CIExplorer: Microarchitecture-Aware Exploration for Tightly Integrated Custom Instruction

Xiaoyu Hao
University of Science and Technology of China
Hefei, China
haoxiaoyu@mail.ustc.edu.cn

Sen Zhang
University of Science and Technology of China
Hefei, China
sen@mail.ustc.edu.cn

Liang Qiao
University of Science and Technology of China
Hefei, China
ql1an9@mail.ustc.edu.cn

Qingcai Jiang
University of Science and Technology of China
Hefei, China
jqc@mail.ustc.edu.cn

Jun Shi
University of Science and Technology of China
Hefei, China
shijun18@ustc.edu.cn

Junshi Chen*
University of Science and Technology of China
Hefei, China
Laoshan Laboratory
Qingdao, China
cjuns@ustc.edu.cn

Hong An*
University of Science and Technology of China
Hefei, China
Laoshan Laboratory
Qingdao, China
han@ustc.edu.cn

Xulong Tang
University of Pittsburgh
Pittsburgh, USA
tax6@pitt.edu

Hao Shu
NIO
Shanghai, China
ming.liu3@nio.com

Honghui Yuan
NIO
Shanghai, China
ethan.song@nio.com

Abstract

Extending existing architectures with customized instruction extensions is emerging to achieve high performance and energy efficiency for specific applications. Automated discovery of custom instructions (CIs) is well-studied nowadays, which requires exploring combinations of different types and quantities of operations, resulting in a vast search space. However, previous works typically use microarchitecture-agnostic cost models, leading to suboptimal CIs that may degrade performance. They leverage graph isomorphism to reduce area overhead, but few of them consider its potential to benefit performance-oriented exploration. To this end, we present CIExplorer, a framework for adaptive CI exploration. We propose a Seed Growth Method (SGM) based on a genetic algorithm to discover CIs with the consideration of graph similarity. We also propose a compiler-assisted modeling strategy that applies a microarchitecture-aware cost model to estimate the potential benefits of CIs in exploration. We evaluate our framework using various benchmarks in SPEC2006 and Mediabench on in-order, 2-wide OOO, and 4-wide OOO processors. Experimental results demonstrate that CIExplorer achieves average performance improvements of 1.09× and 1.13× and energy improvements of 1.07× and 1.10× compared with Novia [56] and MaxClique [57].

CCS Concepts

• **Computer systems organization** → **Architectures**.

Keywords

Custom instruction, Microarchitecture, HW/SW codesign

ACM Reference Format:
Xiaoyu Hao, Sen Zhang, Liang Qiao, Qingcai Jiang, Jun Shi, Junshi Chen, Hong An, Xulong Tang, Hao Shu, and Honghui Yuan. 2025. CIExplorer: Microarchitecture-Aware Exploration for Tightly Integrated Custom Instruction. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3730421

*Corresponding author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730421

1 Introduction

There is a growing trend of extending a general-purpose instruction set architecture (ISA) with a custom extension (ISAX) to improve performance and energy efficiency for specific applications. Two mainstream on-chip ISA specialization techniques that differ from the scope of target code regions, including fine-grained custom functional unit (FU) [2, 12, 14, 16] and coarse-grained built-in accelerators [27, 28, 35, 42, 45], have been proposed in both industry and academia. Compared with the latter, custom FU, which is tightly coupled with general-purpose processors (GPP), leads to better resource utilization [18]. Designing custom instructions (CIs) requires identifying commonly and frequently executed operations and their combinations with the consideration of architectural constraints, demanding substantial domain expertise and manual effort.

Automated CI exploration has been well-studied. Existing approaches [31] typically first convert the source code to dataflow graphs (DFGs) and then enumerate subgraphs under specific architectural constraints, including register ports and bit width of ISA instruction format, which limits subgraphs' live-ins and live-outs [7, 22, 23, 60]. After that, a selection phase is required [9, 13, 32] to select a subset of subgraphs according to predefined strategies, such as speedup. There is also a trend to relax the constraints of live-ins and live-outs to find maximal convex subgraphs [6, 48, 57]. Convexity means no path is outside the subgraph between any two nodes[1] [49]. This trend can be attributed to two primary factors. First, MaxClique [57] has proven that the speedup and size of a subgraph are monotonic for RISC single-issue processors, which means increasing the number of nodes of subgraphs never degrades performance. Second, custom architectural registers can be introduced [26] to implement CIs, which allows a subgraph to have more live-ins and live-outs than register ports. In this case, data-moving instructions are required to communicate between general-purpose registers (GPRs) and special-purpose registers (SPRs) [56]. Multiple-cycle register access is also a solution, which is formulated as an I/O serialization problem [50, 57].

Selecting subgraphs of relaxed live-in/out constraints suffers the following challenges. First, CIs can be extracted from any part of an application, necessitating searching the entire code to identify subgraphs with potential benefits, which contributes to a vast search space. Second, CIs should maximize performance benefits while incurring minimal area overhead. CIs benefit GPPs in many aspects, such as decreasing dynamic instructions, reducing register accesses, and exploiting instruction-level parallelism (ILP). Adding more operations to a CI may enhance these benefits, but it requires more area for hardware implementation of custom FUs. Sharing datapaths can reduce area overhead and raise resource utilization [4, 15, 44, 51], which involves checking graph isomorphism that is an NP-complete problem [29]. Third, as the microarchitectures of modern superscalar GPPs become increasingly complex, employing oversimplified or microarchitecture-agnostic cost models can be inadequate. Cost models should capture the behaviors of host architectures before and after integration with CIs to evaluate benefits and determine the quality of selected subgraphs.

Previous works fail to address these challenges from three perspectives. First, subgraphs need to be enumerated at various granularity to accommodate GPPs of diverse microarchitectures. A predefined enumeration strategy or constraint may not be suitable for different target architectures. The theorem arguing the relationship between speedup and subgraph size leads to the trend of finding maximal convex subgraphs, but this only works for single-issue GPPs [57]. For instance, parallel operations in a subgraph are inherently exposed by out-of-order (OOO) execution of high ILP. Second, selecting good ones from those enumerated subgraphs relies on the cost model. Current cost models typically measure achieved speedup or reduced delay by CIs [31, 59], and they fail to consider some essential microarchitectural features, e.g., pipeline width. Furthermore, there is a gap between selected subgraphs and ISA extension. Explicit data-moving instructions affect slot usage, and a basic block may have multiple CI instances, which can hardly be modeled without a proper representation of CIs. As a result, they produce incorrect estimation results and fail to select subgraphs for different microarchitectures adaptively. Third, sharing hardware resources is an important problem to resolve. Works [4, 9, 13] directly enumerate isomorphic and convex subgraphs through exact algorithms. A more flexible and widely used technique is a two-step selection+fusion paradigm [5, 30, 43, 44], which first selects subgraphs and then fuses them. This technique can share areas between non-isomorphism graphs. However, current works only consider graph isomorphism to reduce area overhead, which can also benefit performance-oriented exploration.

To this end, we propose a framework named **C**ustom **I**nstruction **Explorer** (CIExplorer) for performance-oriented CI exploration. We propose a Seed Growth Method (SGM) based on the genetic algorithm (GA) to efficiently identify **graphs of same order (GSO)** — graphs have identical numbers and types of vertices. Exploiting GSO reduces complexity compared with checking graph isomorphism and enables both modeling interactions between subgraphs and sharing hardware resources. SGM grows seeds (two-node recurring connected subgraphs) into larger subgraphs of varying granularity, suitable for microarchitectures of different ILP. We also propose a Split-and-Combine Method (SCM) to find CIs for IO GPPs and extract subgraphs for basic blocks without

[1] In this paper, we utilize instruction and node interchangeably.

seeds. A microarchitecture-aware cost model is established to prioritize subgraphs for GPPs of different microarchitectures, measuring static instruction sequences consisting of single or duplicated basic blocks. We implement a compiler-assisted intermediate representation (IR) specialization strategy based on LLVM toolchain [37] to automatically generate both custom and corresponding data-moving instructions, bridging the gap between identified subgraphs and ISA extensions. Specialized IR is used as the input to the cost model. In our experiments, we evaluate CIExplorer on in-order (IO), two-wide OOO (OOO2), and 4-wide OOO (OOO4) GPPs using benchmarks from SPEC2006 [1] and MediaBench [39]. Our method achieves an average of 1.09× and 1.13× performance improvement compared with Novia [56] and MaxClique [57], and provides 1.07× and 1.10× energy improvement. Contributions of this paper are summarized as follows:

- We establish a framework named CIExplorer to quickly and exhaustively identify commonly and frequently executed subgraphs as CIs in an iterative way.
- We propose SGM based on GA to discover **graphs of same order** instead of isomorphic graphs, which relaxes the constraint and reduces computational complexity, enabling performance-oriented and resource-sharing exploration simultaneously. We also propose SCM to find CIs from basic blocks without recurring subgraphs and optimize it for finding maximal subgraphs suitable to IO GPPs.
- We propose a new evaluating strategy for selecting CIs by using the compiler to customize and generate static instructions as input to a novel microarchitecture-aware cost model, which is lightweight enough to be applied to a search process.

