional delay of N-1 cycles will be introduced to ensure N-conflicted register accesses during runtime. In contrast, software methods, e.g., compilers, utilize a static approach that does not necessarily incur costs at runtime. It is based on the fact that the bank conflict imposes a unique restriction on the instruction operand usage. Inspired by the well-known graph coloring approach [7], most prior compiler implementations construct the Register Conflict Graph (RCG) for bank assignment through the coloring process [3], [8]–[10]. It is challenging to discover a conflict-free assignment if the RCG is hard to be colored. As a result, resolving bank conflicts still relies on sophisticated hardware designs.

To mitigate the penalties, different hardware enhancements have been proposed. For instance, the Register File Cache (RFC) [5], [11], which is designed for exploiting the temporal locality of register access, eliminates bank conflict penalties through optimization of pre-fetched instruction operands. Alternatively, interleaving register indexes across banks yields improved bank conflict ratios compared to non-interleaved setups. Another straightforward method is to increase the number of register banks, but it leads to increased costs in physical implementation [4]. The side effects for most of the hardware improvements lie in extra runtime overhead, increased complexity of the micro-architecture, and/or limited exploration space.

Meanwhile, reducing bank conflicts during compile time is advantageous, as it avoids introducing runtime overheads. In the experiment, we observed that the bank conflict-related instructions occurred $56.37\%/85.48\%$ of cases in SPECfp and CNN-KERNEL benchmarks during constructing a multi-

banked floating-point register. Notably, Intel's GPU research reported a similar result [12]. Given this pervasive occurrence, there is an increasing demand for employing static methods to reduce bank conflicts considering that peak performance and performance per watt are both crucial in the design of modern processors [2].

Unfortunately, previous compiler researches that are capable of statically reducing bank conflicts have different drawbacks. One of the problems is that they rely on register splitting to tackle the RCG colorability issue, but the splitting requires register copies and can trigger register spilling, while it is sometimes, the splitting choices are hard to make, leading the systems inapplicable. Another notable drawback lies in the absence of consideration for register live ranges during register bank assignments. This oversight can increase register pressure, which interferes with the register allocation (RA) process [8]. And as register spillings are commonly regarded as more expensive than bank conflicts, it is necessary to develop a more efficient bank allocation method for multi-banked register file systems.

In this study, we present *PresCount*, a novel RCG bank assignment method that builds upon LLVM's default register allocator, addressing limitations seen in prior work. Unlike the previous RCG-based approaches focusing on register splitting, our method incorporates cost estimation and register pressure modeling during the bank assignment phase. It results in a substantial reduction in bank conflicts while having minimal impact on the existing LLVM Register Allocation (RA) pipeline, as demonstrated through experimentation on public benchmarks. We further extended the validation of *PresCount* to an AI Domain Specific Architecture (DSA) equipped with a large multi-banked register file [13]. This DSA employs a two-level bank-subgroup design and imposes additional constraints on register usage. With the inclusion of an additional subgroup splitting mechanism, *PresCount* still distributed registers evenly among banks and subgroups. It finally achieved a significantly lower bank conflict ratio compared to the conventional solution.

We have summarized our major contributions below:

- We proposed a novel register bank assignment method capable of cost estimation and register pressure modeling, which is able to significantly reduces the bank conflict ratio in different systems under the large banked register files while having minimal impact on the existing RA pipeline for the compiler. The experiment demonstrated that the bank conflicts can be reduced by 43.28% and 27.76%, respectively, on SPECfp and CNN-KERNEL benchmarks compared with the existing method.
- We applied the proposed method on our DSA specifically designed for AI computing and extended it for subgroup splitting in a multi-banked and sub-grouped register file architecture. The method is able to optimize unbalanced register assignments despite more constraints on the hardware and demonstrated a 99.85% reduction of bank conflicts on the DSA.

II. Background and Motivation

In this section, we first discuss register bank conflict in more detail and demonstrate its pervasiveness by constructing a large multi-banked register file in our experiments. Then we introduce the traditional compiler register allocation and its relationship with the RCG-based bank assignment. We will show examples of the limitation of current RCG-based bank assigners.

A. Bank Conflicts in the Construction of Large Register File

The multi-bank architecture is employed to enhance the efficiency of computer memory and is also extended to register hardware, providing improved efficiency and performance to the hardware. However, this design also has weaknesses. In particular, the use of hardware to mitigate bank conflicts or penalties adds to the complexity, power consumption, and latency challenges of hardware design.

In various architectures, a bank conflict, which is a hardware resource hazard during execution, can often be statically determined. To illustrate, let us consider the following instruction in the intermediate representation (IR) form:

$$vr_a \ = \ operation \ \ vr_b, \ \ vr_c$$

When the compiler assigns virtual register operands vr_b and vr_c from the same bank, a bank conflict can arise at runtime if the hardware microarchitecture lacks support for parallel reads from a single bank. To avoid this conflict, the compiler can assign vr_b and vr_c to registers from different banks, effectively resolving the issue. Consequently, the *bank assignment* process plays a crucial role in minimizing such conflicts.

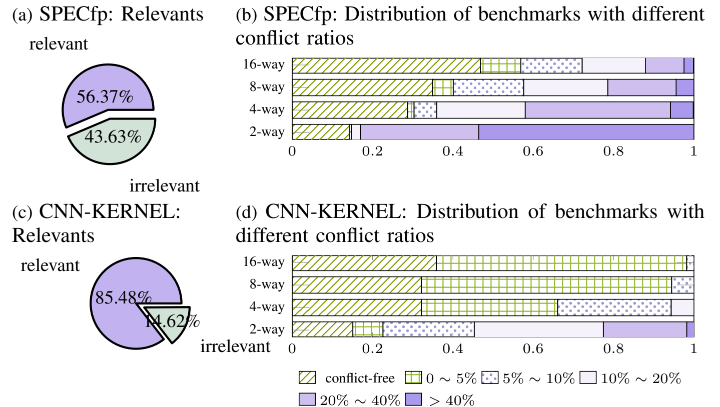


Figure 1: The proposition of workload with different bank conflict in ratios. An "N-way" represents the data of an N-way interleaving register file.

Bank conflicts are ubiquitous in the large register file if no bank assignment is involved. We conducted several preliminary experiments on SPECfp and CNN-KERNEL cases using Clang/LLVM compiler as shown in Figures 1a and 1c which demonstrated the prevalence of bank conflicts in various

programs. Given that not all programs contain bank conflict instructions, we categorized the programs as follows:

- **Conflict-irrelevant** denotes a program containing no bank conflict instructions.
- **Conflict-relevant** denotes a program containing at least one bank conflict instruction.
- **Conflict-free** denotes a *Conflict-relevant* program that does not trigger a bank conflict during runtime.
- **Conflict** denotes a *conflict-relevant* but not *conflict-free* program.

As shown in Fig. 1a, 177 out of 314 (56.37%) SPECfp tests were *conflict-relevant*, whereas 53 out of 64 (85.48%) were *conflict-relevant* for CNN-KERNEL tests as appeared in Fig. 1c. The preliminary statistics illustrated that bank conflict instructions are prevalent in both general computing and AI computing applications, while the latter is more prominent.

Figures 1b and 1d illustrated *conflict-relevant* when compared to the *conflict-free* programs in four distinct register files, i.e., interleaved register indexes in their 2/4/8/16 banks. The data indicated that 50.29% to 71.18% of the programs are not *conflict-free* in the SPECfp set and 64.15% to 84.91% in CNN-KERNEL tests. Moreover, it is still difficult to achieve *conflict-free* even for the hardware system with a large number of register banks, which is normally considered to be able to handle bank conflicts very effectively. Therefore, it is very desirable to design a more efficient bank assignment method based on the existing compiler techniques for bank conflict reduction.

