Optimizing VLIW Instruction Scheduling via a Two-Dimensional Constrained Dynamic Programming

CAN DENG, ZHAOYUN CHEN, YANG SHI, YIMIN MA, and MEI WEN, College of Computer, National University of Defense Technology, Changsha, China and Key Laboratory of Advanced Microprocessor Chips and Systems, Changsha, China
LEI LUO, College of Computer, National University of Defense Technology, Changsha, China and Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, China

Typical embedded processors, such as Digital Signal Processors (DSPs), usually adopt Very Long Instruction Word (VLIW) architecture to improve computing efficiency. The performance of VLIW processors heavily relies on Instruction-Level Parallelism (ILP). Therefore, it is crucial to develop an efficient instruction scheduling algorithm to explore more ILP. While heuristic algorithms are widely used in modern compilers due to simple implementation and low computational cost, they have limitations in providing accurate solutions and are prone to local optima. On the other hand, exact algorithms can usually find the optimal solution, but their high time overhead makes them less suitable for large-scale problems. This article proposes a two-dimensional constrained dynamic programming (TDCDP) approach and a quantitative model for instruction scheduling. The TDCDP approach achieves near-optimal solutions within an acceptable time overhead. Furthermore, we integrate our TDCDP approach into mainstream compiler architecture, encompassing Pre- and Post-RA (register allocation) scheduling. We conduct a quantitative evaluation of TDCDP compared with four heuristic algorithms on a typical VLIW processor. Our approach achieves an efficiency improvement of up to 58.34% in final solutions compared with the heuristic algorithms. Additionally, the Post-RA Scheduling enhances programs with an average speedup of 14.04% than solely applying the Pre-RA Scheduling.

CCS Concepts: • **Software and its engineering**; • **Computer systems organization** → **Architectures**;

Additional Key Words and Phrases: Embedded system, instruction scheduling, quantitative analysis, dynamic programming, VLIW

ACM Reference Format:
Can Deng, Zhaoyun Chen, Yang Shi, Yimin Ma, Mei Wen, and Lei Luo. 2024. Optimizing VLIW Instruction Scheduling via a Two-Dimensional Constrained Dynamic Programming. *ACM Trans. Des. Autom. Electron. Syst.* 29, 5, Article 83 (September 2024), 20 pages. https://doi.org/10.1145/3643135

The preliminary version of this work has been presented at the ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC) in 2022.
Authors' addresses: C. Deng, Z. Chen (Corresponding author), Y. Shi, Y. Ma, and M. Wen, College of Computer, National University of Defense Technology and Key Laboratory of Advanced Microprocessor Chips and Systems, Deya Road, Changsha, Hunan 410000, China; e-mails: dengcan@nudt.edu.cn, chenzhaoyun@nudt.edu.cn, shiyang14@nudt.edu.cn, yimmmin@163.com, meiwen@nudt.edu.cn; L. Luo, College of Computer, National University of Defense Technology and Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Deya Road, Changsha, Hunan 410073, China; e-mail: l.luo@nudt.edu.cn.

ACM 1084-4309/2024/09-ART83
https://doi.org/10.1145/3643135

1 INTRODUCTION

Embedded systems, with their real-time operating systems and specific functions, are widely used for single-task control and execution [1, 5, 40]. To achieve higher performance, **digital signal processors** (**DSPs**) are introduced to enhance their signal processing capabilities. The high specialization, real-time, and efficiency of DSPs bring significant improvements to embedded systems. On this basis, the introduction of the **very long instruction word** (**VLIW**) architecture further improves the system's processing capabilities. VLIW, a parallel processor architecture, achieves highly parallel computing by combining multiple simple instructions into complex instructions, which is particularly suitable for processing stream data, such as audio and video [7, 11, 35]. This makes embedded systems more efficient in handling complex signal tasks, thereby improving overall performance. Instruction scheduling plays a crucial role in the performance of VLIW processors. It involves reordering instructions during compilation to maximize **Instruction-Level Parallelism** (**ILP**) while considering dependencies and resource constraints [9, 28, 29]. As a result, designing an efficient instruction scheduling algorithm for VLIW compilers is of great significance.

Instruction scheduling is a known NP-hard problem [15, 16], and commercial compilers typically employ heuristic algorithms to tackle this challenge [4, 6]. These heuristics include **list scheduling** (**LS**) [39], **simulated annealing** (**SA**) [20], **genetic algorithm** (**GA**) [10], and **ant colony optimization** (**ACO**) [30]. Heuristic algorithms offer feasible solutions with low computational costs. In comparison to traditional algorithms, heuristics demonstrate better robustness and are suitable for addressing large-scale problems [23]. However, heuristics have limitations in that they can get trapped in local optima and cannot guarantee consistently high scheduling efficiency [2].

Exact algorithms, such as branch and bound (B&B) [31] and linear programming (LP) [19], are designed to find the optimal solution to a given problem. These algorithms explore the entire solution space through exhaustive search or divide-and-conquer techniques, gradually eliminating suboptimal solutions until the optimal solution is found. However, due to considering all feasible solutions, exact algorithms are computationally expensive. As a result, they are generally limited to solving small-scale problems [24, 25].

Instruction scheduling for VLIW compilers can be conceptualized as a two-dimensional assignment problem involving time slots and functional units [22, 34]. Heuristic algorithms typically address this problem in two phases. Firstly, time slots are assigned to instructions based on their dependency relationships. Then, functional units are allocated to the instructions, considering resource constraints. It is important to note that each functional unit can only be occupied by one instruction in a single time slot, and instructions must be executed in specific functional units. However, heuristics often divide instruction scheduling into two separate subproblems and solve them sequentially, which can lead to local optima.

In this article, we propose a **two-dimensional constrained dynamic programming** (**TDCDP**) approach for VLIW instruction scheduling. This approach combines both phases and solves them simultaneously, minimizing the likelihood of suboptimal solutions. By imposing strict constraints, TDCDP narrows down the solution space and provides stable near-optimal solutions within an acceptable time overhead.

In modern compilers (i.e., LLVM), instructions are usually scheduled in two phases, which can be concluded as Pre-RA (register allocation) Scheduling and Post-RA Scheduling. The Pre-RA Scheduling focuses on reordering instructions to maximize ILP, while the Post-RA Scheduling primarily addresses issues related to physical register contention and data dependencies, with the goal of further enhancing the quality of the generated code. In our work, we incorporate the scheduling architecture of modern compilers into a specific VLIW compiler. The compiler infrastructure depicted in Figure 1 accepts **machine instructions** (**MIs**) as input and produces target code as output. Within the compiler, the TDCDP approach is implemented in two distinct phases. In the

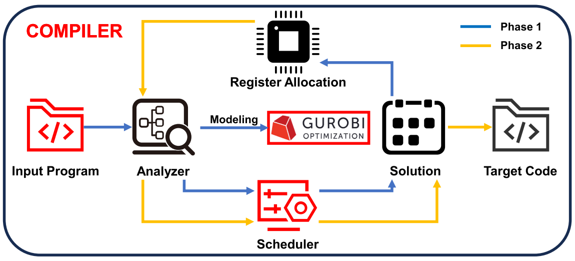


Fig. 1. The infrastructure of the compiler. The compiler optimization process unfolds in two distinct phases. Initially, the compiler constructs an intermediate solution by conducting analysis, performing instruction scheduling, and engaging in register allocation. Due to constraints in register availability, this solution might comprise load/store operations. Subsequently, the second phase commences with the updating of instruction details, encompassing those pertaining to load/store operations, through the analysis component. Following this, the scheduler refines the sequence of instructions and proceeds to produce the final target code. Within this scheduling module, the TDCDP approach is integrated. We underscore our contributions by marking the enhanced components in red.

initial phase, an analyzer component extracts instruction-level details and constructs dependency graphs. Utilizing these details, we formulate an optimization problem and derive the optimal solution utilizing the GUROBI [27] optimization solver. Subsequently, the scheduler component applies the TDCDP algorithm to reorder instructions, culminating in a schedule solution. During this procedure, TDCDP assigns time slots and functional units to instructions in a single step. The register allocation pass then assigns virtual registers from the schedule solution to physical counterparts, potentially introducing load/store operations to manage resource constraints. In the ensuing phase, the output from the register allocation is analyzed afresh to update the instruction data. The scheduler reorders the instructions, including any load/store operations, to refine and enhance the initial result. Finally, the optimized instruction sequence is emitted as target code. In Figure 1, the components influenced by our contributions are distinctly highlighted in red.

In more detail, this work makes significant contributions to the field in the following ways:

— Mathematical Model: We focus specifically on instruction scheduling scenarios in VLIW architecture and develop a mathematical model for the problem. Using a mathematical optimization solver, we determine the theoretical lower bound of program latency. This lower bound serves as a benchmark for evaluating the efficiency of our proposed approach.
— TDCDP Approach: Recognizing that instruction scheduling is a two-dimensional assignment problem involving time slots and functional units, we propose the TDCDP approach for VLIW instruction scheduling. Our objective is to generate near-optimal solutions while keeping the time overhead within acceptable levels.
— Integration with Modern Scheduling Architecture: Aligning the scheduling architecture of modern compilers, we implement our TDCDP approach in a VLIW compiler. This integration allows for practical applicability and compatibility with existing compiler frameworks.
— Quantitative Performance Evaluation: To assess the effectiveness of our approach, we have conducted comprehensive quantitative evaluations to measure its performance against four heuristic algorithms.