2 Background and Motivation

2.1 Fine-grained ISAX

Figure 1 shows the relationship between GPP and custom FU for fine-grained ISAX. CIs are identified within a basic block and they can access GPRs and SPRs. The former requires operands to meet the host ISA's constraints in terms of operand number and instruction bit width, and microarchitecture limitations such as register port. The latter needs additional data-moving instructions to communicate data between GPRs and SPRs.

2.2 ISA Extension

Figure 2A shows three 32-bit RISC instructions: `PACK`, `EXE`, and `EXTRACT`, which are derived from Novia [56]. They are general enough to cover most scenarios for fine-grained ISAX, which is the architecture our paper aims at. Prefixes `sp_` and `gp_` indicate SPR and GPR, respectively. Instruction `PACK` moves data from GPR to SPR, which reads the value of `sp_rs`,

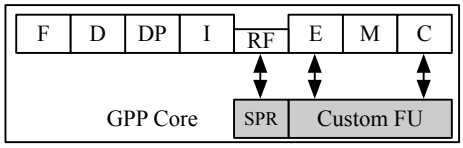
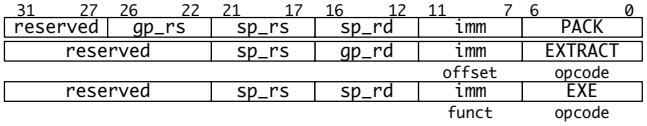


Figure 1: Tightly integrated custom FU.



(A) Instruction Format

```
I0:   R2  = mul     R0, R1
I1:   R5  = sub     R3, R4
I2:   R6  = add     R2, R5
```

(B) Original Code

```
# insert R0 into the 0th position of SP0
I0:   SP0 = PACK    #0, SP0, R0
# insert R1 into the 1st position of SP0
I1:   SP0 = PACK    #1, SP0, R1
I2:   SP0 = PACK    #2, SP0, R3
I3:   SP0 = PACK    #3, SP0, R4
# invoke custom instruction of ID #8 that uses data in SP0
I4:   SP1 = EXE     #8, SP0
# extract 0th element of SP1, and save it into R6
I5:   R6  = EXTRACT SP1, #0
```

(C) Customized Code

Figure 2: (A) shows typical RISC 32-bit custom instruction formats. Prefixes `sp_` and `gp_` represent special-purpose and general-purpose. (B) shows an instruction sequence. (C) shows the customized code by fusing `I0`-`I2` in (B) as a CI.

replaces `sp_rs[offset]` with `gp_rs`, and loads it into `sp_rd`. Instruction `EXTRACT` moves data from SPR to GPR (`gp_rd = sp_rs[offset]`). Instruction `EXE` invokes the execution of a CI specified by an ID (funct), which consumes data of `sp_rs` and writes results into `sp_rd`. Figure 2B shows an instruction sequence containing three instructions, which are fused as a CI. Figure 2C shows the customized code. This CI has four input operands and one output operand. Instructions `I0`-`I3` in Figure 2C move data from GPR (R0, R1, R3, and R4) to the first four positions of SP0. `I4` invokes the computation specified by an ID #8, which produces a result stored in the first element of SP1. `I5` moves data from SP1 to R6, which can be consumed by following instructions.

2.3 Issues of Current Approaches

Motivating Example: Novia [56] has been proposed recently, which directly fuses basic blocks and then splits fused

basic blocks into subgraphs according to memory and branch instructions. We evaluate Novia on IO, OOO2, and OOO4 GPPs using SPEC2006 [1] and MediaBench [39] in our simulator. Results are shown in Figure 3. Negative values mean performance degradation because of CIs. Novia significantly improves performance for all benchmarks on IO GPP. However, it negatively impacts performance for many benchmarks on OOO2 and OOO4 GPPs because it can not adjust discovered subgraphs for microarchitectures. Adaptive exploration for different microarchitectures requires a method to enumerate subgraphs of varying granularity and a cost model to capture microarchitectural behaviors.

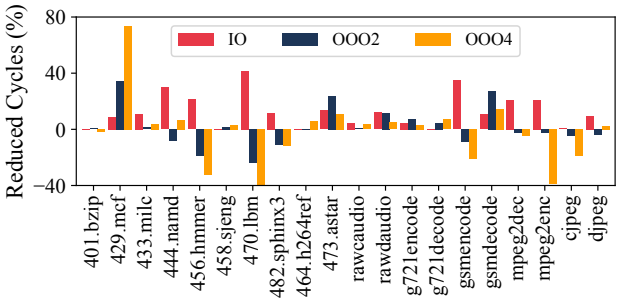


Figure 3: Percentage of reduced cycles by integrating CIs found by Novia on IO, OOO2, and OOO4 GPPs evaluated using SPEC2006 [1] and MediaBench [39].

Graph Enumeration: Checking graph isomorphism poses a challenge for enumerating subgraphs. Existing works mainly focus on exploiting graph isomorphism to reduce hardware area and promote resource utilization. There is still a chance to use it to achieve better performance. Interactions among CIs corresponding to recurring subgraphs within a single basic block may impact overall performance gains, particularly for unrolled code. Meta-heuristic algorithms [41, 49, 59, 61] are used to explore CIs, which are scalable, effectively trade off search time and quality, and can find subgraphs of varying granularity. Thus, we propose a new idea for finding GSO to address both resource-sharing and performance-oriented exploration based on GA.

Cost Model: We can conclude from a survey [31] that many studies apply Equation (1) or its variants as cost models that are microarchitecture-agnostic for subgraph selection. $SW(\cdot)$ represents the sum of node latencies of the entire subgraph, $HW(\cdot)$ denotes the accumulated cycles on the critical path after specialization, and $Comm(\cdot)$ measures the communication time of input and output operands. It poses some issues. First, $SW(S)$ models superscalar GPPs incorrectly. In Figure 2B, instruction `I0` can hide latency of `I1` in a superscalar GPP, but $SW(S)$ cannot capture this behavior by assuming instructions are executed sequentially. Instruction `I1` can start execution when its operands are ready, but it needs to wait for two more operands of `I0` to be ready when fused as a CI. This can be regarded as an implicit instruction-level synchronization that current cost models can not capture. Second, this cost model only evaluates standalone subgraphs without considering other instructions of basic blocks. A problem is that when introducing CIs, the order of other instructions may be changed to maintain the proper data dependency. Third, even though $Comm(S)$ is introduced, it can not model the impact of CIs on slot usage. Figure 2C shows four `PACK` instructions and one `EXTRACT` instruction, which occupy bandwidth for various pipeline stages such as fetch and commit. These factors show a gap between selected subgraphs and the real ISA extension. Thus, we propose an IR specialization method, automatically generating custom and data-moving instructions for instruction formats in Figure 2A. We also establish a graph-based cost model taking specialized IR as input and capable of modeling behaviors of different microarchitectures to score subgraphs.

$$speedup = \frac{SW(S)}{HW(S) + Comm(S)} \tag{1}$$

2.4 Graph-Based Microarchitecture Modeling

Dynamic event-dependence graph (DEG) is widely used for various tasks, such as revealing bottlenecks in micro executions [20], exploring the design space for microarchitecture parameters [8, 25], and modeling hardware specialization [25, 46]. Figure 4 shows a DEG that models the instruction sequence listed in Figure 2B on an OOO2 GPP. Nodes represent pipeline stages and edges model pipeline, structure, and data dependencies. Edge weights are delayed cycles. Structural dependencies include horizontal edges to model limitations of fetch/commit windows, such as the edge between F_{mul} and F_{sub}. Data dependency exists between E_{sub} and E_{add}. Modeling focuses on latency measurement via critical path analysis that finds the longest path from the fetch of the first instruction (F_{mul}) to the commit of the last instruction (C_{add}). This figure shows two critical paths of 5 cycles highlighted in red.

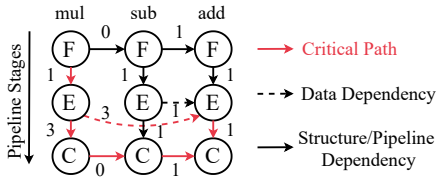


Figure 4: A DEG built for instructions shown in Figure 2B on an OOO2 GPP. The critical path is 5 cycles, assuming 3-cycle mul and 1-cycle add/sub.