B. Unbalanced Bank Assignment

The coloring-based assignment of register banks resembles the assignment of register coloring. It constructs the *Register Conflict Graph (RCG)*, where the set of edges E represents the conflicts between register banks in the graph $G_{RCG} = (V, E)$, and the set of vertices V includes all registers along with their bank information.

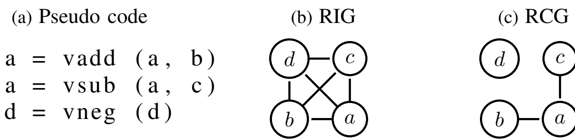


Figure 2: Example code with its corresponding RIG and RCG

Given a code snippet shown in Fig. 2a, the compiler can construct a corresponding Register Interference Graph (RIG) as shown in Figure 2b. Additionally, an associated Register Conflict Graph (RCG) is shown in Fig. 2c, which happens to be a subgraph of the RIG, representing the bank conflicts between registers. RCG-based bank assignment could be employed in the register allocation pipeline to apply bank conflicts reduction. If the bank assignment phase precedes the register allocation, it maps virtual registers to specific banks, limiting their subsequent allocation to registers within the assigned bank. This introduces constraints on register allocation and makes conflict-free assignments more challenging.

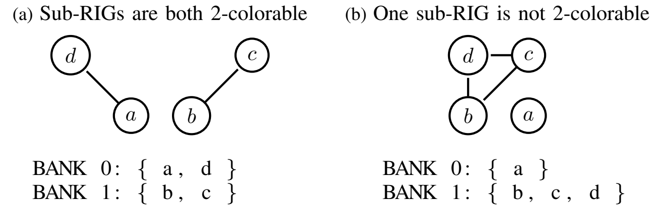


Figure 3: Two possible bank assignment and the colorabilities of corresponding sub-RIGs of Figure 2b

Assuming four registers with interleaved indexes distributed across two banks (`BANK0` containing `r0, vr2` and `BANK1` containing `r1, vr3`), performing a bank assignment is equivalent to a 2-coloring of the RCG shown in Figure 2c. After the bank assignment, the RIG simplifies since registers in different banks no longer interfere with each other. Figure 3a and Figure 3b display two possible bank assignments and their corresponding simplified RIGs. Allocating the RIG depicted in Figure 3a is feasible without bank conflicts due to the sufficient number of registers in each bank. However, Figure 3b is not 2-colorable, indicating that a conflict-free bank assignment is not possible without splitting/spilling of registers.

It is noteworthy that the original RIG can be colored using four colors. However, as the above example shows, a conflict-free bank assignment poses a challenge in coloring the RIG. This challenge, known as an "**unbalanced**" bank assignment, presents a severe problem in existing RCG-based bank assigners, rendering them inapplicable in certain scenarios. The evident influence of the bank assignment phase on register allocation necessitates the design of a bank assigner like *PresCount* that minimizes the negative impact on register assignment.

In the following section, we provide a detailed description of *PresCount*. We first introduce the overall workflow of our register allocation pipeline. Next, we present our approach to model and resolve bank conflicts. Finally, we address the subgroup constraints in the custom DSA design and propose a solution that extends bank conflict reduction on DSA.

III. Methodology

A. Overall workflow

The overall workflow of our register allocation pipeline is shown in Fig. 4. We proposed novel methods to improve LLVM's default register allocator (i.e., the greedy register allocator) for register bank assignment. To be more applicable in the real-world scenarios, we applied the improved register allocation pipeline on our DSA hardware. Given that the custom DSA hardware has a stricter constraint *subgroup alignment* for register usage (explain in detail later), an optional phase (dashed module) is therefore proposed for subgroup splitting for the DSA optimization.

As shown in Fig. 4, there are five phases in the entire pipeline including two standard components of LLVM, i.e., *Register Coalescing* and *Pre-allocation Scheduling* denoted in

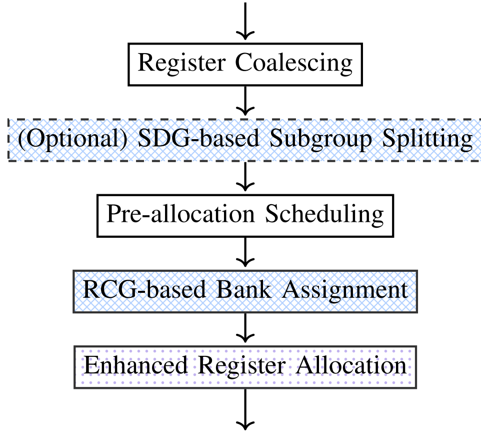


Figure 4: The combined workflow into LLVM *Greedy* register allocation pipeline

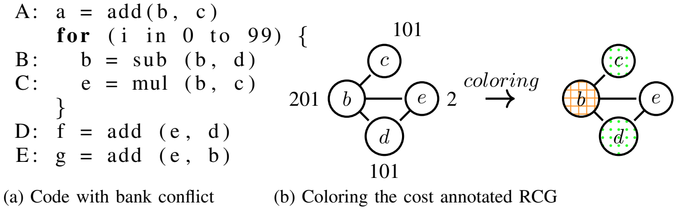
white, an *Enhanced Register Allocation*, and two new phases as denoted in blue modules:

- **RCG-based Bank Assignment** is an optimized phase of building RCG and performing graph coloring which is served to track the pressure of register banks. The phase is triggered right after *pre-allocation Scheduling*, where a bank assignment of its dedicated register(s) can be decided after this phase.
- **SDG-based Subgroup Splitting** is specifically designed for the custom DSA hardware, which takes LLVM Machine IR produced by *Register Coalescing* for building the *Same Displacement Graph (SDG)* (explain later), and collects information of "register grouping" and applies heuristic methods to split large groups when necessary. This module is optional for non-DSA hardware.

where, *RCG-based Bank Assignment* is designed to track the register bank pressure with a coarse pattern to improve unbalanced bank assignment, which is crucial to our bank assigner design. Such a design is inspired by the naive register pressure tracking approach with pre-allocation instruction schedulers.

The *Enhanced Register Allocation* is used to perform register assignment in accordance with the bank assignments decided by the *RCG-based Bank Assignment* phase. If it targets on the custom DSA, the *Enhanced Register Allocation* phase would additionally conduct subgroup assignments by taking into account of the register subgroup pressure.

In the register allocation (RA) pipeline, phase ordering plays a crucial role. To prevent re-coalescing of register copies, the *SDG-based subgroup splitting* phase is positioned after *Register Coalescing*. Moreover, to maximize the reuse of existing live range information, the *RCG-based Bank assignment* phase is placed between *pre-allocation scheduling* and *Enhanced Register Allocation*, as it does not modify the information.



(a) Code with bank conflict (b) Coloring the cost annotated RCG

Figure 5: Pseudo Code with bank conflicts and the relating 2-coloring process

B. Implement the Bank Assignment

Bank Conflict Cost Estimation. The cost of bank conflicts depends on the code's position. Bank conflicts within a loop have a higher impact on performance and energy consumption compared to conflicts in infrequently executed loops.