This article presents an extension of our previous work, which was initially presented at ASP-DAC 2022 [4]. We have made significant advancements in this article, which include:

— Improved Instruction Scheduling: In our prior article, the proposed approach scheduled instructions in two separate phases, resulting in solutions that were susceptible to local optima.

In this work, we have enhanced the instruction scheduling by introducing the TDCDP approach. TDCDP merges the phases and generates near-optimal solutions, effectively reducing the occurrence of local optima. Additionally, we have successfully minimized the time overhead by imposing stringent constraints.

— Integration into a VLIW Compiler: Moreover, we have integrated our TDCDP approach into a specific VLIW compiler. This compiler schedules instructions in two phases, namely Pre-RA Scheduling and Post-RA Scheduling. We have also conducted quantitative evaluations of solutions obtained from each phase.
— Expanded Benchmark Suite: In our previous work, we utilized transcendental functions as benchmarks. However, in this article, we have broadened our benchmark suite to incorporate various types of computations, such as matrix computations, filters, and vector operations. By incorporating these improvements and expanding our benchmark suite, we aim at providing a more comprehensive and enhanced understanding of our approach to instruction scheduling in VLIW architectures.

These advancements enhance the effectiveness of our instruction scheduling approach and provide a more comprehensive evaluation of its performance in different scenarios.

The rest of this article is organized as follows: Section 2 presents the fundamental concepts and characteristics of VLIW compilers; Section 3 elaborates on the problem modeling of instruction scheduling and the proposed TDCDP method. The implementation of our approach is outlined in Section 4. Experimental results and performance evaluation are presented in Section 5. Section 6 discusses related studies, and Section 7 summarizes the findings and outlines future research directions.

2 BACKGROUND

2.1 VLIW Instruction Scheduling

VLIW architecture holds great significance in the field of DSP and finds wide application in chip design for domains such as audio, video, image processing, and communication. By packaging multiple operands into a single long instruction, VLIW architecture enables parallel execution of these operands, thereby enhancing ILP and computational speed.

Exploitation of ILP is achieved through the instruction scheduling passes of a compiler [12, 18, 26]. As a result, the efficiency of instruction scheduling algorithms significantly impacts the performance of a VLIW processor. Instruction scheduling in VLIW is a static procedure that assigns time slots and functional units to instructions during compilation. As is shown in Figure 2, a typical instruction scheduling pass for VLIW involves placing instructions in a two-dimensional plane encompassing a time domain and a functional unit domain. Due to the characteristics of VLIW architecture, multiple instructions can be issued within a single time slot. Compared with traditional architectures, instruction scheduling for VLIW is more complex as it requires considering instruction dependency relationships and resource constraints.

2.1.1 *Dependency Relationship.*

The ILP of VLIW architecture is greatly influenced by the dependency relationships among instructions. Instruction dependency relationship refers to the constraint formed when certain instructions in program execution rely on the execution results of previous or subsequent instructions. Three types of instruction dependency relationships exist: data dependency, control dependency, and structural dependency.

Data dependency is the most common form of relationship among instructions. When one instruction requires the result calculated by another instruction, a data dependency relationship is established. In this scenario, the first instruction must wait for the calculation result of the dependent instruction before proceeding with its execution. Depending on the role of data (use or

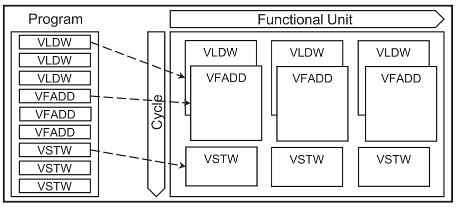


Fig. 2. Instruction scheduling of a VLIW program. The program consists of a series of instructions. The scheduler arranges these instructions on a two-dimensional plane, defined by functional units and time. It allocates functional units and time slots to each instruction, maintaining the appropriate dependencies between instructions to ensure the program's correctness.

define), data dependency can be categorized into three types: **write after write** (**WAW**), **write after read** (**WAR**), and **read after write** (**RAW**).

A control dependency relationship arises when control flow statements in the program prevent the execution of certain instructions. In this dependency relationship, the first instruction must wait for the evaluation result of the control flow statement before continuing with its execution.

A structural dependency relationship is formed when two instructions contend for the same processor functional unit. In this dependency relationship, the first instruction must wait for the functional unit to become available before entering the execution state.

Instruction dependency relationships have a significant impact on program execution efficiency, causing delays in program performance. To maximize program performance and efficiency, compilers can utilize instruction scheduling techniques to minimize the impact of dependency relationships.

2.1.2 Resource Constraints. Resource constraints in VLIW architecture encompass two key aspects: functional units and physical registers. In a VLIW processor, instructions are required to execute within specific functional units. However, the availability of functional units is limited, thereby establishing an upper bound on the achievable ILP.

During the Pre-RA Scheduling, the scheduler aims to optimize the utilization of functional units while ensuring that the width of the long instruction word remains within the designated limit. This involves efficiently mapping instructions to available functional units and maximizing ILP.

In the procedure of Post-RA Scheduling, virtual registers are substituted with physical registers after register allocation. Consequently, the compiler must ensure that instructions assigned to the same physical register in subsequent instruction scheduling stages do not conflict. These conflicts may include duplicate writes, reads and writes to the same physical register, or data dependencies between reads and writes. Thus, the compiler must consider such resource conflicts during both the pre and post-register allocation instruction scheduling phases to ensure accurate program execution.

By carefully managing these resource conflicts of instruction scheduling, both before and after register allocation, the compiler can guarantee the correct functioning and execution of programs within VLIW architectures.

2.2 Scheduling Architecture

A compiler is responsible for converting source code into target code that is platform-dependent. During this process, the compiler optimizes programs to enhance their execution efficiency. One crucial optimization in the compiler is instruction scheduling. As depicted in Figure 3, instruction scheduling of a modern compiler can be divided into two steps: Pre-RA Scheduling and Post-RA Scheduling. These steps differ in three key aspects.

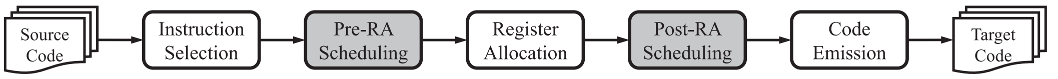


Fig. 3. Scheduling architecture of a modern compiler. Instruction scheduling is conducted in two phases: Pre- and Post-RA Scheduling.

First and foremost, they are executed at different times. Pre-RA Scheduling typically occurs before register allocation, while Post-RA Scheduling is carried out after register allocation.

Secondly, they target different instructions. Pre-RA Scheduling focuses on scheduling machine instructions, while certain instructions like Φ instructions and CALL instructions do not participate in Post-RA Scheduling.

Lastly, these steps operate under distinct resource constraints. Pre-RA Scheduling rearranges instructions before register allocation, utilizing virtual registers that have no limitations in number. However, in Post-RA Scheduling, virtual registers have already been mapped to physical registers. As a result, the compiler must consider the constraints imposed by the available physical registers due to their limited quantity.

In summary, Pre-RA Scheduling and Post-RA Scheduling perform distinct roles within the compiler, working together to enhance program performance and efficiency.

3 METHODOLOGY

In this section, we present a comprehensive model for addressing the instruction scheduling problems encountered in VLIW processors, and then introduce our TDCDP approach.

3.1 Problem Modeling

This section focuses on devising a mathematical model for the instruction scheduling problem in VLIW processors. First of all, the program is divided into multiple basic blocks, which are defined as a set of sequential instructions with exactly a single entrance and exit. Considering the strict execution order of the blocks, the scheduling problem for the entire program can be partitioned into several subproblems involving individual blocks. For a VLIW architecture that comprises m functional units ($\mathcal{F} = \{F_0 \dots F_f \dots F_{(m-1)}\}$). We define $X_{i,t}^{f}$ as the decision variable of instruction I_i, where I_i is the i_{th} instruction in the instruction set $\mathcal{I} = \{I_0 \dots I_i \dots I_{(n-1)}\}$, t denotes the t_{th} time slot in the given time sequence ($\mathcal{T} = \{0 \dots t \dots (T_0 - 1)\}$), and f denotes the f_{th} functional unit F_f in sequence $\mathcal{F}$. Additionally, T_0 denotes the total latency of the block when all instructions are executed in a sequential manner. It can be expressed mathematically as follows:

$$T_0 = \sum_{i=0}^{n-1} C_i \ (C_i \in C). \tag{1}$$

The binary decision variable $X_{i,t}^{f}$ signifies whether instruction I_i is executed starting from time slot t on functional unit F_f. If instruction I_i is scheduled to begin at time slot t and assigned to functional unit F_f, $X_{i,t}^{f} = 1$; otherwise, $X_{i,t}^{f} = 0$. $Y_{i,f}$ is defined as the coefficient of $X_{i,t}^{f}$. If the instruction I_i is assigned to functional unit F_f, $Y_{i,f} = 1$; otherwise, $Y_{i,f} = 0$. Considering a block $\mathcal{B}$ that contains n instructions, each instruction can be executed on specific functional units and has a fixed execution cycle ($C = \{C_0, C_1 \dots C_{(n-1)}\}$), which is dependent no the specific VLIW architecture.