In this paper, we specialize the DEG for estimating performance gains of CIs by introducing two innovations. First, we build the DEG from static code instead of traces of microarchitecture events from cycle-level simulators. Trace-based methods only model behaviors of the host simulator, which requires modifying the simulator to model new architectural features such as execution ports. Second, we extend DEG to model custom FUs using the output of our IR specialization method, which decouples ISAX representation and graph building. By doing this, we can manipulate the order of instructions to maintain correct data dependency after introducing CIs and enable the measurement of entire specialized basic blocks consisting of multiple CI instances. Compared with TDG [46] and Calipers [25] proposed for accurate timing evaluation for hardware specialization by directly manipulating DEGs, our method aims to capture different microarchitectural behaviors rapidly.

3 Framework Overview

In this section, we introduce several essential terms, show the main concept of our method, and then show the workflow of CIExplorer.

3.1 Main Concept

Graphs of same order (GSO): Graphs of same order have identical numbers and types of nodes. Let $G1$ be a directed acyclic graph with nodes $x1, x2, x3$ and edges $x1x2, x1x3$. Let $G2$ be another graph with nodes $x1, x2, x3$ and edges $x1x3, x2x3$. $G1$ and $G2$ are graphs of order 3, but are not isomorphic. GSO has a relaxed constraint compared with the isomorphism graph, which can cover more code regions. We can measure entire basic blocks containing multiple instances of GSO to determine the benefit by considering their interactions. Even though instances of GSO are not isomorphic, a previously proposed merging procedure is applied to generate datapaths for GSO to reduce area overhead.

Seed and Seed Component: A seed is defined as a set $S = \{sc_0, sc_1, ..., sc_n\}$, where sc_i is a seed component, a recurring subgraph containing only two connected nodes. Panel A of Figure 5 presents a seed containing four seed components (subgraphs 0-3).

Candidate and Candidate Component: A candidate is also a set defined as $R = \{g_0, g_1, ..., g_m\}$, where g_i donates a candidate component. A candidate component is a subgraph grown from a seed component. Panel B of Figure 5 shows a candidate with four candidate components (subgraphs 4-7). Notably, for a R grown from S, each candidate component $g_i \in R$ is uniquely associated with one seed component $sc_j \in S$.

Seed Growth: One seed can grow into many candidates. Panels A-C of Figure 5 represent three possible candidates,

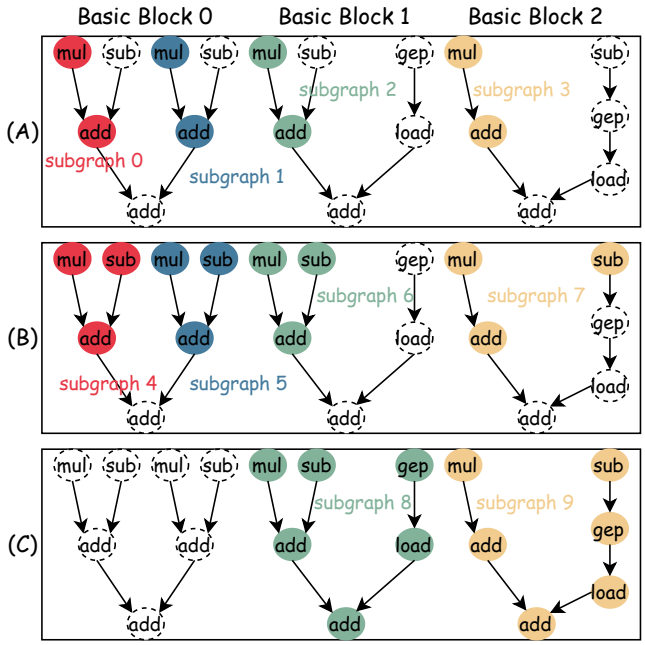


Figure 5: (A) A seed contains four seed components (subgraphs 0-3). (B) Subgraphs 4-6 are isomorphic. Subgraph 7 is a GSO of them, but not connected. They are grown from subgraphs in (A). (C) Subgraphs 8 and 9 are also GSOs. Subgraphs grown from subgraphs 0 and 1 are discarded because they are unable to grow into GSOs of subgraphs 8 or 9.

including the seed itself. However, according to the cost model, only one of them will be selected as a CI. Panel B shows subgraphs 4-7 that grow from the subgraphs in panel A by adding one additional sub instruction each. In this case, subgraphs 4-6 are isomorphic, but not subgraph 7, which has the same numbers and types of nodes. Panel C shows two GSOs grown from Subgraphs 2 and 3. No subgraph can be grown into GSO of subgraphs 8 or 9 in basic block 0, so subgraphs 0 and 1 and their grown results are discarded. Thus, for a R grown from S, $|R| \leq |S|$. Candidate components are generated simultaneously without any overlap between any two of them. Moreover, the candidate component whose score does not meet the predefined lowest threshold will be discarded. Basic block 0 in panel C is not customized, which can be handled by extracting seeds and growing them again. Notably, even though our method can find subgraphs containing memory instructions, only arithmetic instructions are considered to be CIs in this paper. Solving memory disambiguation through LSU, which requires allocating and releasing slots corresponding to sub-memory operations, can be a problem, introducing additional complexity for datapaths. A solution is treating custom instruction exploration

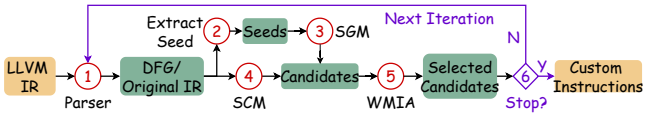


Figure 6: The workflow of CIExplorer

as a macro-op fusion problem, which requires different exploration rules [11].

Connectivity: Graphs of same order are required to be connected. When we add an instruction to a seed component, there may be more than one node in the DFG corresponding to that instruction, and we want the node being added to be as close to the seed component in the graph as possible. Hence, we achieve this by requiring connectivity. In this case, subgraph 7 is invalid.

3.2 Workflow

Figure 6 illustrates the workflow of CIExplorer. CIExplorer begins by taking LLVM [37] IR as input and parses static code into DFGs at the granularity of basic blocks (①). It also instruments and runs the program to collect executed counts of basic blocks. CIExplorer then extracts recurring subgraphs of two connected nodes as seeds (②) and conducts SGM (③). CIExplorer employs SCM to split basic blocks without seeds (④). The compiler-assisted modeling strategy for selecting subgraphs is applied in SGM and SCM. Candidates may overlap since SGM is applied to seeds separately, so Weighted Maximum Independent Set Analysis (WMIA) (⑤) is used to select a set of candidates without overlap. The entire search is performed iteratively to avoid missing possible CIs. After exploration finishes, a merging procedure is applied to generate datapaths for CIs. Note that we use an optimized SCM to find maximal convex subgraphs for IO GPPs, eliminating the need for iterative execution. Specifically, the optimized implementation only executes steps ① and ④.

4 Seed Growth Method

4.1 Workflow

Figure 7 shows the entire workflow to apply Seed Growth Method (SGM) to grow seeds into candidates. S and sc_i donate a seed and a seed component. R and g_i donate a candidate and a candidate component. In panel A of Figure 7, we apply SGM to each seed individually to get a candidate. Panel B of Figure 7 shows the main steps of SGM implemented based on GA, which executes iteratively until reaching a specific iteration or a predefined condition and then returns the best candidate.

Seed Extraction: We extract seeds by enumerating recurring instruction pairs with data dependency and counting their occurrences. These instruction pairs may overlap, which is addressed by removing those that are later in the program order.

Initialize Population: An individual is a subgraph representing a candidate component grown from a seed component. Figure 7's panel E illustrates a basic block of eight nodes, which has a subgraph of four nodes marked with solid borders. It is an individual encoded as `01011100`. We initialize GA using the encoding of sc_x, a pivot selected from seed S, whose corresponding basic block has maximal size. In the initial population, many individuals are randomly generated, but they are ensured to contain sc_x.

Evaluate Fitness: As shown in panel C of Figure 7, this step calls a fitness function to each individual g_x, which outputs a candidate R and its score. GA will track the R with the maximal score as the final best candidate. Individuals of a population are derived from sc_x since we initialize the GA with it. Panel D of Figure 7 illustrates the procedure of the fitness function based on the idea of growing a seed component sc_x into a larger subgraph g_x and checking if sc_i can grow into g_i that is a GSO of it. In this case, each g_i corresponds to only one sc_i as shown. The final fitness is the output of a cost model, which scores the entire R. Panel F of Figure 7 shows the implementation of the fitness function. Since a basic block may contain several seed components, we assign each basic block a bit vector to indicate if an instruction has been visited to avoid creating overlapping candidate components. We apply function `grow` to each sc_i to get a g_i that is a GSO of g_x. Function `grow` returns empty if growing fails. Details of `grow` are shown in Section 4.2. Notably, R will be empty if g_x is not connected.