In *PresCount*, the cost $Cost_I$ of a bank conflict introduced by instruction I is defined in Equation 1, where $trip_count(i)$ represents the total trip count of its i-th enclosing loop within an n-level loop nest ($i \geq 0$, $n \geq 0$). The cost $Cost_R$ of a bank conflict for register R is calculated by summing the costs of all instructions that access register r, as showed in Equation 2. *PresCount* utilizes $Cost_R$ value as one judgement to do prioritized bank assignment, in order to minimize the runtime instances of bank conflicts.

$$\text{Cost}_I = \prod_{i=1}^{n} \text{trip_count}(i), \quad \text{where } n \text{ is the nest level of loops} \tag{1}$$

$$\text{Cost}_R = \sum_{i=0}^{n} \text{Cost}_i(R), \text{where } n \text{ is the number of } I \text{ instances} \tag{2}$$

RCG coloring with annotated Costs After annotating registers with conflict costs, the bank assignment process begins. Unlike the traditional Chaitin-Briggs coloring method, our approach generates a prioritized *AssignList* primarily based on conflict costs. For example, in Figure 5a, we have five *conflict-relevant* instructions (A-E) with their corresponding RCG in Figure 5b. Nodes in the graph represent registers with computed conflict costs. Assuming a 2-bank register file, a 2-coloring is performed on the RCG based on the decreasing order of register conflict costs, namely $b \to c \to d \to e$, as depicted in Figure 5b.

Ultimately, the remaining bank conflict introduced by accessing register e cannot be eliminated, as it is uncolorable. If we assume that each conflict produces a 1-cycle delay in the hardware, it incurs a total overhead of 2 cycles. This represents the minimum cost introduced when no other optimizations are applied.

Bank Pressure Counting Heuristic In practice, it is common to encounter equal conflict costs, especially when bank conflicts are predominantly associated with hot arithmetic operations within the innermost loop. To optimize bank assignments and ensure the colorability of RIG subgraphs (Chapter II-B), the concept of bank pressure count is introduced. This count represents the maximum overlap of register live ranges within a bank, considering the registers already assigned to that bank.

Algorithm 1: Bank Assignment in *PresCount*

```
Data: A RCG with conflicting registers, a bank conflict cost
      estimation
Result: A bank assignment of all registers in the RCG
unProcessed ← bank conflicting registers;
while unProcessed ≠ ∅ do
    workList ← {MaxConflictCost(unProcessed)};
    while workList ≠ ∅ do
        v ← MaxCostDegree(workList);
        workList ← workList \ {v};
        unProcessed ← unProcessed \ {v};
        availColors ← ALLCOLORS \ NeighborColors(v);
        if availColors ≠ ∅ then
            colors ← PresCountPrioritize(availColors);
        else
            regPressure ← OverallRegPressure();
            if regPressure > THRES then
                colors ←
                  PresCountPrioritize(ALLCOLORS);
            else
                colors ←
                  NeighbourCostPrioritize(ALLCOLORS);
            end
        end
        color ← Front(colors);
        AssignColor(v, color);
        workList ← workList ∪ UncoloredNeighbors(v);
    end
end
```

It serves as a guide for the bank assigner in scenarios where there are equal choices based on RCG colorability. When a register can be assigned to multiple banks, the bank with the minimal pressure count is selected. This approach effectively resolves uncolorable cases discussed in Chapter II-B.

Algorithm for the bank assignment Algorithm 1 provides a detailed illustration of the RCG-based bank assignment process. In practice, the RCG consists of multiple disjoint sub-graphs. We process each subgraph in descending order of conflict cost. The graph coloring starts by selecting a node from the $workList$ based on higher conflict cost, followed by a higher degree order. This prioritization ensures that *PresCount* addresses bank conflict cost before considering RCG colorability.

The algorithm then excludes any conflicting colors assigned to the node's neighbors, creating a prioritized assignment list (*colors*) based on the bank pressure count. As mentioned, an unbalanced bank assignment in the RCG may prevent the later register allocator from completing a conflict-free bank assignment, rendering the bank assignment unexpected. Improper handling of bank assignments can also trigger register spilling during implementation. Thus, it is crucial to evaluate the bank pressure dynamically to ensure a proper assignment. The algorithm tests all possible bank assignments and chooses the bank that adds the least to the bank pressure count for the register.

Uncolorable nodes are assigned conflicting colors using heuristics. To trade-off between minimizing register pressure and reducing bank conflict cost, a comprehensive evaluation of the overall register pressure ($regPressure$) comes into play. This assessment dictates that if the register pressure surpasses a predefined threshold ($THRES$), an optimal color is selected from all the candidates (ALLCOLORS) to minimize the register pressure. Or else, the heuristic choose the color that has the least accumulated $Cost_R$, with R encompassing all the neighboring nodes that share this color. It concentrates on the reduction of penalties related to bank conflicts. Then *PresCount* assigns the node with the decided color and updates bank pressure counts accordingly. Finally, any uncolored neighbor node is added to the $workList$. And the process continues iteratively until the $workList$ is empty.

It is worth noting that Algorithm 1 only handles registers present in the RCG. However, to achieve a balanced bank assignment, the bank assigner must also consider registers not present in the RCG but belonging to the same register class. These registers, often referred to as *free registers*, need to be properly handled to prevent the register allocator from assigning registers from random banks to them. Neglecting the proper handling of *free registers* can also lead to an unbalanced bank assignment.

C. Subgroup Bank Assignment for Custom DSA

Custom DSA and its constraint features for register usage The DSA discussed in this paper aims to improve the processing efficiency of AI models at a large scale. It addresses the temporal locality observed in AI operations by incorporating a large on-chip register file that enables the reuse of intermediate outputs. The micro-architecture design of the DSA is close to modern GPUs, but unlike the SIMT, it introduces SIMD vector instructions with large vector lengths to address the Instruction-Level Parallelism. Each processing element (PE) of the DSA has a 128k-bytes vector register file that is divided into 1024 128-bytes vector registers. Two simultaneous threads share each vector register in a lock-step, shared PC manner, effectively making the vector register appear as a 64-byte register for each thread. Similar to GPUs, the multiple-banked vector registers can experience bank conflicts when instructions read two vector register operands.

To avoid the complexity and power consumption associated with implementing a crossbar network for data routing between the register file and Arithmetic Logic Units (ALUs), a micro-architecture simplification is adopted. Instead of the crossbar, a direct 1-1 bank-to-ALU pattern is used with the primary target of reducing complexity and power consumption.

However, this hardware simplification further subdivides the multi-banked register file, which makes the register usage more restrictive. The register file is conceptually structured as a two-level "bank-subgroup" design, as shown in Figure 6 for the 2-4 bank-subgroup register file. The figure illustrates how the banks and subgroups are interleaved based on register indexes, along with the calculation method for determining the bank and subgroup number of a register based on its register number.

The bank-subgroup register file imposes two key constraints on register usage:

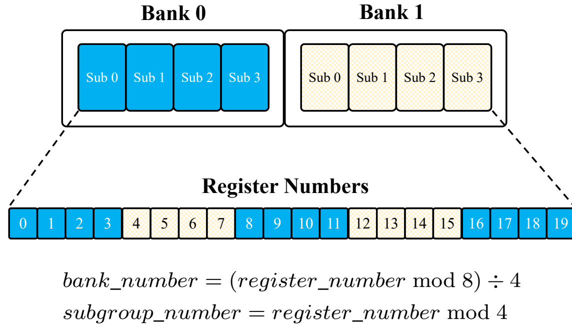
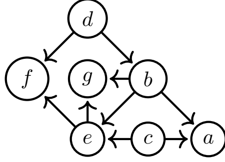


Figure 6: The register number interleaving of banks and subgroups

(a) SDG



(b) Psuedo code

```
I1: vr1 = fadd vr5, vr9
I2: vr1 = fadd vr5, vr13
I3: vr1 = fadd vr5, vr10
```

Figure 7: Bank Conflict and Subgroup Alignment

- The input operands of each instruction must belong to different banks, known as the *bank conflict constraint*.
- The subgroup of each instruction's operands must be identical, referred to as the *"**subgroup alignment**" constraint*.