The starting and completion time slots of instruction I_i are denoted as S_i and E_i, respectively. Here,

$$E_i = S_i + C_i, \tag{2}$$

where S_i can be computed with the following formulation:

$$S_i = \sum_{q=0}^{T_0-1} \left(1 - \sum_{t=0}^{q} X_{i,t}^{f} \right). \tag{3}$$

The latency T of the given block $\mathcal{B}$ can be defined as follows:

$$T = \max(E_i), \tag{4}$$

where $\max(E_i)$ is the last time slot occupied by the instructions in this block.

The optimization goal of our instruction scheduling problem is to minimize the latency of block $\mathcal{B}$, which can be presented as follows:

$$\min(T) = \min\left(\max\left(\sum_{q=0}^{T_0-1}\left(1 - \sum_{t=0}^{q} X_{i,t}^{f}\right) + C_i\right)\right). \tag{5}$$

The constraint conditions are presented as follows:

$$\sum_{f=0}^{m-1} \sum_{t=0}^{T_0-1} X_{i,t}^{f} = 1 \ (\forall I_i \in \mathcal{I}), \tag{6}$$

$$\sum_{f=0}^{m-1} \sum_{t=0}^{T_0-1} Y_{i,f} \cdot X_{i,t}^{f} = 1 \ (\forall I_i \in \mathcal{I}), \tag{7}$$

$$\sum_{i=0}^{n-1} X_{i,t}^{f} \leq 1 \ (\forall t \in \mathcal{T}, \forall F_f \in \mathcal{F}), \tag{8}$$

$$S_i + Edge_{i,l} \leq S_l \ (I_i, I_l \in \mathcal{I}). \tag{9}$$

Constraint 6 guarantees that every instruction within the given block is executed. Constraint 7 ensures that the instruction I_i is executed on the correct functional unit. Constraint 8 guarantees that at most one instruction is executed on each functional unit during any given time slot. Constraint 9 enforces the dependency relationship between parent and child instructions, where S_l is a child of S_i. $Edge_{i,l}$ represents the edge from S_i to S_l. For this edge, $Edge_{i,l}$ denotes the interval between the start time of the parent instruction I_i and that of the child I_l. The specific value is determined by the compiler.

Our proposed mathematical model is designed specifically for instruction scheduling in VLIW architecture. This model formulates the problem as an integer **linear programming problem** (**ILP**), with a time complexity of $O(2^n)$. To find an optimal scheduling solution, we utilize GUROBI [27], a linear programming solver. By leveraging GUROBI, we can effectively assign instructions within a block to suitable positions, minimizing latency while ensuring that the constraints related to functional units and dependencies are satisfied. The latency of the optimal solution obtained from the solver serves as the theoretical lower bound for quantifying and evaluating the performance of other schedulers.

3.2 Two-Dimensional Constrained Dynamic Programming

The GUROBI solver is instrumental in calculating the lower bounds for the basic blocks of input programs. However, due to the high time complexity of searching for optimal solutions, we introduce the TDCDP approach. This approach aims to provide near-optimal solutions while maintaining an acceptable time overhead by incorporating additional constraints. The TDCDP approach schedules instructions by simultaneously considering the assignment of time slots and functional units. The objective is to minimize the latency of basic blocks while adhering to the specified constraints. The method utilizes a bottom-up **dynamic programming** (**DP**) approach and is implemented using the following state transition equation:

$$dp[i, f(i)] = \frac{max}{0 \le k \le n} \left(\frac{min}{0 \le f(k) \le m} (dp[k, f(k)] + Edge_{k,i}) \right), \tag{10}$$

where the state $dp[i, f(i)]$ represents the position information in the plane, consisting of the time slot domain and the functional unit domain. The value of $dp[i, f(i)]$ denotes the issue time of instruction I_i, which occupies the functional unit $F_{f(i)}$. $f(i)$ represents the number of functional unit $F_{f(i)}$ in $\mathcal{F}$ and ranges from $[0, m-1]$. Here, m is the functional unit amount. I_k represents a successor instruction of I_i and n is the amount of successor instructions. $dp[k, f(k)]$ denotes the issue time of instruction I_k, which occupies functional unit $F_{f(k)}$. $Edge_{k,i}$ is the edge between instruction I_i and its successor I_k, representing the minimum issue interval between them. The term $dp[k, f(k)] + Edge_{k,i}$ denotes the nearest available time slot for I_i, considering all functional units. By selecting the minimum value of $dp[k, f(k)] + Edge_{k,i}$ among all functional units, we ensure that I_i is tightly assigned without any functional unit conflicts. Furthermore, by selecting the maximum value among all successors' nearest time slots, we ensure that I_i is assigned without dependency conflicts. The state transition equation simultaneously assigns time slots and functional units to instructions while considering dependency and resource constraints. These constraints narrow down the search space for instruction scheduling by excluding the conflicting solutions. The scheduling solution for the given basic block is obtained once all dp values are determined.

The boundary condition of TDCDP is

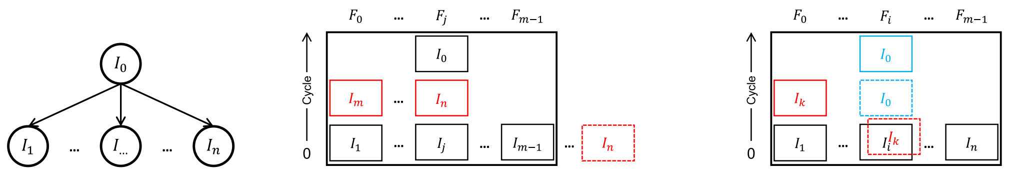
$$dp[i, f(i)] = 0 \ (\forall i, n = 0). \tag{11}$$

Boundary condition 11 ensures that instructions with no successors are assigned an issue time of 0. This condition establishes the initial states using a linear scan algorithm. When the number of instructions lacking successors surpasses the capacity of available functional units, the issue times for these instructions must be adjusted downward until a suitable time slot and functional unit become available. Consider a program with $n+1$ instructions; its **Directed Acyclic Graph** (**DAG**) is depicted in Figure 4(a). The nodes represent instructions, and edges symbolize dependencies. In this DAG, I_0 precedes n successors, which are initially allocated cycle 0 as their issue time and assigned functional units through linear scan, as shown in Figure 4(b). If the number of successors, n, is greater than the number of available functional units, m (i.e., $n > m$), instructions beyond the capacity are deferred to the subsequent time slot. The time complexity of this assignment method is O(n). However, if a DP approach is employed, the time complexity increases to O(nm). The boundary condition not only facilitates initialization but also establishes a basis for problem-solving by constraining the assignment of leaf instructions and reducing time complexity through a linear scan algorithm. This method enables efficient and effective instruction scheduling within the constraints of the given architecture.

In addition, the TDCDP approach should be conducted under the following constraint:

$$F_{i,t} \neq F_{k,t} \ (\forall t \in \mathcal{T}), \tag{12}$$

where $F_{i,t}$ denotes the functional unit occupied by instruction I_i at time slot t, and $F_{k,t}$ denotes the functional unit occupied by its successor instruction I_k at the same time slot t. Constraint 12 ensures that an instruction does not share its functional unit with any successors. As illustrated in Figure 4(a), if $n \le m$, the resulting schedule is depicted in Figure 4(c). Given that I_i and I_k are specifically executable on F_i, one of them must be deferred to a subsequent time slot in accordance with Constraint 12. This constraint is instrumental in preventing conflicts over functional units,



(a) DAG (b) Boundary condition (c) Constraint

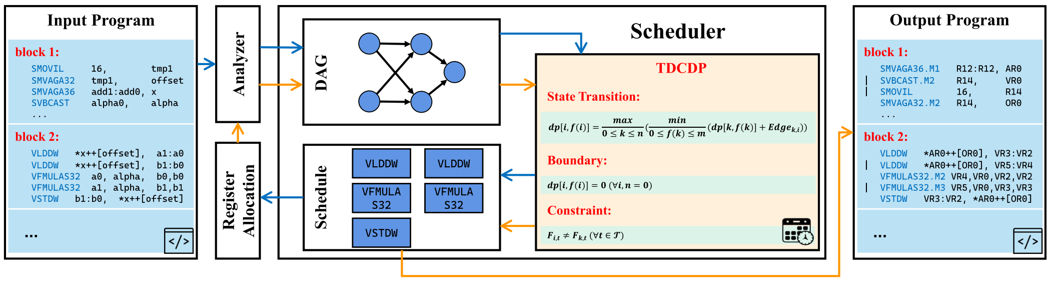
Fig. 4. The scenarios that the boundary condition and constraint impact. Within a given program, represented by a DAG as depicted in Figure 4(a), the instruction I_0 has n immediate successors. When scheduling these instructions, they are initially allocated starting from cycle 0. However, if $n > m$ (as illustrated in Figure 4(b)), the number of successors exceeds the available functional units, and the excess instructions are deferred to subsequent cycles. Conversely, Figure 4(b) illustrates the situation where $n \leq m$, allowing all successors of I_0 to be scheduled in cycle 0. Nonetheless, if both I_i and I_k are to be executed specifically on functional unit F_i, a conflict arises, necessitating the rescheduling of one of the instructions to the following cycle to avoid overlap on the same functional unit within a single cycle. This constraint ultimately delays the issue time of I_0 to a later cycle.

which are potent during the process of instruction scheduling. Enforcing this rule ensures that each instruction is allocated a unique functional unit, thereby diminishing resource contention and enhancing the efficiency of instruction execution.