Select/crossover/mutate: These operations with different strategies generate a new population for the next iteration, which selects individuals of higher scores, exchanges parts of encodings between them, and flips bits of encodings, respectively. One detail is that each individual will have a R and a score after evaluating fitness, and only the score is used to guide these operations.

4.2 Grow Function

We present function `grow` in Algorithm 1, where g_i is the output subgraph that keeps growing. sc_i and g_x donate a candidate component and a target subgraph to grow, respectively. g_i grows according to *nodes* donating newly added nodes. We ensure the connectivity of g_i in `grow`, which poses an issue that the order of adding nodes to a subgraph impacts connectivity checking. For example, if a basic block has a subgraph containing a sequence of `sub-mul-div` and we have an instruction `sub` at the beginning. To grow `sub` to this sequence, we need to add `mul` first and then `div` to get a connected subgraph. Thus, we implement `grow` based on the idea of keeping adding nodes of various types to the g_i

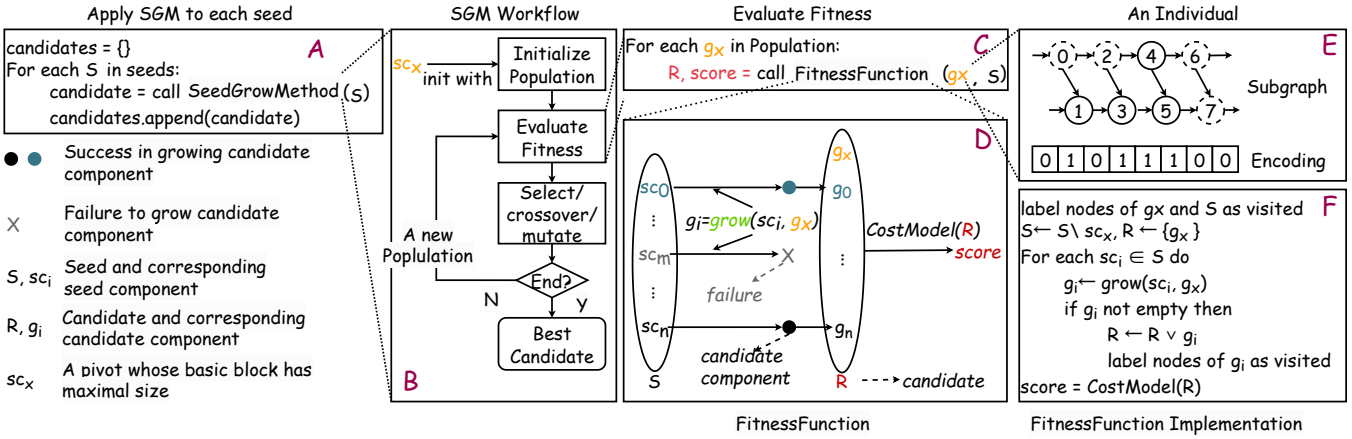


Figure 7: Entire workflow of applying SGM to grow seeds into candidates.

until no node can be added or *nodes* becomes empty as the while loop in line 3. It allows adding nodes in different orders. We enumerate $node_i$ and check if $nodes_{BB}$ has the same node, and this node is connected to g_i. $nodes_{BB}$ are nodes of the basic block to which sc_i belongs, which have not been added to any other subgraphs. The function `checkSameInst` checks if $node_i$ and $node_j$ share the same opcode. Function `checkConnect` checks if g_i is still connected after adding $node_j$. Finally, if g_i and g_x have different numbers of nodes, then g_i and g_x are not GSOs, and we set g_i to empty.

5 Split-and-Combine Method

Not all basic blocks contain seeds, and additionally, some basic blocks no longer have seeds after extracting subgraphs, so we propose a Split-and-Combine Method (SCM) to handle these cases. SCM is shown in Algorithm 2. SCM takes a basic block's DFG (*graph*) as input and outputs *candidates*. This method extracts connected subgraphs by removing branch and memory instructions in `split` function (line 1) and then enumerates all combinations of them through a while loop. Function `combine` builds a new subgraph with instructions from *candidate* and *subgraphs*[j]. Function `evaluate` returns true if *combined* is convex, and the score of *combined* is higher than *candidate*, *subgraphs*[j], and a predefined *threshold*. SCM continues with the last successful combined result (line 13). If too many connected subgraphs are extracted, it can be time-consuming to enumerate their combinations, so we directly use convex connected subgraphs containing more than one node as candidates. Notably, we have optimized SCM for IO GPP to find maximal convex subgraphs. In this case, `evaluate` only checks if *combined* is convex instead of pursuing a higher score, but threshold checking in line 16 is preserved to remove low-scoring subgraphs. SGM is omitted for this simple exploration target.

Algorithm 1: Grow Function

Input: sc_i — a seed component, g_x — target subgraph
Output: g_i — a candidate component that is a GSO of g_x

```
1  g_i ← sc_i, nodes ← g_x \ sc_x, terminate ← 0
2  nodes_BB ← not visited nodes of the basic block to which sc_i belongs
3  while nodes.size() > 0 ∧ ¬terminate do
4      terminate ← 1
5      for node_i ∈ nodes do
6          for node_j ∈ nodes_BB do
7              same ← checkSameInst(node_i, node_j)
8              connect ← checkConnect(node_j, g_i)
9              if same ∧ connect ∧ node_j ∉ g_i then
10                 nodes ← nodes \ node_i
11                 g_i ← g_i ∨ node_j
12                 terminate ← 0
13                 break
14 if g_i.size() ≠ g_x.size() then
15     g_i ← ∅
```

6 Cost Model

6.1 Compiler-Assisted IR Specialization

We propose a compiler-assisted method to specialize IR for CIs in single or multiple basic blocks, which will be the input of the cost model. We implement it based on instruction formats shown in Figure 2A using LLVM toolchain. It supports an arbitrary number of `PACK` operands. Since we use static DFGs constructed by LLVM to perform SGM, the IR specialization method is easy to apply. Figure 8 shows the specialized IR of a CI having two sub-instructions `fmul` and `fadd`. Function `fmul_fadd` represents the CI, and `PACK` moves three operands at once. Data-moving instructions are represented by function `PACK` and IR `extractvalue` (`EXTRACT`), respectively. Due to LLVM IR's limitation, each `extractvalue` can only extract a single value each time. The specialized IR is

Algorithm 2: Split-and-Combine Method

Input: graph — DFG of a basic block
Output: candidates

```
1  subgraphs ← split(graph)
2  visit ← ∅, candidates ← ∅
3  for i = 0 to subgraphs.size() do
4      if i ∈ visit then
5          continue
6      candidate ← subgraphs[i], terminate ← 0
7      while ¬terminate do
8          terminate ← 1
9          for j = i + 1 to subgraphs.size() do
10             combined ←
                 combine(candidate, subgraphs[j])
11             improve ← evaluate(combined)
12             if improve ∧ j ∉ visit then
13                 candidate ← combined, visit ← visit ∨ j
14                 terminate ← 0
15     visit ← visit ∨ i
16     if candidate.score > threshold then
17         candidates ← candidates ∨ candidate
```

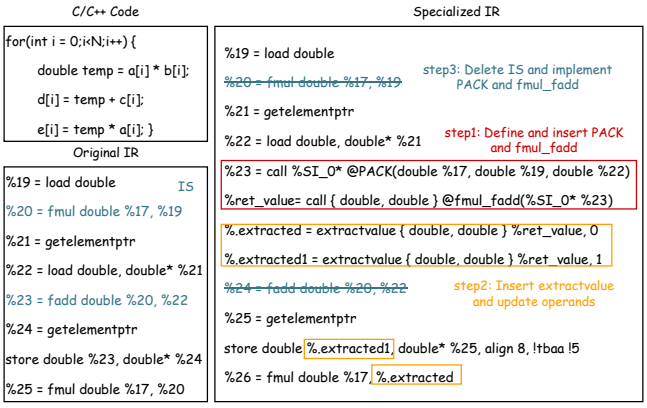


Figure 8: Example of specializing a CI by fusing an instruction sequence containing fmul and fadd.

generated by iterating candidate components. Each iteration applies the steps as shown in Figure 8 to one candidate component, utilizing original IR or specialized IR output from the previous iteration as its input. Let *IS* be an instruction sequence to be fused as CI.