For instance, consider the code in Fig. 7b. The bank/subgroup numbers for registers $vr1$, $vr5$, $vr9$, $vr10$, and $vr13$ are 0/1, 1/1, 0/1, 0/2, and 1/1, respectively. Based on this, we can analyze the following:

- Instruction I1 satisfies all the constraints and is considered *conflict-free*.
- Instruction I2 violates the *bank conflict constraint* as both of its input operands are from bank 1.
- Instruction I3 violates the *subgroup alignment constraint* because its input operands have different subgroup numbers.

Hence, as the register usage is more constrained, it is more prone to bank conflicts. To mitigate the challenge posed by a high register conflict ratio and eliminate subgroup alignment violations, the subsequent subsections present optimization methods that are necessary for addressing these issues.

SDG for the *Subgroup Alignment* The *subgroup alignment* constraint is represented using the *Same Displacement Graph (SDG)*. The SDG, denoted as $G_{SDG} = (V, A)$, is a directed graph where the vertex set V comprises all registers that require subgroup alignment, and the set A consists of ordered pairs representing alignment constraints among subgroups. The directed edges connect input operands to output operands, indicating that the connected vertices share the same displacement alignment requirement. Each edge points from an input operand to the corresponding output operand of the instruction.

Algorithm 2: Generating Register Allocation Hints to Ensure Bank Assignment and Subgroup Alignment Constraint

Data: Register bank assignment $registerBanks$, subgroup displacement assignment $groupDispls$, virtual register v
Result: Allocation hints $Hints$ for the virtual register v
if $v \notin registerBanks$ **then**
  /* Handle split-generated register */
end
$bank \leftarrow registerBank[v]$;
$subGroup \leftarrow$ FindASubGroupContains(v);
if $subGroup \in groupDispls$ **then**
  $displ \leftarrow groupDispls[subGroup]$;
else
  $displ \leftarrow groupDispls$[MinUsed(ALLSUBGROUPS)];
  $groupDispls[subGroup] \leftarrow displ$;
  /* Increase the usage of $subGroup$ by its size */
end
$Hints \leftarrow$ FindAllRegistersConforming($bank$, $displ$);

Balanced subgroup assignment Subgroup assignment is an integral part of the register allocation process. In our implementation, register bank assignments are determined before subgroup assignments. Algorithm 2 outlines the pseudo-implementation of subgroup assignment during register allocation using register hints. The algorithm monitors register assignments, tracks subgroup utilization, and guides the register allocator to assign registers with less-used subgroups. The balance of subgroups is maintained through the bookkeeping performed by $groupDispls$ and the selection of the minimum-used subgroup in the function MinUsed. The $Hints$ list for bank assignment contains physical register candidates derived from subgroup assignment. The existing register allocation mechanism utilizes $Hints$ while considering register live-range interference to complete the assignment. This ensures the satisfaction of bank conflict, subgroup, and live-range interference constraints.

Algorithm 2 does not provide details on other interactions within register allocation operations, such as register reassignment, register splitting, and register spill. Register splitting, in particular, can be problematic as it introduces new virtual registers with shorter live ranges. To maintain *subgroup alignment*, newly created registers must be assigned the same subgroup as the registers they are copied from, necessitating dynamic subgroup management in certain cases.

SDG-based subgroup splitting It is not uncommon to encounter large subgroups consisting of a significant number of registers. However, having such large subgroups often leads to unbalanced subgroup assignments. To address this issue, subgroup splitting is employed to avoid this situation. We have identified two primary schemes for large subgroups:

- *Input Sharing*: Operations that share the same input operand, such as the "a" operand in Figure 8.
- *Output Sharing*: Operations that involve value reduction and commonly share the output, as seen in the "a"

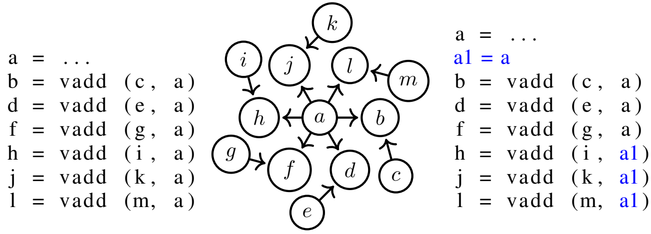


Figure 8: Example to resolve the "Input Sharing"

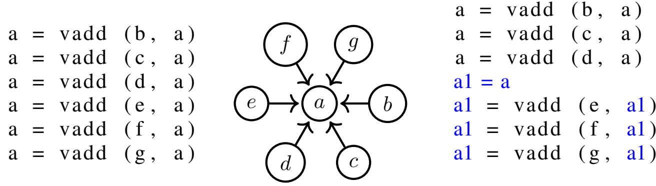


Figure 9: Example to resolve the "Output Sharing"

operand in Figure 9.

Figures 8 and 9 provide examples of resolving the scenarios of input sharing and output sharing, respectively. In both cases, a single subgroup encompasses all the registers. The input sharing scenario exhibits a "centered" vertex with a high number of outgoing edges, while the output sharing scenario features a "centered" vertex with a significant number of incoming edges.

To address the subgroup with a high number of outgoing edges (6) in Figure 8, a copy instruction is introduced to divide the graph into two groups, each consisting of three vertices. A similar splitting heuristic is applied as illustrated in Figure 9. This approach effectively creates balanced subgroups during the assignment process (Algorithm 2), ensuring even subgroup assignments.

IV. Experiments and Results

A. Experiment Settings and Metrics

1) Test suites: We evaluated our approach using three distinct test suites: SPECfp, CNN-KERNEL, and DSA-OP. SPECfp includes compute-intensive floating-point tests from the SPEC CPU benchmark, with a focus on eight tests primarily written in the C/C++ programming languages. The CNN-KERNEL test suite consists of 64 kernels from a Convolutional Neural Network called MobileNet, which includes commonly used AI computing kernels such as convolution, pooling, and activation. Finally, the DSA-OP test suite is designed to run on DSA hardware and contains C/C++ implementations of high-performance handwriting AI computing kernels, including element-wise, reduction, and special operations such as the *Inverse Discrete Fourier transform (IDFT)*.