The overview of our proposed TDCDP approach is shown in Algorithm 1. The main goal of the TDCDP approach is to generate near-optimal solutions for instruction scheduling while maintaining an acceptable time overhead. The TDCDP approach combines the assignment of time slots and functional units, allowing for scheduling of instructions in a single step. By incorporating constraints into the state transition equation, TDCDP effectively reduces the search space for instruction scheduling solutions. This reduction in search space enables TDCDP to obtain near-optimal solutions while minimizing the computational overhead. Overall, the TDCDP approach offers an efficient solution for instruction scheduling, striking a balance between optimality and computational efficiency.

3.3 Merging TDCDP into Compiler

Our proposed TDCDP approach utilizes DP to generate near-optimal solutions for instruction scheduling, while keeping the time overhead in check by incorporating constraints. This approach is particularly useful in exploring ILP for VLIW processors. However, it is important to note that increasing ILP also raises the risk of code spills. When the required number of registers exceeds the available allocation limit, the compiler must insert load/store instructions to spill code. This introduces new dependency relationships within the program, potentially impacting execution efficiency. To mitigate this issue, we adopt the architecture of modern compilers, which employs two phases for instruction scheduling: Pre-RA Scheduling and Post-RA Scheduling. Post-RA Scheduling can effectively reduce the impact of load/store instructions. Additionally, it enhances target code by increasing ILP, optimizing code layout, and eliminating instruction conflicts. The optimization workflow, as shown in Figure 5, involves several steps. Firstly, the input programs are divided into multiple basic blocks. The analyzer extracts information from instructions and establishes dependency relationships among them. These dependencies can be effectively represented using a DAG. Next, the scheduler employs TDCDP to assign time slots and functional units to instructions, generating schedule solutions for the basic blocks. Subsequently, the compiler utilizes the linear scan algorithm for physical register allocation, leveraging its efficiency, simplicity, and scalability commonly observed in industrial compilers. In this stage,


Input Program
Analyzer
Register Allocation
DAG
Schedule
VLDDW
VLDDW
VFMULA S32
VFMULA S32
VSTDW
Scheduler
TDCDP
State Transition:
$dp[i, f(i)] = \max_{0 \le k \le n}(\min_{0 \le f(k) \le m}(dp[k, f(k)] + Edge_{k,i}))$
Boundary:
$dp[i, f(i)] = 0\ (\forall i, n = 0)$
Constraint:
$F_{i,t} \neq F_{k,t}\ (\forall t \in \mathcal{T})$
Output Program

block 1:
```
SMOVIL      16,         tmp1
SMVAGA32    tmp1,       offset
SMVAGA36    add1:add0,  x
SVBCAST     alpha0,     alpha
...
```
block 2:
```
VLDDW     *x++[offset],  a1:a0
VLDDW     *x++[offset],  b1:b0
VFMULAS32  a0, alpha,  b0,b0
VFMULAS32  a1, alpha,  b1,b1
VSTDW     b1:b0,  *x++[offset]
```
...

block 1:
```
   SMVAGA36.M1   R12:R12, AR0
|  SVBCAST.M2    R14,     VR0
|  SMOVIL        16,      R14
   SMVAGA32.M2   R14,     OR0
   ...
```
block 2:
```
   VLDDW   *AR0++[OR0], VR3:VR2
|  VLDDW   *AR0++[OR0], VR5:VR4
   VFMULAS32.M2 VR4,VR0,VR2,VR2
|  VFMULAS32.M3 VR5,VR0,VR3,VR3
   VSTDW   VR3:VR2, *AR0++[OR0]
```
...

Fig. 5. The workflow of optimization in compiler. The input is composed of machine instructions, and the output is target code. The optimization is implemented in 5 steps: (1) The input program is divided into multiple basic blocks. For each block, the analyzer extracts information from instructions and build dependency relationships for them. The relationships are represented by a DAG. (2) The scheduler reorders instructions using the TDCDP approach and generate a solution. (3) The linear scan algorithm is applied to register allocation. (4) The analyzer updates the instructions of the blocks and rebuild the dependency relationships. and (5) Instructions are rescheduled through TDCDP, and the compiler generates the target code.

ALGORITHM 1: TDCDP Approach

```
Input: A basic block $\mathcal{B}$
Output: A schedule solution
  for each instruction $I_i$ in $\mathcal{I}$ do
      if $n = 0$ then                          ▷ $n$ is the successor instruction number of $I_i$
          for $t$ in $\mathcal{T}$ do
              for $f$ in $\mathcal{F}$ do
                  if $F_f$ is not occupied then
                      Place instruction $I_i$ in $(t, F_f)$
                  else
                      Shift to the next time slot
                  end if
              end for
          end for
      else
          Assign time slot and functional unit via state transition equation for instruction $I_i$
      end if
  end for
```

there may be a need to introduce load/store instructions due to limited resources. These load/store instructions introduce new dependency relationships. The analyzer then updates the instructions of the basic blocks and rebuilds the dependencies accordingly. Finally, the scheduler reorders the updated instructions for further optimization, ultimately yielding the target code.

By leveraging these techniques and workflows, our approach strives to achieve efficient and optimized instruction scheduling while effectively managing ILP and minimizing the impact of code spills.

4 IMPLEMENTATION

The implementation of our TDCDP approach involves modifying and extending the compiler. We enhance the compilation process by introducing a new scheduling algorithm TDCDP and modern scheduling architecture. Our TDCDP approach comprises two major components: intra-block instruction scheduling and dependency analysis. The TDCDP approach is integrated into the intra-block instruction scheduling phase, which takes place during both the Pre- and Post-RA Scheduling stages. In this phase, instructions are reordered while considering the constraints imposed by dependency relationships and available resources. The goal is to optimize the ordering of instructions to enhance performance and ensure efficient resource utilization. The dependency analysis phase is responsible for determining the execution order of instructions to avoid data conflicts and maximize ILP. This phase is performed both before and after register allocation. Prior to instruction scheduling, instructions are analyzed to determine their dependency relationships and ensure proper order of execution. After register allocation, the analyzer will build dependency relationships for the generated instructions, including load/store instructions that may be introduced. Both the intra-block instruction scheduling and dependency analysis phases are typically carried out statically during the compilation process. By incorporating these two significant components, we improve the overall compilation process, enabling effective instruction scheduling and dependency management. This, in turn, leads to optimized code generation and enhanced performance of the compiled software.

Furthermore, as is shown in Figure 6, our approach is implemented within a specific VLIW compiler. This involves replacing the original instruction scheduling algorithm with our TDCDP approach. Following the architecture of modern compilers, we introduce an additional dependency analysis and instruction scheduling phase into the compiler, which takes place after the register allocation stage. To incorporate these modifications, we have developed approximately 1500+ lines of C++ code. This code encompasses the necessary adjustments and extensions needed to integrate our TDCDP approach into the compiler. These modifications enable us to effectively utilize the power of our approach and enhance the overall performance of the compiler in terms of instruction scheduling and optimizing code generation.

5 EXPERIMENT

5.1 Experimental Setup

5.1.1 Platform. Our TDCDP approach is specifically evaluated on the FT-Matrix compiler [3].[1] FT-Matrix is an architecture based on a VLIW GPDSP framework. With 11 distinct functional units at its core, this architecture demonstrates a significant capacity for efficient computation across a variety of operational tasks. This not only illustrates the vast potential of the FT-Matrix but also reinforces its versatile nature in accommodating to differing computational needs.

5.1.2 Benchmark. In our evaluation, we utilize a set of 34 functions obtained from the **digital signal processing library** (**DSPLIB**) provided by the vendor. These benchmarks cover a range of categories, including transcendental functions, matrix computation, filters, and vector operations.

5.1.3 Baseline. The baselines, which are adopted for comparison against our TDCDP approach, consist of four heuristic algorithms, including **Longest Job First** (**LJF**) [13], **Critical Path Node Dominant** (**CPND**) [21], **Heterogeneous Earliest Finish Time** (**HEFT**) [33], and DPS [4].

— LJF: Instructions with longer execution cycles have a higher priority of being scheduled by the LJF method.
— CPND: The CPND recognizes three categories of nodes: Critical Path Nodes (CPNs), In-Branch Nodes (IBNs) and Out-Branch Nodes (OBNs). The scheduler allocates the instructions in the order of CPNs > IBNs > OBNs.