In the first step, we define and insert PACK and CI. CI denotes a function representing a custom instruction, such as `fmul_fadd` in Figure 8. PACK creates a structure based on its input parameters, such as PACK in Figure 8 returns a structure of three double types. Function CI takes structures returned by PACK as inputs and also returns a structure based on output operand types of *IS* such as `fmul_fadd` return a structure of two double types. We insert PACK functions right after the last instruction they depend on, and insert CI after them. Notably, if the number of input operands is not larger than a predefined value, no PACK will be created, and if the number of output operands is one, `ret_value` will be a single value.

In the second step, we use `extractvalue` to extract values from `ret_value` and update users of them, such as the instruction `store` using `%.extracted`. Moreover, we manipulate the order of instructions to make data dependency right. For example, if an instruction `I0` that uses values produced by CI precedes CI after specialization, then we use the function `moveInstAfter(I0, CI)` to move `I0` after CI. This function also checks whether the user (`I1`) of `I0` precedes `I0`. If so, the function calls `moveInstAfter(I1, I0)` again.

In the third step, we remove *IS* and finish implementing PACK and CI. In PACK, we allocate memory for structures with malloc and use `inservalue` to initialize values. CI has only one basic block containing three parts. The first part contains `extractvalue` to unpack data from input parameters of structure types. The second part is a copy of *IS* whose operands are updated with those unpacked data to ensure correct functionality. The third part uses `insertvalue` to create `ret_value` and returns it.

6.2 DEG Construction

We build DEGs based on original or specialized IR with two assumptions that all memory operations hit L1Cache and branches never change control flow, like existing static analysis methods [38, 55]. Table 1 shows edge types and corresponding weights. Algorithm 3 shows how we build a DEG for an instruction sequence. Unlike current methods based on simulation traces [8, 25, 46], the exact time when architectural events occur is unavailable. In OOO execution, building DEG with instructions following program order may cause wrong estimated cycles because an instruction may request a resource like an ALU before preceding instructions. We propose to dynamically prioritize instructions by the time of entering the execution stage using `getEarliestExecutionInsn`. It adds nodes of each instruction iteratively and separately to DEG to get the cycle when they start executing in a pseudo way (not really changing DEG) and selects the earliest one (line 5). To reduce computational complexity, instructions are separated into chunks of the size of pipeline width (line 3). We build a node and all corresponding edges before another (line 6). We build resource edges from lines 8 to 12 by managing a scoreboard that records all hardware resources and the latest access instructions. In each stage, when an instruction requests a resource, we find the earliest available resource of the type in a round-robin way, get the instruction that already occupies it, and then build an edge between the related stages of the two instructions. After that, we update the scoreboard.

Table 1: Basic edges of pipeline, structure, and data dependencies.

Edge	Weight	Description
$F(i) \rightarrow DP(i)$	1 cycle	Fetch instruction i.
$DP(i) \rightarrow E(i)$	1 cycle	Dispatch and schedule instruction i to execution.
$E(i) \rightarrow C(i)$	Latency of instruction i	Execute instruction i.
$E(i) \rightarrow E(j)$	1 cycle; Latency of instruction i; Latency of instruction i;	Instruction j requests a fully pipelined FU that instruction i releases; Instruction j requests a not fully pipelined FU that instruction i releases; True data dependency from instruction i to j.
$F(i) \rightarrow F(i+1)$	0 cycle; 1 cycle;	Instructions i and $i+1$ are in a fetch window; Instructions i and $i+1$ are in two consecutive fetch windows.
$C(i) \rightarrow C(i+1)$	0 cycle; 1 cycle;	Instructions i and $i+1$ are in a commit window; Instructions i and $i+1$ are in two consecutive commit windows.

Algorithm 3: DEG Construction Algorithm

Input: insns — A static instruction sequence of a single or duplicated basic blocks
Output: DEG — Built DEG

1 $i \leftarrow 0$
2 **while** $i < insns.size()$ **do**
3 $insns_{pipe} \leftarrow insns[i : i + pipewidth]$
4 **while** $insns_{pipe}$ *not empty* **do**
5 $insn \leftarrow getEarliestExecutionInsn(insns_{pipe})$
6 **for** $stage$ ***in*** $[fetch, dispatch, execute, commit]$ **do**
7 add the pipeline $stage$ node to DEG and build pipeline/data edges
8 $resouces \leftarrow getRequiredResources(insn, stage)$
9 **for** res ***in*** $resources$ **do**
10 $insn_{prev} \leftarrow scoreboard[res]$
11 build edges between nodes that have resource dependencies of $insn_{prev}$ and $insn$
12 update $scoreboard$ with $insn$
13 remove $insn$ from $insns_{pipe}$
14 update edges for stage commit nodes.
15 $i \leftarrow i + pipewidth$

Finally, we need to correct edges to ensure instructions are committed in order in line 14, since we add instructions prioritized by the cycle when starting execution.

Our method supports duplicating a basic block multiple times to model dependencies between loop iterations. Panel A of Figure 9 shows a basic block, where `add` depends on a value from the previous iteration, which can be captured by duplicating this basic block. After duplication, one static instruction may have several instances. Data dependencies are built for instructions with the latest instance of the ones they depend on.

6.3 Modeling Resource Dependency

The panel B of Figure 9 shows an edge between E_{I0} and DP_{In} to model slot usage of Issue Queue (IQ), which means `In` requests a slot that `I0` releases. We model resource dependency of slot usage of Reorder Buffer (ROB) by introducing a new vertex Release (R), as shown in panel C of Figure 9. We cannot directly model it with an edge $C_{I0} - DP_{In}$, since it introduces a cyclic ($C_{I0} - DP_{In}$, $DP_{In} - C_{In}$, and $C_{In} - C_{I0}$) when instruction `In` precedes `I0` in program order. Load Queue (LDQ) and Store Queue (STQ) are also modeled in this way by assuming slots are released when instructions load/store are committed.

Modern processors decode instructions into micro operations (uops), which execute on execution ports [3]. Each port is connected to a set of FUs. The left panel of Figure 10 shows FU mapping for a 2-wide OOO GPP. Port 0 is connected to an integer multiplier and an ALU. Port 1 is connected to a custom FU. Our DEG supports the model of this behavior by simplifying that an instruction always has one uop that uses a port for predefined cycles. The middle panel of Figure 10 shows the DEG built improved based on Figure 4, which models port usage in red edges. Since `mul` and `sub` compete for port 0, there is a red edge $E_{mul} - E_{sub}$. Another red edge $E_{sub} - E_{add}$ models that `add` request a port that `sub` releases. We model execution ports and FUs separately, so a blue edge $E_{sub} - E_{add}$ is built to indicate `add` request an ALU that `sub` releases. This DEG has a critical path of 5 cycles. The right panel of Figure 10 also shows a DEG by introducing a CI fusing `mul` and `add`, which uses the custom FU and eliminates resource dependencies due to FU and port usage. It has a critical path of 6 cycles, assuming 4-cycle `mul_add`.

6.4 Integration for Search

This section presents how we use DEG as a cost model for searching CIs. The score for each candidate can be concluded as Equation (2), where $constraint_i$ denotes the i-th constraint term. We use two boolean constraints, *convex* and *acyclic*, which are checked after GSOs are generated. The growing procedure ensures connectivity.

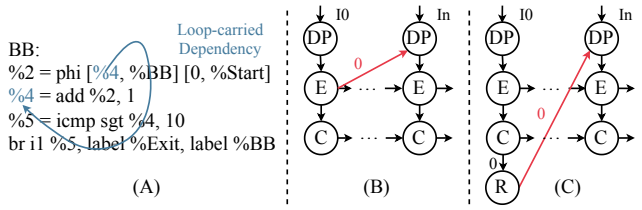


Figure 9: (A) Loop-carried dependency. (B) and (C) show structure dependencies due to IQ and ROB, respectively. A new vertex R is introduced to avoid loops.

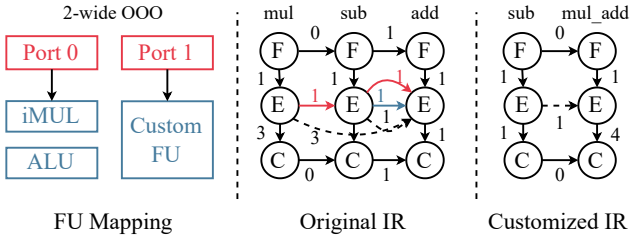


Figure 10: DEGs on a two-wide OOO GPP. Resource dependencies of Port and FU are modeled with red and blue edges.

$$score = \prod constraint_i \times speedup \tag{2}$$

$$speedup = \sum_{i}^{n} \frac{cycles^i_{orig}}{cycles^i_{cstm}} \times w_i \tag{3}$$

Equation (3) measures the weighted average *speedup*. n is the number of basic blocks, which is equal to or less than the number of candidate components because a basic block can have more than one candidate component. $cycles^i_{orig}$ and $cycles^i_{cstm}$ are measured cycles for original and customized basic blocks, which are duplicated by preset times. w_i is a weight for i-th basic block, calculated by Equation (4). $count_i$ is the executed count of the i-th basic block.

$$w_i = \frac{count_i}{\sum_i^n count_i} \tag{4}$$

7 Other Techniques

Remove Overlap: When applying SGM independently to each seed, it fails to consider the influence of other seeds during growth, resulting in overlapping candidates. We apply a method similar to [62] that uses the WMIA on the conflict graph to remove overlaps. Two candidates sharing nodes are considered in conflict.