Table I provides a summary of the test suites used in our evaluation, including information about executable generated, modules compiled, functions compiled, conflict-relevant instructions, and floating-pointer register spills using default

Table I: Characteristics of SPECfp and CNN-KERNEL

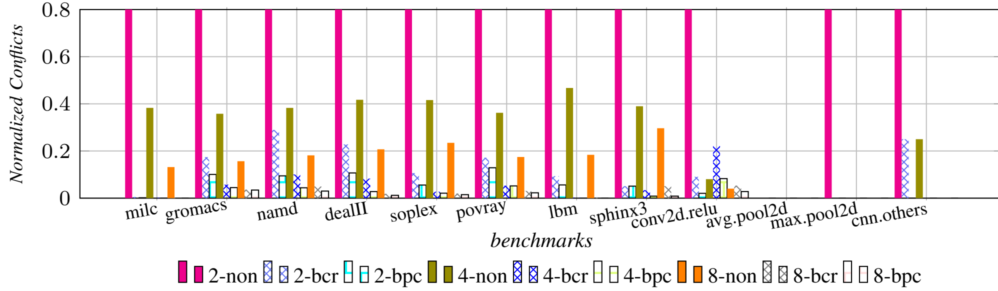
Benchmark	Exes	Mods	Fns	Reles	Sp32	Sp1k
SPECfp.433.milc	1	68	235	1,730	2	2
SPECfp.435.gromacs	1	131	925	10,143	130	145
SPECfp.444.namd	1	11	94	9,012	381	29
SPECfp.447.dealII	1	116	7,373	19,191	1,197	74
SPECfp.450.soplex	1	63	1,240	2,741	9	11
SPECfp.453.povray	1	100	1,537	19,749	171	173
SPECfp.470.lbm	1	2	17	672	0	0
SPECfp.482.sphinx3	1	44	318	361	0	0
CNN.conv2d.relu†	42	2	5	1,089	1.2	0.1
CNN.avg.pool2d†	6	2	6	1,010	0	0
CNN.max.pool2d†	6	2	5	326.8	0	0
CNN.other†	3	2	5	41.7	0	0

Exes - Executables. **Mods** - Modules. **Fns** - Functions. **Reles** - Conflict-relevant instructions. **Sp32** - Spillings on *RV-Platform#1*. **Sp1k** - Spillings on *RV-Platform#2*.
† Values in the columns (excluding "**Exes**") represent the geometric mean value of conflict-relevant executables. Conflict-irrelevant executables are excluded from the calculations.

LLVM register allocation. In contrast to SPECfp, the CNN-KERNELs are categorized based on the operation semantics of each program due to the large number of executables. Therefore, we report the geometric mean values to represent the CNN-KERNELs of the same operation. However, since most of the CNN-KERNELs consist of only a few floating-point arithmetic operations within multiple-level loop nests, it results in insufficient bank pressure. To address this, we manually unroll the loops to obtain different numbers of conflict-relevant instructions, thereby creating different levels of bank pressure.

2) Platforms: Two platforms, Platform-RV and Platform-DSA, were used for experiments with different objectives. Platform-RV was used to test the generality of the *PresCount* method and compare it with different implementations. Platform-DSA was used to test the *PresCount* method on a specific banked-and-subgrouped DSA register design to reveal how the method performs on a stricter register micro-architecture design. The following describes the RV platform in detail.

a) Platform-RV: has an LLVM/Clang (11.0.0) compiler equipped with our *PresCount* register allocation method that compiles programs to riscv-64 target executables. Two settings were used for the experiments. Setting #1 had 1024 floating-point registers, which were either 2, 4, or 8 banked, with 512, 256, and 128 registers for each bank. This setting is similar to some GPU architectures where a program can access rich registers in each bank. Setting #2 by default had 32 registers, which is consistent with riscv-64 Instruction Set Architecture(ISA). The assumed bank setting includes 2 and 4, with 16 and 8 registers in each bank. This setting covers situations where the register count is tight in a single bank, which is also observed in some GPU and CPU architectures. An LLVM pass was developed to collect static statistics of bank-conflict count, register spill count, etc., at compilation time. QEMU (7.0.50) simulator was used to run the riscv-64 executable. The trace generated by analyzing the trace reported



(a) Normalized bank conflicts with varying bank counts: The 'N' in the legends indicates the register file is N-banked.

benchmark	baseline
milc	645
gromacs	3618
namd	3934
dealII	7738
soplex	810
povray	7103
lbm	321
sphinx3	118
conv2d.relu	57.12
avg.pool2d	0
max.pool2d	13.57
cnn.other	9.03

(b) Maximum bank conflict count of SPECfp benchmarks

Figure 10: Static bank conflicts with different bank counts in SPECfp: Comparing *non*,*bcr*,*brc* and *bpc* on Platform-RV#1

dynamic instances of bank-conflict for Setting #2[1].

Four different register allocation methods were compared:

- *non*: Applies the default LLVM register allocation without mitigating register bank conflicts.
- *bcr*: Applies a greedy bank assignment method inside the LLVM register allocator by register hinting, partly mimicking the Intel Graphic Compiler's bank conflict reduction (bcr) method, but without consideration of inter-threading bank-conflict reduction.
- *bpc*: Applies bank assignment with our *PresCount* method for the register allocation. It applies the liveness-conflict-counting heuristic to track bank pressure at bank assignment.

Experiments performed on platform-RV collected the statistics including bank conflicts and register spills, in order to evaluate the effectiveness of bank-conflict mitigation, and also its impact on the existing register allocator.

b) Platform-DSA: also has LLVM/Clang (11.0.0) compilers with the *PresCount* method but including the SDG subgroup alignment functionality. It compiles high-performance AI kernels to work on real custom DSA hardware where its vector registers are 2-4 banked. We evaluate "non" and "brc" implementations to compare metrics including bank conflict, register copy, register spilling as well as precise cycles.

B. Results

1) Results of Plaform-RV#1: Figure 10a compares the normalized bank conflicts for various SPECfp and CNN-KERNEL benchmarks. The maximum static bank conflict count for each benchmark is shown in Figure 10b, with color-filled bars representing the results for 2/4/8-banked *non*. The graph clearly demonstrates a decrease in bank conflicts as the number of banks increases, indicating the effectiveness of the hardware method for reducing bank conflicts. The reduction follows a linear trend, with conflicts halving when the bank count is doubled. The cross-hatched bars represent the results of *bcr* register allocation, while the blocks bars represent *bpc*. Both methods effectively reduce bank conflicts across different bank settings compared to the baseline (*non*). Notably, CNN-KERNEL cases tend to exhibit higher reductions, which indicates that the methods perform well when dealing with tests that have small conflict-relevant instructions, as CNN-KERNEL cases have plenty of such cases.

[1]We limited the dynamic instance collection to Setting #2 since the register count is consistent between the compiler and simulator.

Table II: Comparing Conflicts and Reductions of RA Methods in the Combined SPECfp and CNN-KERNEL Benchmarks with Different Bank Settings

BANK	CONFS	Reduction		IMPV
		bcr	*bpc*	
2	33374	27777	30663	2886
4	10023	6616	8426	1810
8	4815	3684	4084	400

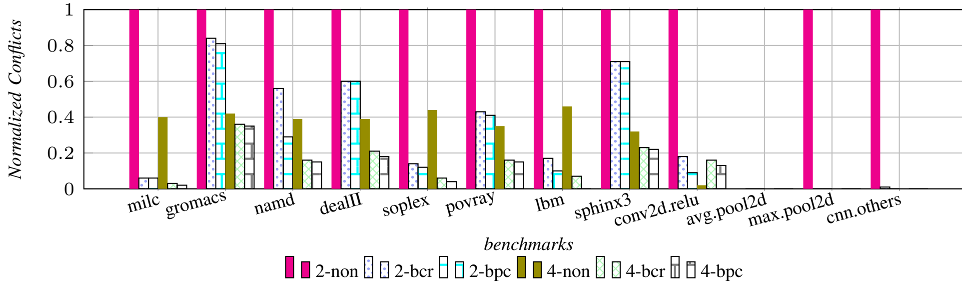
BANK *- bank count.* ***CONFS*** *- conflict count.* ***IMPV*** *- bpc conflict count improvement over the bcr method.*
** The conflicts and reductions are calculated based on the combined SPECfp and CNN-KERNEL benchmarks.*

TABLE II demonstrates the effectiveness of different RA methods. The *bcr* method achieves an overall bank conflict reduction of 94.16%, 76.51%, and 73.23% for the 2/4/8 bank settings (geometric mean). However, the *bpc* implementations yield even better results in all settings. Compared to the baseline of *bcr*, *bpc* achieves conflict reduction improvements of 51.21%, 38.64%, and 40.98% (geometric mean) respectively, indicating exceptionally high bank conflict reduction performance.