[1]Source code is available at https://github.com/Nudt-Dc/FT-Matrix.git

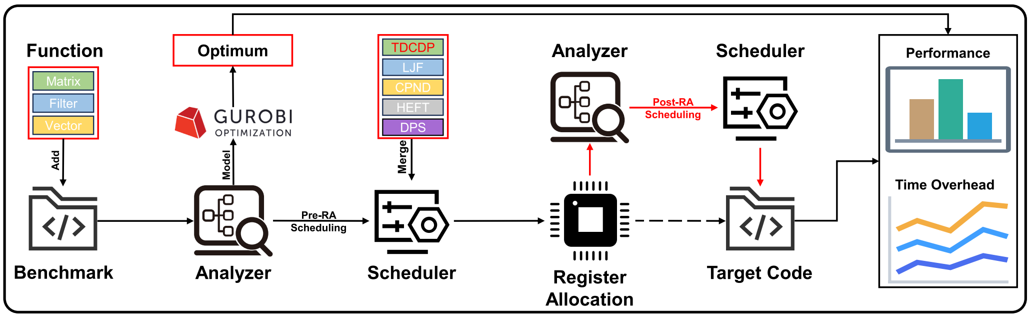


Fig. 6. The overview of the implementation. We have introduced major enhancements over our prior work, which are denoted in red. Initially, we have expanded the benchmark suite to encompass additional functions. Subsequently, we formulated the optimization problem and determined the optimal solution using the GUROBI solver. In the scheduling component, we introduced an innovative instruction scheduling algorithm, alongside four heuristic methods serving as baselines. Contrary to our earlier work, where the compiler generated target code directly post-register allocation, the current approach re-engages in scheduling after register allocation through Post-RA Scheduling. Ultimately, we conduct a comprehensive analysis of the optimal solutions alongside the target code to evaluate the execution performance and assess the time overhead.

— HEFT: The priorities are determined by the finish time of instructions in the HEFT method. The earlier the instruction is finished, the higher the priority.
— DPS: In this method, instructions are heuristically scheduled with priorities which are calculated in a DP manner. The DPS is proposed in our previous work.

The aforementioned heuristic algorithms follow a two-step process to generate instruction scheduling solutions. This process involves calculating priorities for instructions and subsequently allocating functional units to schedule those instructions. In contrast, our TDCDP approach combines these two steps into a single integrated process. By doing so, we streamline the instruction scheduling procedure, eliminating the need for separate priority calculation and functional unit allocation stages. This integrated approach allows for better coordination and optimization of instruction scheduling, ultimately improving the overall efficiency and performance of the compiler.

5.1.4 Metrics. The scheduling objectives of our approach encompass two main aspects: generating near-optimal solutions within an acceptable time overhead and optimizing programs with Post-RA Scheduling. Accordingly, the metrics used in this section focus on execution performance and time overhead. To assess the efficiency of the different methods, we also leverage the GUROBI solver, which provides optimal solutions for given scheduling problems. The total latency of an optimal solution serves as the theoretical lower bound for the problem at hand. To calculate the efficiencies of the various methods, we employ the following formulation:

$$E_{i,j} = \frac{G_i}{C_{i,j}} \times 100\%, \tag{13}$$

where $E_{i,j}$ is the efficiency of algorithm j in benchmark i. $C_{i,j}$ represents the latency of benchmark i obtained by algorithm j, while G_i represents the optimal solution of benchmark i generated by the GUROBI solver.

Through the formulation, we can determine the efficiency of that algorithm for the given benchmark. This efficiency metric provides a quantitative measure of how close each algorithm comes to the theoretically optimal solution, allowing for a thorough evaluation and comparison of their performance.

5.2 Evaluation

In our experimental evaluation, we aim at answering three crucial questions:

(1) Improvement over heuristic algorithms: We assess the extent of improvement achieved by TDCDP compared with other adopted heuristic algorithms. This evaluation allows us to determine the effectiveness and superiority of our proposed approach.
(2) Profitability of Post-RA Scheduling: We measure the benefits gained from utilizing Post-RA Scheduling during compilation. This evaluation enables us to quantify the advantages and optimizations that this scheduling technique brings to the compilation process.
(3) Efficiency of the adopted algorithms: We evaluate the efficiency of the algorithms utilized in our research. This evaluation encompasses both execution performance and time overhead, providing insights into the computational efficiency and resource utilization of the adopted algorithms.

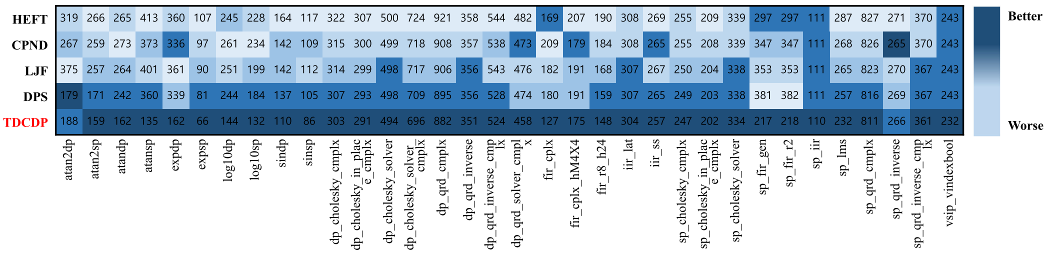
By considering these measures, we gain a comprehensive understanding of the performance, profitability, and efficiency of our proposed approach, enabling us to make informed conclusions about its effectiveness in instruction scheduling.

5.2.1 Execution Performance. To showcase the performance of the algorithms, we analyze the scheduling solutions obtained for the benchmarks. Specifically, we evaluate the execution performance, focusing on two key metrics: latency and efficiency. The latency metric refers to the actual number of execution cycles required to complete a solution. A lower latency value indicates a more efficient solution generated by the algorithm. On the other hand, efficiency quantifies the deviation between the optimal solution and the obtained solution. As previously mentioned in Section 5.1.4, the calculation of efficiency is outlined in Equation (13).

Latency. The latency of the final generated code serves as a crucial criterion for evaluating instruction scheduling algorithms. As depicted in Figure 7, the latency of the final solutions produced by the algorithms adopted in this article is showcased. The deeper color represents a better performance in latency. Notably, our TDCDP approach exhibits a significant superiority over the other heuristics in 32(94.12%) out of 34 benchmarks. On average, the final solutions achieved through TDCDP demonstrate a latency of 281.88 cycles, whereas DPS, LJF, CPND, and HEFT yield latencies of 318.24, 330.97, 328.91, and 332.09 cycles, respectively. This indicates that TDCDP outperforms these heuristics by reducing the latency by up to 225, 266, 238, and 278 cycles, respectively. On average, TDCDP achieves an improvement of 36.35 cycles compared with the competing algorithms. These results clearly establish the superior performance of TDCDP in reducing latency and outperforming the other heuristics across a significant majority of benchmarks. Its ability to achieve substantial latency savings further underscores its effectiveness and potential for enhancing execution performance in instruction scheduling.

Efficiency. As mentioned earlier, efficiency plays a crucial role in evaluating the performance of algorithms in instruction scheduling problems. It directly reflects the disparity between the obtained solution and the optimal solution derived from the GUROBI solver. In this section, we quantitatively assess the algorithms based on their efficiency, as illustrated in Figure 8. It's worth noting that the solver times out in *expdp* and *log10sp* benchmarks. Among the algorithms examined, TDCDP exhibits an average efficiency of 77.45% and generally demonstrates higher efficiency compared with the heuristics. Furthermore, TDCDP achieves average efficiency improvements of 9.48%, 12.8%, 12.84%, and 13.71%, respectively, when compared with the other algorithms.

Figure 8 also presents the number of load/store instructions generated during the compilation process. TDCDP exhibits a significantly lower number of load/store instructions compared with the other heuristics. It is evident that the greater the difference in the number of load/store



	atan2dp	atan2sp	atandp	atansp	expdp	expsp	log10dp	log10sp	sindp	sinsp	dp_cholesky_cmplx	dp_cholesky_in_place_cmplx	dp_cholesky_solver	dp_cholesky_solver_cmplx	dp_qrd_cmplx	dp_qrd_inverse	dp_qrd_inverse_cmplx	dp_qrd_solver_cmplx	fir_cplx	fir_cplx_hM4X4	fir_r8_h24	iir_lat	iir_ss	sp_cholesky_cmplx	sp_cholesky_in_place_cmplx	sp_cholesky_solver	sp_fir_gen	sp_fir_r2	sp_iir	sp_lms	sp_qrd_cmplx	sp_qrd_inverse	sp_qrd_inverse_cmplx	vsip_vindexbool
HEFT	319	266	265	413	360	107	245	228	164	117	322	307	500	724	921	358	544	482	169	207	190	308	269	255	209	339	297	297	111	287	827	271	370	243
CPND	267	259	273	373	336	97	261	234	142	109	315	300	499	718	908	357	538	473	209	179	184	308	265	255	208	339	347	347	111	268	826	265	370	243
LJF	375	257	264	401	361	90	251	199	142	112	314	299	498	717	906	356	543	476	182	191	168	307	267	250	204	338	353	353	111	265	823	270	367	243
DPS	179	171	242	360	339	81	244	184	137	105	307	293	498	709	895	356	528	474	180	191	159	307	265	249	203	338	381	382	111	257	816	269	367	243
TDCDP	188	159	162	135	162	66	144	132	110	86	303	291	494	696	882	351	524	458	127	175	148	304	257	247	202	334	217	218	110	232	811	266	361	232

Fig. 7. The latency of final solutions obtained by the adopted algorithms. A lower number of cycles in the latency indicates better performance. TDCDP exhibits superior performance compared with the adopted heuristic algorithms in 32(94.12%) benchmarks.