Iterative Search: We apply an iterative way to exhaustively explore possible subgraphs. Each iteration progresses from Parser (①) to WMIA (⑤) in Figure 6. In Parser, we remove nodes of candidates identified in the previous iteration from DFGs before extracting seeds. Exploration stops if no seed is extracted or no candidate is selected.

Merging Procedure: Upon completing the exploration process, we employ the method introduced by [56] to construct datapaths by trying to merge all components of a candidate into a single datapath. We construct an additional datapath for any candidate component that results in a loop in the merged datapath.

Table 2: Benchmark

MediaBench	rawcaduio, rawdaduio, g721encode, g721decode, gsmencode, gsmdecode, mpeg2dec, mpeg2enc, cjpeg, djpeg
SPEC2006	401.bzip, 429.mcf, 433.milc, 444.namd, 456.hmmer, 458.sjeng, 464.h264ref, 470.lbm, 473.astar, 482.sphinx3

Table 3: Microarchitecture parameters for GPPs

GPP	**Specifications**
IO	single issue, out-of-order writeback
OOO2	2-wide pipeline, 64-entry ROB, 32-entry IQ, 1 ALU, 1 Mul/Div, 1 BRU, 1 FPU, 1 LD/ST
OOO4	4-wide pipeline, 168-entry ROB, 48-entry IQ, 3 ALU, 1 Mul/Div, 1 BRU, 2 FPU, 2 LD/ST
Memory	
L1DCache	32KB / 8-way / 5-cycle latency / deg-8 PC-based Prefetcher
L1ICache	32KB / 8-way / 5-cycle latency / Fetch-directed Prefetcher
L2Cache	256KB / 8-way / 8-cycle latency
L3Cache	8MB / 32-way / 35-cycle latency
DRAM	25GB/s BW / 118-cycle latency

8 Experimental Methodology

Benchmark: Various benchmarks are chosen from SPEC2006 [1] and MediaBench [39] to evaluate our method and conduct comparisons with existing techniques (see Table 2).

Simulation Method: We use an in-house cycle-level and trace-driven (memory addresses and branch decisions) simulator to evaluate the performance of GPPs and CIs. The simulator uses LLVM IR as the ISA and supports simulating customized IR as shown in Figure 8. We use McPAT [40] and GEM5-SALAM [52] to evaluate energy/power of GPPs and area of custom FUs' datapaths, respectively. The latency of a CI is measured as the accumulated cycles of the critical path of a subgraph. Due to additional experiments, we have

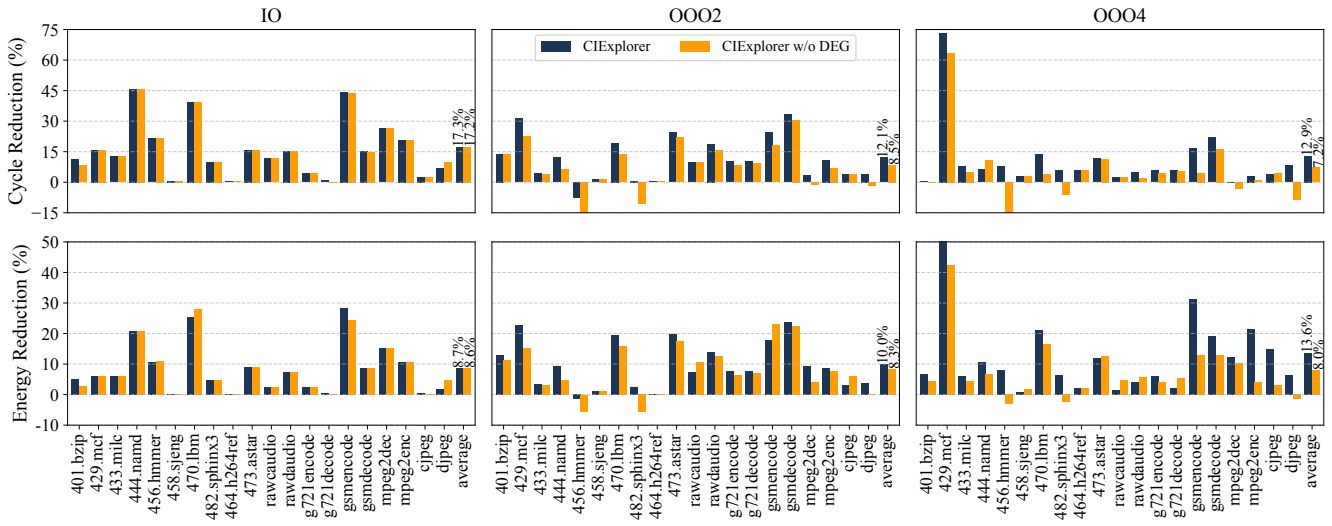


Figure 11: Percentage of decreased runtime cycles and energy for IO, OOO2, and OOO4 GPP cores. We also report results of CIExplorer without the DEG model.

found that the number of duplications for basic blocks has little effect on scoring, so we chose a value of 1 for efficiency.

Experiments are conducted on three cores, microarchitecture parameters of which are listed in Table 3. All cores use TAGE/ITTAGE branch predictor with a 12-cycle penalty and a common memory configuration. The in-order core has a scoreboard to enable out-of-order write-back. In experiments, we specify that each general or custom instruction requires only one execution port. A GPP has two ports for custom 1-cycle data-moving instructions. `PACK` instructions can move three operands at once. We use Simpoint [54] to choose the most representative chunk of code that contains 50 million instructions from a manually specified function for each benchmark. Specified functions are selected according to [36] and [45]. Notably, we apply only optimized SCM to IO GPP to discover maximal convex subgraphs.

9 Results

9.1 Performance and Energy Efficiency

Figure 11 presents the cycle and energy reduction of CIExplorer for IO, OOO2, and OOO4 GPP cores. We also report the results of CIExplorer without the DEG model (non-DEG for short) by replacing it with the cost model in Equation (1), which only measures subgraphs instead of entire basic blocks. Optimized SCM is also used for the non-DEG version. Negative values of figures mean performance and energy efficiency degradation because of the integration of CIs.

IO GPP: CIExplorer achieves a geometric mean of 17.3% cycle reduction and 8.7% energy reduction. Notably, comparable gains are observed for the non-DEG model (17.2% cycle reduction and 8.6% energy reduction), which is consistent with a previous observation [6] that a simple merit function can be as good as a complex one for IO GPP. This phenomenon stems from the inherent characteristics of IO GPPs that expose limited ILP. Fusing larger instruction patterns into CIs effectively hides more operations, delivering significant performance benefits. Our framework schedules specialized IR with a simple strategy that inserts CIs immediately after the last dependent instruction, ensuring they start execution as early as possible. This strategy may compromise the quality of selected CIs, particularly given that the DEG model evaluates whole basic blocks.

OOO2 GPP: CIExplorer achieves 12.1% cycle and 8.5% energy reduction, outperforming its non-DEG counterpart (10% cycle and 8.3% energy reduction). However, both methods exhibit performance degradation and reduced energy efficiency on the 456.hmmer whose CIs are extracted from function `P7Viterbi`, which has many short branches comparing array elements [11]. Our static DEG model falls short in evaluating dynamic branches and memory operations, so it may estimate potential benefits inaccurately. The non-DEG version increases energy consumption on 482.sphinx3 and degrades performance on 482.sphinx3, mpeg2dec, and djpeg. 482.sphinx3 (`vector_gautbl_eval_logs3`) contains reduction operations [63], mpeg2dec (`motion_compensation`) has various kernels of different characterization, such as element-wise sum and intensive branches, and djpeg has indirect memory access but high ILP [45] in `ycc_rgb_convert`. In contrast, the DEG version performs better on these benchmarks.