Table III: Platform-RV#1: Comparing the bank conflict reduction with spilling increment

BK-IMPL	*2-bcr*	*2-bpc*	*4-bcr*	*4-bpc*	*8-bcr*	*8-bpc*
SPEC.CR	19411	21738	7732	8442	3785	3985
SPEC.SI	8	45	2	82	21	144
CNN.CR	8366	8925	-1116	-16	-101	93
CNN.SI	0	1	2	82	2	175

BK-IMPL *- Bank setting and the register allocation implementations.* ***CR*** *- Bank conflict reduction count, which is the average of all reduction values.* ***SI*** *- Register spilling increment count.*



(a) Normalized bank conflicts with varying bank counts: The 'N' in the legends indicates the register file is N-banked.

benchmark	base-count
milc	652
gromacs	1681
namd	2241
dealII	1213
soplex	211
povray	1435
lbm	429
sphinx3	282
conv2d.relu	107.73
avg.pool2d	0
max.pool2d	1.20
cnn.others	16.18

(b) Maximum bank conflict count of SPECfp benchmarks

Figure 11: Dynamic bank conflicts with different bank counts in SPECfp: Comparing non,bcr,brc and bpc on Platform-RV#2

TABLE III provides a comparison of the reductions in bank conflicts with the increments in register spilling, addressing concerns about the potential side effects of bank assignment. From an execution cycle perspective, the increase in register spills often does not offset the benefits of conflict reduction in many architectures. In the 2-bank setting, *bpc* achieves significant conflict reduction with minimal spill increase compared to *bcr*. For the 8-bank setting, *bcr* controls spills better and may be more beneficial. Notably, negative conflict reductions are observed in CNN cases due to some convolution cases of CNN-KERNEL with high bank-conflict relevant instructions(approximately 5000) were the primary contributing factor. *PresCount*, accounting for bank pressure, enables more conflict-free assignment than *bcr* consequently.

2) Results of Platform-RV#2: In Fig.11, we present a comparison of different RA methods for Platform-RV#2, where the register budget is tight in each bank. As mentioned in Chapter IV-A2, we collected dynamic bank conflict instances. Therefore in Fig. 11 illustrates the dynamic instances of bank conflicts. Similar to Platform-RV#1, both *bcr* and *bpc* demonstrate significant reduction in bank conflicts, although not as much as when rich bank registers are available.This reduction is particularly noticeable in cases such as gromacs and dealII. Nonetheless, the reduction is still beneficial given the significant count of bank conflicts.

Table IV: Comparing Conflicts and Reductions of RA Methods in the Combined SPECfp and CNN-KERNEL Benchmarks with Different Bank Settings

BANK-METHOD	CONFS	Reduction		IMPV
		bcr	*bpc*	
2-STATIC	32432	23677	26898	3211
2-DYNAMIC	21457	14956	16653	1697
4-STATIC	9472	5470	5648	178
4-DYNAMIC	3461	-121	442	521

BANK-METHOD *- bank count and RA method.* ***CONFS*** *- conflict count.*
IMPV *-* ***bpc*** *conflict count improvement over the* ***bcr*** *method.*
** The conflicts and reductions are calculated based on the combined SPECfp and CNN-KERNEL benchmarks.*

Detailed reduction statistics are presented in TABLE IV. In the 2-bank setting, the number of dynamic instances is only half of the static instances, while for bank conflict reduction, it is nearly tripled. The geomean reduction shows an improvement of 59.75% and 18.99% over *bcr* for the 2-bank and 4-bank settings, respectively, in static execution. However, in dynamic execution, the improvement is only 22.36% and 15.19%. This gap can be attributed to the fact that dynamic execution only runs a portion of the code, leading to an exaggeration of encountered conflicts and a neglect of unexecuted instances. Nevertheless, considering that compilers cannot predict many runtime behaviors, static bank reduction remains valuable for evaluating conflict reduction effectiveness.

Table V: Comparing the bank conflict reduction with spilling increment

BK-IMPL	*2-bcr*	*2-bpc*	*4-bcr*	*4-bpc*
SPEC.CR	17722	19578	6947	7196
SPEC.SI	74	177	156	552
CNN.CR	5955	7320	-1477	-1548
CNN.SI	-7	-8	-6	13

BK-IMPL *- Bank setting and the register allocation implementations.*
CR *- Bank conflict reduction.* ***SI*** *- Register spilling increment.*

The bank conflict reduction count was compared against the spill increment count in TABLE V. Similar to the previous experiment, the spill increment count is generally much smaller than the instances of conflict reduction. However, for the 4-bank setting, the gap becomes closer, and both *bcr* and *bpc* result in negative reduction. This can be attributed to the fact that as the register budget decreases, there is a significant increase in register re-assignment, splitting, spilling, and other RA operations. This makes it more challenging for the heuristic method to maintain consistency with the register allocation behaviors, leading to inefficient bank assignment.

3) Result of Platform-DSA: Lastly, we evaluate the DSA-OP workload tailored for our 2-4 banked DSA architecture to investigate the potential co-design opportunities for register architecture. Since the *bcr* method cannot align subgroups on the *Platform-DSA*, we compare the performance of *bpc* with the default register allocation (*non*), assuming different register bank configurations. This comparison allows us to assess how

Table VI: Compare bank conflicts bewteen *bpc* and *non* with multiple bank settings

DSA-OP	BASE	Conflict Ratio (%)				
		2x4-bpc	*2-non*	*4-non*	*8-non*	*16-non*
reduce	5	0	100	60	40	20
red-ur	50	0	100	50	24	12
shruse	10	0	100	100	100	100
sr-ur	200	0	100	100	100	100
dw-conv2d	9	0	100	33.33	0	0
tr18987	175	0.57	100	44.57	22.86	10.86
tr15651	512	0	100	50	25	12.5
idft	16269	0	100	48.84	24.78	12.43
average	98.92	0.07	100	59.22	38.2	28.72

***BASE** - the baseline bank conflict count. **2x4-bcp** - the register file is 2-4 bank-subgrouped. The **bcp** method is applied in the register allocation. **2/4/8/16-non** - the register file is 2/4/8/16 banked. The default register allocation is applied.*

well *bpc* and associated simplified hardware is, compared with more sophisticated hardware design.

The experimental results are presented in TABLE VI. In comparison to the hardware solution, *PresCount* achieves comparable or even better bank reductions in the given benchmark on the 2-4 banked DSA. As observed from *2x4-bpc*, it eliminates nearly all bank conflicts (except *tr18987*) and reaches a geomean reduction of 99.85%. On the other hand, we observed from *2/4/8/16-non* that increasing the bank count physically alone generally results in fewer bank conflicts yet fails to eliminate them in most cases (except *dw-conv2d*). The leading performer, *16-non*, typically achieves a reduction of around 90% in bank conflicts. This can be attributed, in part, to *PresCount*'s effectiveness in doing conflict-free assignments with a large register count.

Table VII: Spills, copies and cycles of *bpr* and *2/4-non* on *Platform-DSA*.