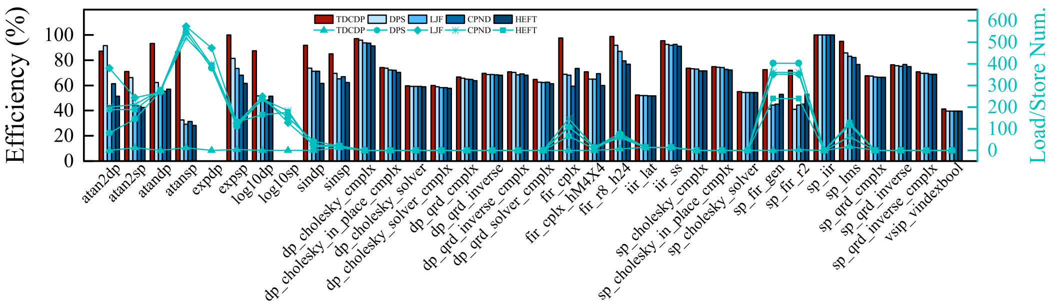


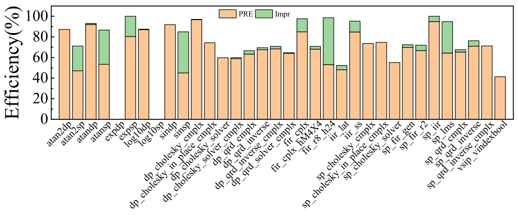
Fig. 8. The results of efficiency and load/store numbers. The bars represent the efficiency of the final solutions obtained by the adopted algorithms. The TDCDP achieves an average efficiency of 77.45%. Compared with the other heuristics, TDCDP demonstrates a significant efficiency improvement of up to 58.34% in *atansp*. The point plots illustrate the number of load/store instructions generated during compilation. The TDCDP achieves a greater efficiency improvement in the cases where there is a larger disparity in the number of load/store instructions between TDCDP and the other heuristic algorithms. The GUROBI solver times out in *expdp* and *log10sp* benchmarks.

Table 1. Correlations between Efficiency Improvement and Load/Store Instruction Number

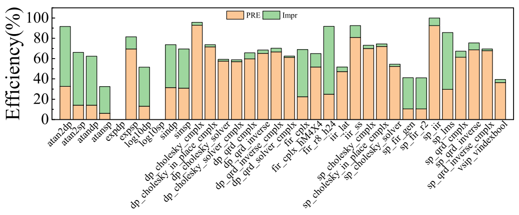
	TDCDP-DPS	TDCDP-LJF	TDCDP-CPND	TDCDP-HEFT
$\mathcal{R}$	0.768	0.783	0.817	0.837
p	<0.001	<0.001	<0.001	<0.001

The discrepancy in the load/store instructions between the TDCDP and other heuristic algorithms shows a positive correlation with the efficiency improvement attained by TDCDP. In this context, TDCDP-DPS represents the test between TDCDP and DPS.

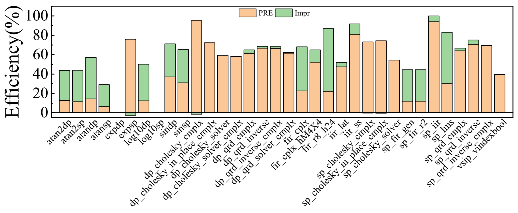
instructions between TDCDP and the other heuristic algorithms, the greater the efficiency improvement achieved by TDCDP. To assess the correlation between these two factors, we employ SPSS22 [14] software to perform a Spearman correlation test. The resulting correlation coefficients ($\mathcal{R}$) are presented in Table 1. The coefficients for TDCDP-DPS, TDCDP-LJF, TDCDP-CPND, and TDCDP-HEFT are found to be 0.768, 0.783, 0.817, and 0.837, respectively. The statistical analysis demonstrates a positive correlation between the difference in the number of load/store instructions and the efficiency improvement achieved by TDCDP. The p-values indicate the statistical significance of these results.



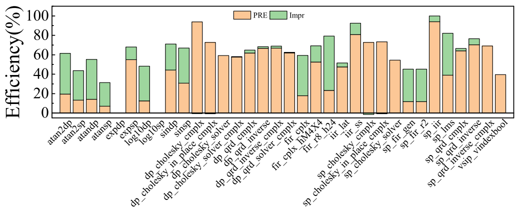
(a) TDCDP



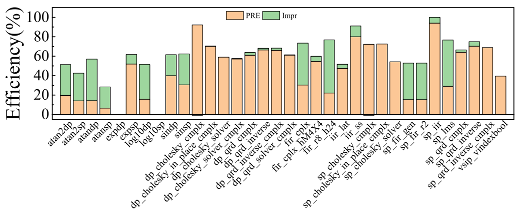
(b) DPS



(c) LJF



(d) CPND



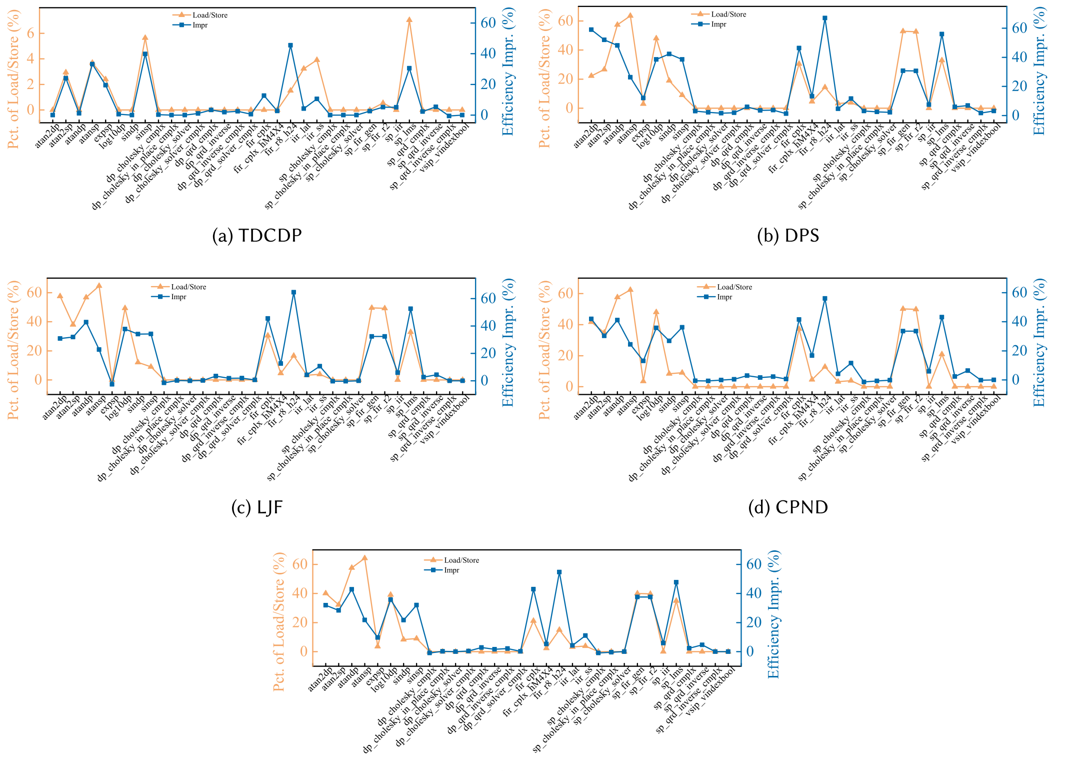
(e) HEFT

Fig. 9. The efficiency of the adopted algorithms in different phases. The yellow bars denote the efficiency of solutions obtained prior to Post-RA Scheduling, while the green bars represent the efficiency improvement achieved after applying Post-RA Scheduling. Unfortunately, the efficiency data for *expdp* and *log10sp* is unavailable due to time-outs of the GUROBI solver during these two benchmarks.