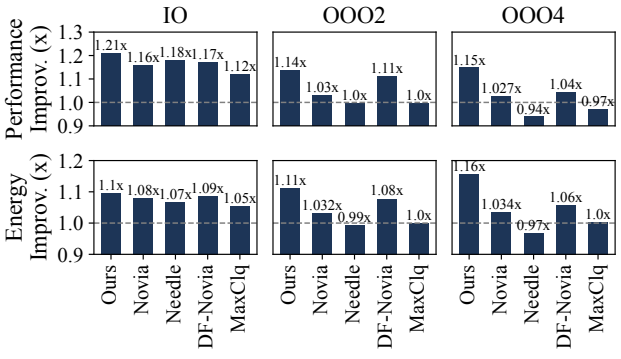


Figure 12: Improved performance and energy efficiency of different methods compared with baseline GPPs.

Table 4: Area (mm^2) of custom FUs' datapaths of different methods.

Method	IO	OOO2	OOO4
Ours	0.165	**0.089**	**0.054**
Novia	0.196	-	-
MaxClique	0.128	-	-
Needle	**0.124**	0.116	0.110

OOO4 GPP: CIExplorer delivers 12.9% cycle and 11.4% energy reduction, outperforming the non-DEG results (7.2% cycle and 7.4% energy reduction). While the non-DEG implementation suffers more performance losses on benchmarks, e.g. 456.hmmer (-20%), the DEG version demonstrates performance gains on all benchmarks, validating its effectiveness for out-of-order architectures. Compared with OOO2, the DEG version finds smaller subgraphs for 456.hmmer on OOO4, which may benefit short branches. An interesting example is 429.mcf, which achieves a high speedup. Its CIs are extracted from the function `price_out_impl`. Our method finds vector-like patterns containing two or three `getelementptr` instructions, which are modeled as standalone arithmetic operations in our simulator. These CIs accelerate address generation for the strided memory access pattern and then expose more memory-level parallelism exploited by high pipeline width.

One observation is that the non-DEG version degrades performance less severely than Novia [56] for both OOO2 and OOO4. GA enumerates and selects subgraphs simultaneously. Simply applying Equation (1) as a cost model is not enough to discover maximal convex subgraphs because pursuing maximal is not consistent with the target of maximizing speedup for a subgraph. As a result, it is unable to be as good as Novia towards IO GPPs. Similarly, our performance-oriented method performs well for OOO GPPs. The goal of exploring maximal subgraphs remains highly effective for IO GPPs. This is why we use an optimized SCM for IO GPPs.

9.2 Comparison with Existing Techniques

We compare CIExplorer with Novia [56] and MaxClique [57] on three GPPs, both identifying CIs without live-in/out constraint. We also compare CIExplorer with two other specialization techniques: (1) DF-Novia, which accelerates Novia on a dataflow architecture similar to Dyser [27], and (2) Needle [36] proposed for speculative offloading. We simulate DF-Novia and Needle similarly to CIs by including all operations in their datapaths. Figure 12 shows comparison results. We manually measure and select the path of the highest speedup from the top 10 most executed paths extracted by Needle for different cores. Novia and MaxClique can not adjust extracted regions for OOO microarchitectures, which achieves minor improvements and even decreases performance. Our method provides performance improvements of 1.09× and 1.13× on average across all cores compared with Novia and MaxClique, respectively, and improves energy by 1.07× and 1.10×. DF-Novia generally produces better results than basic Novia because it eliminates all explicit invoking (`EXE`) instructions. However, even with an advanced dataflow-based technique, Novia still cannot achieve better performance for OOO cores than ours. Needle accelerates hot paths containing several basic blocks, which are identified only according to executed counts of basic blocks. It also suffers from speculation failure overhead. Our method achieves 1.12× and 1.11× performance and energy improvements over it.

Table 4 compares the required average area of all benchmarks. Novia and MaxClique are unaware of microarchitectures, so they have the same area for all cores, while Needle's area varies because of our manual selection. For IO GPP, Novia and Needle require the most and the least area, respectively. CIExplorer requires less area than Novia but more than others due to our maximal graph extraction strategy. MaxClique employs a pruning strategy that removes subgraphs of low potential benefits and requires less area than ours. For OOO2 and OOO4, CIExplorer achieves the least area requirement.

9.3 Characterization for CIs

Figure 13 presents the breakdown of dynamic instructions and counts of static CIs for three GPP types. Percentages are calculated by normalizing values to baseline GPPs. Three bars are IO, OOO2, and OOO4 from left to right. Yellow nodes are static CI counts. Customization reduces dynamic instructions for most benchmarks, which improves performance by saving pipeline slots, e.g., ROB, issue, and commit width. `EXTRACT` instructions occupy a large percentage on IO GPP, while `PACK` instructions are fewer since one `PACK` instruction

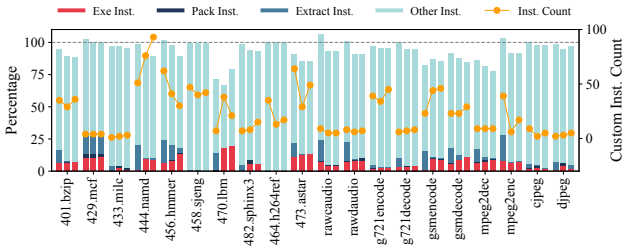


Figure 13: Three bars from left to right breakdown percentage of dynamic instructions for IO, OOO2, and OOO4 GPPs normalized to baseline. Yellow nodes show static CI counts.

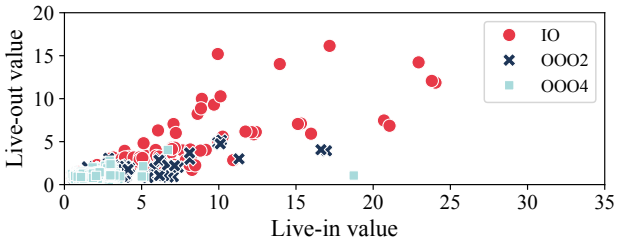


Figure 14: Distribution of live-in and live-out values.

can move three operands. The count of EXTRACT instructions decreases for OOO2 and OOO4 GPPs because CIs for OOO2 and OOO4 have fewer live-ins and live-outs. We show the distribution of live-ins and live-outs in Figure 14. Most CIs of OOO2 GPP are located within [0, 10] in the x-axis and [0, 5] in the y-axis, meaning the ranges of live-ins and live-outs, while most CIs of OOO4 GPP have fewer than 5 live-ins and live-outs. It shows how the microarchitecture-aware cost model affects the selected CIs. Figure 13 shows a trend that OOO4 has more static instructions than OOO2 (except for 456.hmmer and 470.lbm) but costs less area (see Table 4), indicating that more and smaller subgraphs are extracted from fewer code regions guided by our DEG model.

9.4 Design Space Exploration

Area Budget: Figure 15A shows geometric means of normalized cycles across benchmarks under different area budgets. Maximal areas (100%) are shown in Table 4. IO and OOO2 GPPs show gradually improved cycles with increasing area budgets. OOO4 achieves a good performance at a budget of 40%, showing a curve with small variations after it, and achieves the best at 100% area budget. The results for OOO4 indicate that CIs corresponding to 60% of the area provide only minor improvements to overall performance. Ideally, a higher area budget would allow the utilization of more CIs to accelerate computations, as the curves of IO and OOO2. Such a trend is mainly because 74.4% CIs have been covered under the 40% area budget on OOO4. Since we select subgraphs for a specific area budget by solving a knapsack problem, low-scoring subgraphs, which contribute slightly or negatively to overall performance, are involved in evaluations of more area budgets. On the one hand, even though such low-scoring subgraphs are discovered, our method still manages to prevent performance degradation and improve overall performance. On the other hand, subgraphs of higher scores contribute more significantly to performance, demonstrating our cost model's effectiveness.

Pack Operand: Since our simulator uses LLVM IR as the ISA, and due to LLVM IR's limitation that each instruction can only have a single return value, we only consider the operands of PACK. CIs are rediscovered separately under different operand constraints. Figure 15B shows results in normalized cycles. The IO GPP exhibits a clear trend: performance improves consistently as the number of operands increases, but only minor performance gains are observed for OOO2 and OOO4. Increasing the operand count of PACK allows subgraphs to have more live-in values, thereby expanding the subgraph size and bringing out more live-out values. As this paper discussed, larger graph sizes may not be beneficial, and our cost model suppresses this trend and identifies suitable subgraphs for OOO2 and OOO4 that are more sensitive to subgraph size.

Execution Port: Due to ILP, CIs may compete for the same execution port. We evaluated the impact of execution port count for CIs on the OOO4 GPP, assigning FUs to ports in an interleaving way. Figure 15C presents the results, where N denotes that each custom FU is connected to a dedicated port. The GPP achieves good performance when the number of ports is 2, further increasing the port count yields only minor performance gains.