DSA-OP	Spills		Copies		Cycles		
	bpc	*2/4-non*	*bpc*	*2/4-non*	*bpc*	*2-non*	*4-non*
reduce	0	0	0	0	89	93	91
red-ur	0	0	0	0	719	768	743
shruse	0	0	0	0	233	233	233
sr-ur	0	0	0	0	4603	4603	4603
dw-conv2d	0	0	0	0	126	124	124
tr18987	0	0	0	0	253	239	239
tr15651	0	0	14	96	2777	3991	3993
idft	2	0	2936	193	16843	16270	16269

***bcp** - The **bcp** method is applied in the register allocation for the 2-4 bank-subgrouped register file. **2/4-non** - The default register allocation is applied for a either 2-banked or 4-banked register file.*

TABLE VII presents the cost associated with bank conflict reduction, including register spills, register copies, and cycles. As observed in benchmark *reduce*, *red-ur*, and *tr15651*, reducing bank conflicts, which increase the latency of generating instruction output, is beneficial for the cycle count of compute-intensive kernels that yield single results from multiple inputs. However, for certain data-parallel workloads, despite notable reductions in bank conflicts, the performance impact remains minor due to the AI DSA's capability of parallelizing independent data computations. Benchmarks such as *shruse*, *su-ur* and *tr18987* exemplify this scenario. In cases like *tr15651* and *idft*, where substantial bank and subgroup pressure exists, the subgroup splitting heuristic is invoked to avoid register spills. *tr15651*, benefiting from fewer copies due to the heuristic and register coalescing optimization, showcases a positive impact on runtime performance. However, *idft* experiences a cycle increase due to the substantial copies generated, even though spills are reduced significantly. Given the restrictive register file usage, such trade-offs could be justified from a hardware-software co-design perspective.

Furthermore, the architecture-specific VLIW bundle constraint occasionally leads to cycle changes, as it forbids bundling instructions accessing the same bank, affecting instruction scheduling. This constraint negatively affects *dw-conv2d* and *tr18987*, rendering performance declines. However, it is challenging to address such inter-instruction restrictions with RCG. We plan to tackle it for future improvements.

V. Related Work

Large register files are segmented into SIMD vectors and utilized to mark fundamental paradigms in modern computing architectures. GPUs efficiently harness these vector registers through thread sharing, while CPUs leverage them via explicit programming or the auto-vectorization process, like loop vectorization [14], [15], superword-level-parallelism (SLP) vectorization [16]–[18], etc.

Notably, multi-banked register file [19] has been extensively studied in GPU architectures as GPU have a larger register file than CPU [20]. There are many GPU software/hardware co-design researches to address register reuse and underutilization. Gebhart et al. proposed a compiler-controlled register file hierarchy in their *Operand Register File (ORF)* study [5] as an alternative to *Register File Cache (RFC)* [21]. Esfeden et al. [22] combined compiler register allocation and register coalescing hardware to reduce register reads. J. Klosterman et al. accelerated access to short live-range operands by preloading instruction operands into a staging unit with compiler hints [23]. Xie et al. tried to address register file underutilization through exploration of register allocation and thread throttling on GPUs [24]. C. Yu et al. extended GPU registers to scratchpad memory to improve utilization of the register file [25].

Register allocation has been researched extensively for decades. Chaitin et al. implemented the first register allocator that viewed register allocation as a graph coloring problem [7]. Briggs et al. enhanced the register spilling with an optimistic coloring method [26]. Polleto et al. developed the linear scan approach to improve the time complexity [27].

Some GPU compilers, however, prefer register allocators based on SSA-form as it is capable of identifying the register required in polynomial time to facilitate dynamical thread-level register allocation [28]. Hack et al. were among the first researchers to demonstrate that the interference graph of an SSA program is chordal, thus coloring, spilling, and coalescing can be separated in SSA-based programs [29]. Pereira et

al. observed that the majority of Java Library methods offer chordal interference graphs [30].

All aforementioned register allocation approaches tried to address live range interference. Additional architectural register limitations are modeled as live range interference [7]. Scholz et al. formulated the global register allocation problem for irregular architectures as Partitioned Boolean Quadratic Problems (PBQP) under a heuristic approach that tried to this NP-complete problem in linear runtime complexity [31]. Hames et al. proposed a heuristic and a branch-and-bound solver to enhance the register allocation quality [32]. LLVM [33] adopts PBQP register allocator to address irregularity like register aliasing, register pairing, etc [34].

RCG-based register allocation is commonly employed to handle the irregularity of register bank conflicts. Although an RCG is a subgraph of RIG [8], it is built independently for bank assignment purposes. Pre-allocation and post-allocation methods are used to integrate RCG-based bank assignments into a register allocator. F. Zhou et al. [9] built the RCG ahead of the register assignment using a cost evaluation approach and a live-range splitting method for a 2-banked register file. proved that building an RCG bipartite with minimal effort is NP-complete [10]. The unbalanced bank assignments and register spills in the pre-allocation method can result in poor register allocation quality. Patney et al. selected post-allocation register renumbering despite the RCG being harder to color where massive register copies are generated to split an uncolored RCG and align subgroups, which requires many unassigned registers as a consequence [8]. The *Latency-Tolerant Register File (LTRF)* proposed by Sadrosadat et al. constructed an *Interval Conflict Graph (ICG)* and renumbered physical registers in distinct intervals to avoid bank conflicts [3], [35]. However, it has similar limitations as the post-allocation method.

Intel Graphics utilized a bank-conflict-reduction (BCR) heuristic to assign instruction operands to different banks only when feasible at register allocation time to avoid register spilling and to mitigate bank conflicts [12]. However, it does not model bank conflict restrictions more than a single instruction.

Regarding register pressure tracking, which is used to avoid register spilling while increasing ILP, many heuristic methods have been applied in pre-allocation instruction scheduling. Touati proposed a register saturation (RS) analysis with limited simultaneous live ranges in *Data Dependence Graph (DDG)* [36]. Similarly, Gavindarajan et al. used a coloring method on *Lineage Interference Graph (LIG)* to find *Heuristic Register Bound (HRB)* to achieve a similar goal [37].

VI. Conclusion

In this study, we addressed the bank conflict problem in multi-banked register files and proposed effective solutions. Our experiments confirmed the prevalence of bank conflicts in systems with multi-banked register files. To mitigate performance degradation and runtime overhead caused by bank conflicts, we introduced the *PresCount* method, which utilizes RCG-based bank assignment and bank pressure tracking. The results demonstrated superior performance compared to existing methods. Additionally, we presented a novel subgroup assignment technique that enabled our custom DSA to achieve performance on par with fully banked register file architectures, despite having a simplified micro-architecture.

We observed that tracking bank pressures proved to be a valuable heuristic for achieving balanced bank assignments, and early splitting of shared values played a critical role in achieving balanced subgroup assignments. Future research could focus on further enhancing these heuristics, as the current register allocation process may encounter challenges under high register bank pressure. Moreover, investigating the incorporation of *PresCount* with other RA methods could offer further insights.

VII. Ackownledgement

We extend our gratitude for the support received from various sources, including the National Key Research & Development Program of China (No. 2022YFB4500103), Programs for NSFC (No. 62032008), STCSM (No. 23511100100), Shanghai Pujiang Program (22PJ1422000), and Shanghai Rising-Star Program (22QB1404600). Jianguo Yao is the corresponding author.

Special thanks are extended to Chunling Hu for expertly guiding the finalization process. Additionally, we deeply appreciate the anonymous reviewers whose insightful feedback significantly contributed to enhancing this work.