Post-RA Scheduling. We have integrated Post-RA Scheduling into the VLIW compiler architecture to enhance its performance. As shown in Figure 9, the application of Post-RA Scheduling has resulted in significant improvements. Our experiments with various algorithms have demonstrated an average efficiency boost of 7.95%, 19.77%, 15.83%, 15.9%, and 15.12% in TDCDP, DPS, LJF, CPND, and HEFT, respectively. Specifically, TDCDP has achieved a remarkable improvement of up to 33.24% in the *atansp* benchmark, while DPS has shown an impressive enhancement of up to 66.95% in *fir_r8_h24*. LJF has achieved up to 64.68% improvement in *fir_r8_h24*, CPND has achieved up to 56.06% improvement in *fir_r8_h24*, and HEFT has demonstrated an improvement of up to 54.69% in *fir_r8_h24*.

The introduction of load/store instructions can negatively impact program execution speed. However, Post-RA Scheduling offers a solution by strategically rescheduling instructions. As is shown in Figure 10(a)–10(e), the efficiency improvement achieved by Post-RA Scheduling and the percentage of load/store instructions in programs generated by register allocation exhibit a similar trend. The relationship between these factors is summarized in Table 2. The $\mathcal{R}$ for TDCDP, DPS, LJF, CPND, and HEFT have a value of 0.735, 0.833, 0.828, 0.858, and 0.867, respectively. Their correlation coefficients indicate a significantly positive correlation between the two factors.

Overall, the incorporation of Post-RA Scheduling into the compiler has proved to be beneficial in optimizing code generation processes, leading to notable performance enhancements.



(a) TDCDP (b) DPS

(c) LJF (d) CPND

(e) HEFT

Fig. 10. The percentage of load/store instructions in the programs generated by register allocation and the efficiency improvement obtained by Post-RA Scheduling. These two factors show a similar trend on the whole.

Table 2. The Correlation Involves Two Factors

	TDCDP	DPS	LJF	CPND	HEFT
$\mathcal{R}$	0.735	0.833	0.828	0.858	0.867
p	<0.001	<0.001	<0.001	<0.001	<0.001

One is the efficiency improvement obtained by Post-RA Scheduling. The other is the load/store instruction percentage in the generated code after register allocation. The $\mathcal{R}$ values indicates that the two factors are positively correlated.

5.2.2 Time Overhead. Both time overhead and efficiency are crucial considerations when evaluating compilers. In our study, we also assessed the time overhead associated with the adopted algorithms and the GUROBI solver. As depicted in Figure 11, both the TDCDP approach and the heuristics incur a time overhead ranging from $[0.14s, 4.82s]$. In practical applications, this level of time overhead is generally deemed acceptable. Furthermore, the TDCDP approach exhibits lower time overhead compared to the heuristics. On the other hand, the GUROBI solver, which utilizes integer linear programming to produce optimal solutions, incurs a time overhead ranging from $[1s, 4.5\times10^4 s]$. This discrepancy arises from the fact that the solver typically exhaustively searches all possible solutions. When dealing with smaller-scale problems, the time overhead of the GUROBI solver is comparable to that of the aforementioned algorithms. However, as the problem scale increases, the time overhead of the GUROBI solver grows geometrically. In our work, the time

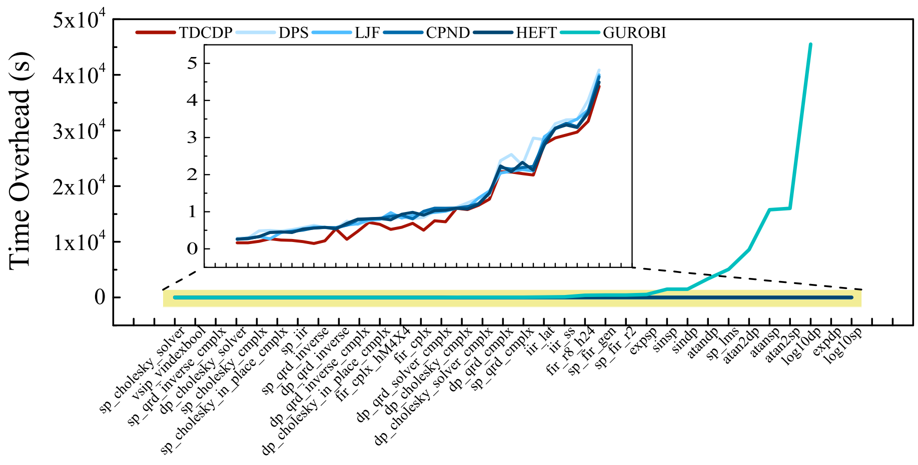


Fig. 11. Time overhead of the algorithms and GUROBI solver. The time overhead of the adopted algorithms ranges from $[0.14s, 4.82s]$. Compared with the heuristic algorithms, TDCDP takes lower time overhead. The time overhead of the GUROBI solver ranges from $[1s, 4.5 \times 10^4 s]$. The GUROBI solver produces optimal solutions and the time overhead is up to 1×10^4 orders of magnitude higher than the other approaches.

overhead of the GUROBI solver can be up to several orders of magnitude higher (i.e., 1×10^4 times higher) than that of the heuristics and TDCDP. Consequently, exact methods like the GUROBI solver are generally not considered practical for instruction scheduling in industry-standard compilers.

6 RELATED WORK

Instruction scheduling is a challenging problem that falls into the category of NP-hard problems [15, 16, 37]. Searching for optimal solutions for instruction scheduling problems often incurs high costs in terms of time or space. As a result, modern compilers typically rely on heuristic algorithms for instruction scheduling. Heuristics offer a tradeoff between time and solution quality, allowing for efficient handling of large-scale problems [8, 38]. Given the complexity of instruction scheduling, heuristics employ greedy strategies to tackle the multitude of instructions and constraints involved, making them robust and adaptable. Over the past few decades, heuristic algorithms such as LS, GA, and ACO have been proposed to address instruction scheduling problems [10, 20, 30, 39].

In a study by Giesemann et al. [9], a dynamic heuristic for instruction scheduling problems based on evolutionary algorithms was introduced. Their approach was evaluated on a VLIW processor and demonstrated superior performance in reducing code size compared with static list scheduling. Additionally, their approach achieved a 2% improvement in instruction scheduling speed. Dynamic heuristic algorithms exhibit better adaptability compared with static heuristics, although the efficiency improvement of their approach was not significantly pronounced.

Exact algorithms can provide optimal solutions and are often employed to establish upper limits for the performance of other algorithms. Examples of exact algorithms include B&B, **integer programming** (**IP**), and DP [17, 32, 36]. However, due to their high computational complexity, exact algorithms are typically limited to small-scale problems. In the context of industry compilers, exact algorithms are not preferred as the primary choice for instruction scheduling problems.

Shobaki et al. [30] presented a B&B algorithm for solving instruction scheduling problems, utilizing it to search for an optimal solution to a subproblem. However, due to the limitations of exact algorithms, their instruction scheduling was ultimately implemented in a heuristic manner. The solutions obtained from B&B served as a reference for the heuristics employed.

In contrast to the aforementioned methods, our proposed TDCDP approach combines the assignment of time slots and functional units via DP. By incorporating strict constraints, TDCDP is capable of generating near-optimal solutions for instruction scheduling while effectively reducing the time overhead to an acceptable level.

7 CONCLUSION

This article presents TDCDP, a novel instruction scheduling approach for VLIW compilers. TDCDP addresses the simultaneous assignment of time slots and functional units for instructions while incorporating strict constraints to refine the search space. The scheduling architecture of modern compilers is integrated into the VLIW compiler to facilitate the implementation of TDCDP. The experimental results showcase the high efficiency of TDCDP in solving instruction scheduling problems, surpassing the performance of heuristic algorithms. Furthermore, the integration of Post-RA Scheduling greatly enhances the quality of the generated code. The quantitative evaluation highlights the adaptability and robustness of TDCDP. However, there is still potential for further improvement in scheduling efficiency. In future research, exploring the adjustment of constraints to optimize the generated code would be an interesting avenue.

REFERENCES

[1] Paul Bogdan, Fan Chen, Aryan Deshwal, Janardhan Rao Doppa, Biresh Kumar Joardar, Hai (Helen) Li, Shahin Nazarian, Linghao Song, and Yao Xiao. 2019. Taming extreme heterogeneity via machine learning based design of autonomous manycore systems. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion* (New York, New York) *(CODES/ISSS '19)*. Association for Computing Machinery, New York, NY, Article 21, 10 pages. DOI: https://doi.org/10.1145/3349567.3357376

[2] Sanjoy Chakraborty, Apu Kumar Saha, Sushmita Sharma, Seyedali Mirjalili, and Ratul Chakraborty. 2021. A novel enhanced whale optimization algorithm for global optimization. *Comput. Ind. Eng.* 153 (2021), 107086. DOI: https://doi.org/10.1016/J.CIE.2020.107086

[3] Shuming Chen, Yaohua Wang, Sheng Liu, Jianghua Wan, Haiyan Chen, Hengzhu Liu, Kai Zhang, Xiangyuan Liu, and Xi Ning. 2014. FT-Matrix: A coordination-aware architecture for signal processing. *IEEE Micro* 34, 6 (2014), 64–73. DOI: https://doi.org/10.1109/MM.2013.129

[4] Can Deng, Zhaoyun Chen, Yang Shi, Xichang Kong, and Mei Wen. 2022. Exploring ILP for VLIW architecture by quantified modeling and dynamic programming-based instruction scheduling. In *27th Asia and South Pacific Design Automation Conference (ASP-DAC'22)*. IEEE, 256–261. DOI: https://doi.org/10.1109/ASP-DAC52403.2022.9712500

[5] Aryan Deshwal, Syrine Belakaria, Ganapati Bhat, Janardhan Rao Doppa, and Partha Pratim Pande. 2021. Learning pareto-frontier resource management policies for heterogeneous SoCs: An information-theoretic approach. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 607–612. DOI: https://doi.org/10.1109/DAC18074.2021.9586283

[6] Andreas Diavastos and Trevor Carlson. 2022. Efficient instruction scheduling using real-time load delay tracking. *ACM Transactions on Computer Systems* 40 (2022), 1:1–1:21. DOI: https://doi.org/10.1145/3548681

[7] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. M. O. Homewood. 2000. Lx: A technology platform for customizable VLIW embedded processing. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 203–213. DOI: https://doi.org/10.1109/ISCA.2000.854391

[8] E. Figielska. 2014. Algorithms using list scheduling and greedy strategies for scheduling in the flowshop with resource constraints. *Zeszyty Naukowe Warszawskiej Wyższej Szkoły Informatyki* nr 11 (2014), 29–39. DOI: https://doi.org/10.26348/znwwsi.11.29

[9] Florian Giesemann, Lukas Gerlach, and Guillermo Payá-Vayá. 2020. Evolutionary algorithms for instruction scheduling, operation merging, and register allocation in VLIW compilers. *Journal of Signal Processing Systems* 92, 7 (2020), 655–678. DOI: https://doi.org/10.1007/s11265-019-01493-2

[10] Florian Giesemann, Guillermo Payá Vayá, Lukas Gerlach, Holger Blume, Fabian Pflug, and Gabriele von Voigt. 2017. Using a genetic algorithm approach to reduce register file pressure during instruction scheduling. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'17)*. Yale N. Patt and S. K. Nandy (Eds.). IEEE, 179–187. DOI: https://doi.org/10.1109/SAMOS.2017.8344626

[11] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Wen-mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1-2 (1993), 229–248. DOI: https://doi.org/10.1007/BF01205185

[13] Howard Iams. 1985. Characteristics of the longest job for new retired workers: Findings from the New Beneficiary Survey. *Social Security Bulletin* 48 (04 1985), 5–21.

[14] IBM. 2021. IBM SPSS Statistics 22. Retrieved from https://www.ibm.com/spss

[15] J. Jonsson and J. Vasell. 1997. On objective function selection in list scheduling algorithms for digital signal processing applications. *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on* 1, 667–670 vol.1. DOI: https://doi.org/10.1109/ICASSP.1997.599856

[16] C. W. Kessler. 2018. *Compiling for VLIW DSPs*. 979–1020. DOI: https://doi.org/10.1007/978-3-319-91734-4_27

[17] Christoph Keßler and Andrzej Bednarski. 2001. A dynamic programming approach to optimal integrated code generation. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems* (Snow Bird, Utah, USA) *(LCTES '01)*. Association for Computing Machinery, New York, NY, 165–174. DOI: https://doi.org/10.1145/384197.384219

[18] Christoph W. Keßler and Andrzej Bednarski. 2006. Optimal integrated code generation for VLIW architectures. *Concurr. Comput. Pract. Exp.* 18, 11 (2006), 1353–1390. DOI: https://doi.org/10.1002/CPE.1012

[19] David Kiessling, Andrea Zanelli, Armin Nurkanović, Joris Gillis, Moritz Diehl, Melanie Nicole Zeilinger, Goele Pipeleers, and Jan Swevers. 2022. A feasible sequential linear programming algorithm with application to time-optimal path planning problems. *2022 IEEE 61st Conference on Decision and Control (CDC)* (2022), 1196–1203. Retrieved from https://api.semanticscholar.org/CorpusID:248496018

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680. DOI: https://doi.org/10.1126/science.220.4598.671

[21] Yu-Kwong Kwok and I. Ahmad. 1995. Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*. 36–43. DOI: https://doi.org/10.1109/SPDP.1995.530662

[22] Chingren Lee, Jenq Kuen Lee, TingTing Hwang, and Shi-Chun Tsai. 2003. Compiler optimization on VLIW instruction scheduling for low power. *ACM Trans. Design Autom. Electr. Syst.* 8, 2 (2003), 252–268. DOI: https://doi.org/10.1145/762488.762494

[23] Roberto Castañeda Lozano and Christian Schulte. 2019. Survey on combinatorial register allocation and instruction scheduling. *ACM Comput. Surv.* 52, 3 (2019), 62:1–62:50. DOI: https://doi.org/10.1145/3200920

[24] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2016. Register allocation and instruction scheduling in unison. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, 263–264. DOI: https://doi.org/10.1145/2892208.2892237

[25] Roberto Castañeda Lozano and Christian Schulte. 2019. Survey on combinatorial register allocation and instruction scheduling. *ACM Comput. Surv.* 52, 3, Article 62 (jun 2019), 50 pages. DOI: https://doi.org/10.1145/3200920

[26] Rahul Nagpal and Y. N. Srikant. 2008. Pragmatic integrated scheduling for clustered VLIW architectures. *Softw. Pract. Exper.* 38, 3 (mar 2008), 227–257. DOI: https://doi.org/10.5555/1345485.1345486

[27] GUROBI OPTIMIZATION. 2023. GUROBI Solver. Retrieved from http://www.gurobi.com/

[28] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-NoÃńl Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. Andreas Krall and Thomas R. Gross (Eds.). ACM, 168–182. DOI: https://doi.org/10.1145/3178487.3178500

[29] Zili Shao, Bin Xiao, Chun Xue, Qingfeng Zhuge, and Edwin H.-M. Sha. 2006. Loop scheduling with timing and switching-activity minimization for VLIW DSP. *ACM Trans. Des. Autom. Electron. Syst.* 11, 1 (jan 2006), 165–185. DOI: https://doi.org/10.1145/1124713.1124724

[30] Ghassan Shobaki, Vahl Scott Gordon, Paul McHugh, Theodore Dubois, and Austin Kerbow. 2022. Register-pressure-aware instruction scheduling using ant colony optimization. *ACM Trans. Archit. Code Optim.* 19, 2 (2022), 23:1–23:23. DOI: https://doi.org/10.1145/3505558

[31] Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. 2020. Optimizing occupancy and ILP on the GPU using a combinatorial approach. In *18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO'20)*. ACM, 133–144. DOI: https://doi.org/10.1145/3368826.3377918

[32] Ghassan Shobaki, Maxim Shawabkeh, and Najm Eldeen Abu Rmaileh. 2013. Preallocation instruction scheduling with register pressure minimization using a combinatorial optimization approach. *ACM Trans. Archit. Code Optim.* 10, 3, Article 14 (sep 2013), 31 pages. DOI: https://doi.org/10.1145/2512432

[33] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274. DOI: https://doi.org/10.1109/71.993206

[34] Huijun Wang and Oliver Sinnen. 2018. List-scheduling versus cluster-scheduling. *IEEE Trans. Parallel Distributed Syst.* 29, 8 (2018), 1736–1749. DOI: https://doi.org/10.1109/TPDS.2018.2808959

[35] Yaohua Wang, Chen Li, Chang Liu, Sheng Liu, Yuanwu Lei, Jian Zhang, Yang Zhang, and Yang Guo. 2021. Advancing DSP into HPC, AI, and beyond: Challenges, mechanisms, and future directions. *CCF Transactions on High Performance Computing* 3 (03 2021). DOI: https://doi.org/10.1007/s42514-020-00057-2

[36] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. *SIGPLAN Not.* 35, 5 (may 2000), 121–133. DOI: https://doi.org/10.1145/358438.349318

[37] Shu Xiao, E. M.-K. Lai, and A. P. Vinod. 2005. VLIW instruction scheduling for DSP processors based on rough set theory. In *Proceedings of the Eighth International Symposium on Signal Processing and Its Applications, 2005.*, Vol. 1. 311–314. DOI: https://doi.org/10.1109/ISSPA.2005.1580258

[38] Xuemeng Zhang, Hui Wu, and Jingling Xue. 2011. An efficient heuristic for instruction scheduling on clustered vliw processors. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Taipei, Taiwan) *(CASES '11)*. Association for Computing Machinery, New York, NY, USA, 35–44. DOI: https://doi.org/10.1145/2038698.2038707

[39] Yi Zhao, Suzhi Cao, and Lei Yan. 2019. List scheduling algorithm based on pre-scheduling for heterogeneous computing. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. 588–595. DOI: https://doi.org/10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00089

[40] Huichuan Zheng, Hao Zhang, Shuo Xu, Fanjin Xu, and Mengying Zhao. 2022. Adaptive mode transformation for wear leveling in nonvolatile FPGAs. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 41, 11 (nov 2022), 3591–3601. DOI: https://doi.org/10.1109/TCAD.2022.3197685

Received 10 November 2023; revised 17 January 2024; accepted 21 January 2024