Moreover, we acknowledge the contributions of Chuang Feng and Yuanzhi Hua for their insights into register file micro-architecture, Zhongjun Zhang for his involvement in the CNN-KERNEL benchmark work, and Dr. Heng Shi for his invaluable suggestions during the paper writing process.

References

[1] A. M. Radaideh and P. V. Gratz, "Exploiting zero data to reduce register file and execution unit dynamic power consumption in gpgpus," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020, doi:10.1109/DAC18072.2020.9218547.

[2] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 487–498, doi:10.1145/2485922.2485964.

[3] M. Sadrosadati, A. Mirhosseini, A. Hajiabadi, S. B. Ehsani, H. Falahati, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, "Highly concurrent latency-tolerant register files for gpus," *ACM Trans. Comput. Syst.*, vol. 37, no. 1–4, jan 2021, doi:10.1145/3419973.

[4] N. Jing, S. Chen, S. Jiang, L. Jiang, C. Li, and X. Liang, "Bank stealing for conflict mitigation in gpgpu register file," in *2015 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2015, pp. 55–60, doi:10.1109/ISLPED.2015.7273490.

[5] M. Gebhart, S. W. Keckler, and W. J. Dally, "A compile-time managed multi-level register file hierarchy," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 465–476, doi:10.1145/2155620.2155675.

[6] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li, "Exploration of gpgpu register file architecture using domain-wall-shift-write based racetrack memory," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6, doi:10.1145/2593069.2593137.

[7] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, “Register allocation via coloring,” *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[8] A. P. et al., “Conflict-free register allocation using a multi-bank register file with input operand alignment,” 2013, uS Patent 8,555,035.

[9] F. Zhou, J. Zhang, C. Wu, and Z. Zhang, “A register allocation framework for banked register files with access constraints,” vol. 3740, 10 2005, pp. 269–280, doi:10.1007/11572961_22.

[10] X. Zhuang and S. Pande, “Resolving register bank conflicts for a network processor,” in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’03. USA: IEEE Computer Society, 2003, p. 269, doi:10.1109/PACT.2003.1238022.

[11] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, “A hierarchical thread scheduler and register file for energy-efficient throughput processors,” *ACM Trans. Comput. Syst.*, vol. 30, no. 2, apr 2012, doi:10.1145/2166879.2166882.

[12] W.-Y. Chen, G.-Y. Lueh, P. Ashar, K. Chen, and B. Cheng, “Register allocation for intel processor graphics,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 352–364, doi:10.1145/3168806.

[13] J. Yao, H. Zhou, Y. Zhang, Y. Li, C. Feng, S. Chen, J. Chen, Y. Wang, and Q. Hu, “High performance and power efficient accelerator for cloud inference,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1003–1016, doi:10.1109/HPCA56546.2023.10070941.

[14] D. Nuzman, I. Rosen, and A. Zaks, “Auto-vectorization of interleaved data for simd,” in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 132–143, doi:10.1145/1133981.1133997.

[15] A. E. Eichenberger, P. Wu, and K. O’Brien, “Vectorization for simd architectures with alignment constraints,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI ’04. New York, NY, USA: Association for Computing Machinery, 2004, p. 82–93, doi:10.1145/996841.996853.

[16] S. Larsen and S. Amarasinghe, “Exploiting superword level parallelism with multimedia instruction sets,” *SIGPLAN Not.*, vol. 35, no. 5, p. 145–156, may 2000, doi:10.1145/358438.349320.

[17] H. Zhou and J. Xue, “A compiler approach for exploiting partial simd parallelism,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, mar 2016, doi:10.1145/2886101.

[18] H. Zhou and J.Xue, “Exploiting mixed simd parallelism by reducing data reorganization overhead,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO’16, 2016, p. 59–69, doi:10.1145/2854038.2854054.

[19] J.-L. Cruz, A. González, M. Valero, and N. Topham, “Multiple-banked register file architectures,” vol. 28, 02 2000, pp. 316 – 325, doi:10.1145/339647.339708.

[20] M. A. Ibrahim, O. Kayiran, Y. Eckert, G. H. Loh, and A. Jog, “Analyzing and leveraging decoupled l1 caches in gpus,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 467–478, doi:10.1109/HPCA51647.2021.00047.

[21] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, “Energy-efficient mechanisms for managing thread context in throughput processors,” in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 235–246, doi:10.1145/2000064.2000093.

[22] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, “Corf: Coalescing operand register file for gpus,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 701–714, doi:10.1145/3297858.3304026.

[23] J. Kloosterman, J. Beaumont, D. A. Jamshidi, J. Bailey, T. Mudge, and S. Mahlke, “Regless: Just-in-time operand staging for gpus,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 151–164, doi:10.1145/3123939.3123974.

[24] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan, “Enabling coordinated register allocation and thread-level parallelism optimization for gpus,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 395–406, doi:10.1145/2830772.2830813.

[25] C. Yu, Y. Bai, Q. Sun, and H. Yang, “Improving thread-level parallelism in gpus through expanding register file to scratchpad memory,” *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, nov 2018, doi:10.1145/3280849.

[26] P. Briggs, K. D. Cooper, and L. Torczon, “Improvements to graph coloring register allocation,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, p. 428–455, may 1994, doi:10.1145/177492.177575.

[27] M. Poletto and V. Sarkar, “Linear scan register allocation,” *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, p. 895–913, sep 1999, doi:10.1145/330249.330250.

[28] C. Abbott and D. Schürmann, “SSA-based register allocation for gpu architectures,” 2021, x.Org Developer Conference 2021. [Online]. Available: https://indico.freedesktop.org/event/1/contributions/7/

[29] S. Hack and G. Goos, “Optimal register allocation for ssa-form programs in polynomial time,” *Inf. Process. Lett.*, vol. 98, no. 4, p. 150–155, may 2006.

[30] F. M. Q. Pereira and J. Palsberg, “Register allocation via coloring of chordal graphs,” in *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, ser. Lecture Notes in Computer Science, K. Yi, Ed., vol. 3780. Springer, 2005, pp. 315–329, doi:10.1007/11575467_21.

[31] B. Scholz and E. Eckstein, “Register allocation for irregular architectures,” in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ser. LCTES/SCOPES ’02. New York, NY, USA: Association for Computing Machinery, 2002, p. 139–148, doi:10.1145/513829.513854.

[32] L. Hames and B. Scholz, “Nearly optimal register allocation with pbqp,” in *Proceedings of the 7th Joint Conference on Modular Programming Languages*, ser. JMLC’06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 346–361, doi:10.1007/11860990_21.

[33] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86, doi:10.1109/CGO.2004.1281665.

[34] F. M. Q. Pereira, “A survey on register allocation,” Tech. Rep., 2008.

[35] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, “Ltrf: Enabling high-capacity register files for gpus via hardware/software cooperative register prefetching,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 489–502, doi:10.1145/3173162.3173211. [Online]. Available: https://doi.org/10.1145/3173162.3173211

[36] S.-A.-A. Touati, “Register saturation in instruction level parallelism,” *International Journal of Parallel Programming*, vol. 33, no. 4, pp. 393–449, 2005, doi:10.1007/s10766-005-6466-x.

[37] R. Govindarajan, H. Yang, J. Amaral, C. Zhang, and G. Gao, “Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures,” *IEEE Transactions on Computers*, vol. 52, no. 1, pp. 4–20, 2003, doi:10.1109/TC.2003.1159750.