10 Related Work

Fine-grained CI exploration is generally limited to the scope within basic blocks [31]. MaxClique [57] leads to the flourishing of methods of enumerating maximal convex subgraphs[6, 48]. [23] proposes to enumerate maximum convex subgraphs for specific live-in/out constraints, optimizing speedup instead of the number of nodes. To share hardware resources, [4, 9, 13] directly enumerate isomorphic graphs, while some other works introduce a separate merging procedure [5, 30, 43, 44]. Works [56, 64, 65] have improved this paradigm by introducing delay constraints in the merging process. For solving the selection problem, except optimal [7, 13, 17] and heuristic algorithms [41, 49, 59, 61], [58] applies a machine learning-based method to solve the selection problem. Works [21, 24] are proposed for VLIW processors. Some

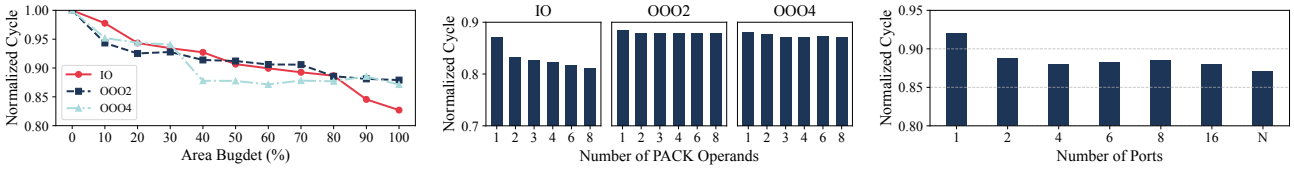


Figure 15: Design space exploration. (A) Normalized cycles under different area budgets. (B) Impact of the number of operands of PACK instruction. (C) Impact of the number of execution ports for CIs on OOO4 GPP.

works try to find and accelerate coarse-grained regions containing several basic blocks. [53] splits hot paths into chains to maximize parallelism and reduce data movement. Moreover, [62] and [10] attempt to extract one or more functions to offload as standalone accelerators. SIMD [33], GPU [47], and standalone accelerators [34] can accelerate regular code. A trend is to accelerate irregular codes in terms of control flow and memory access by specializing GPPs, such as Beret [28] and [19] supporting speculative execution and [27, 42, 45] exploring the potential of dataflow architectures.

11 Conclusion

This paper presents a framework named CIExplorer for CI exploration. Our framework addresses the problem of current works that cannot discover CIs adaptively to different microarchitectures by introducing a graph-based cost model. We also propose a compiler-assisted IR specialization method capable of automatically generating custom instructions, bridging the gap between subgraphs and complex ISA extensions. We improve the quality of discovered subgraphs by proposing the SGM based on GA to efficiently enumerate graphs of same order at different granularity. This method enables modeling interactions among CIs during the selection and helps reduce area overhead. Experimental results demonstrate the feasibility and generality of CIExplorer in terms of awareness of microarchitectures for finding CIs by testing various benchmarks on three different GPPs. Results also show that our method outperforms current approaches in terms of both performance and energy efficiency. In conclusion, our work shows achievable benefits by exploiting a microarchitecture-aware exploration for CIs.

Acknowledgments

We thank anonymous reviewers for their constructive comments. This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No.XDB0500102), Laoshan Laboratory (No.LSKJ202300305).

References

[1] 2018. SPEC CPU 2006. https://www.spec.org/cpu2006/. Accessed: 2024-12.

[2] 2020. Arm Custom Instructions: Enabling Innovation and Greater Flexibility on Arm. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-custom-instructions-wp.pdf.

[3] Andreas Abel and Jan Reineke. 2019. uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 673–686.

[4] Junwhan Ahn and Kiyoung Choi. 2012. Isomorphism-aware identification of custom instructions with I/O serialization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 1 (2012), 34–46.

[5] Kubilay Atasu, Günhan Dündar, and Can Özturan. 2005. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. 172–177.

[6] Kubilay Atasu, Wayne Luk, Oskar Mencer, Can Ozturan, and Günhan Dundar. 2010. FISH: Fast instruction synthesis for custom processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20, 1 (2010), 52–65.

[7] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. 2003. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th annual Design Automation Conference*. 256–261.

[8] Chen Bai, Jiayi Huang, Xuechao Wei, Yuzhe Ma, Sicheng Li, Hongzhong Zheng, Bei Yu, and Yuan Xie. 2023. ArchExplorer: Microarchitecture exploration via bottleneck analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 268–282.

[9] Paolo Bonzini and Laura Pozzi. 2008. Recurrence-aware instruction set selection for extensible embedded processors. *IEEE transactions on very large scale integration (VLSI) systems* 16, 10 (2008), 1259–1267.

[10] Iulian Brumar, Georgios Zacharopoulos, Yuan Yao, Saketh Rama, David Brooks, and Gu-Yeon Wei. 2023. Early dse and automatic generation of coarse-grained merged accelerators. *ACM Transactions on Embedded Computing Systems* 22, 2 (2023), 1–29.

[11] Christopher Celio, Palmer Dabbelt, David A Patterson, and Krste Asanović. 2016. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *arXiv preprint arXiv:1607.02318* (2016).

[12] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE,

272–283.

[13] Nathan Clark, Amir Hormati, Scott Mahlke, and Sami Yehia. 2006. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 147–157.

[14] Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *37th international symposium on microarchitecture (MICRO-37'04)*. IEEE, 30–40.

[15] Nathan Clark, Hongtao Zhong, and Scott Mahlke. 2003. Processor acceleration through automated instruction set customization. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. IEEE, 129–140.

[16] Nathan T Clark, Hongtao Zhong, and Scott A Mahlke. 2005. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.* 54, 10 (2005), 1258–1270.

[17] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. 2004. Application-specific instruction generation for configurable processor architectures. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. 183–189.

[18] Mihaela Damian, Julian Oppermann, Christoph Spang, and Andreas Koch. 2022. SCAIE-V: an open-source scalable interface for ISA extensions for RISC-V processors. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 169–174.

[19] Muhammad Umar Farooq, Lizy John, and Margarida F Jacome. 2009. Compiler controlled speculation for power aware ilp extraction in dataflow architectures. In *High Performance Embedded Architectures and Compilers: Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings 4*. Springer, 324–338.

[20] Brian Fields, Rastislav Bodik, and Mark D Hill. 2002. Slack: Maximizing performance under technological constraints. *ACM SIGARCH Computer Architecture News* 30, 2 (2002), 47–58.

[21] Joseph A Fisher, Paolo Faraboschi, and Giuseppe Desoli. 1996. Custom-fit processors: Letting applications define architectures. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE, 324–335.

[22] Carlo Galuzzi, Koen Bertels, and Stamatis Vassiliadis. 2007. A linear complexity algorithm for the generation of multiple input single output instructions of variable size. In *International Workshop on Embedded Computer Systems*. Springer, 283–293.

[23] Emanuele Giaquinta, Anadi Mishra, and Laura Pozzi. 2015. Maximum convex subgraphs under I/O constraint for automatic identification of custom instructions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 3 (2015), 483–494.

[24] Vikkitharan Gnanasambandapillai, Jorgen Peddersen, Roshan Ragel, and Sri Parameswaran. 2020. Finder: Find efficient parallel instructions for asips to improve performance of large applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3577–3588.

[25] Hossein Golestani, Rathijit Sen, Vinson Young, and Gagan Gupta. 2022. Calipers: a criticality-aware framework for modeling processor performance. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.

[26] Ricardo E Gonzalez. 2000. Xtensa: A configurable and extensible processor. *IEEE micro* 20, 2 (2000), 60–70.

[27] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[28] Shantanu Gupta, Shuguang Feng, Amin Ansari, Scott Mahlke, and David August. 2011. Bundled execution of recurring traces for energy-efficient general purpose processing. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 12–23.

[29] Juris Hartmanis. 1982. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review* 24, 1 (1982), 90.

[30] Hui Huang, Taemin Kim, and Yatin Hoskote. 2014. Edit distance based instruction merging technique to improve flexibility of custom instructions toward flexible accelerator design. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 219–224.

[31] Eslam Hussein, Bernd Waschneck, and Christian Mayr. 2024. Automating application-driven customization of ASIPs: A survey. *Journal of Systems Architecture* (2024), 103080.

[32] Paolo Ienne and Rainer Leupers. 2006. *Customizable embedded processors: design technologies and applications*. Elsevier.

[33] Intel Corporation. 2024. Intel®Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. Accessed: 2024-06-13.

[34] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[35] Hyungyo Kim, Gaohan Ye, Nachuan Wang, Amir Yazdanbakhsh, and Nam Sung Kim. 2024. Exploiting Intel® Advanced Matrix Extensions (AMX) for Large Language Model Inference. *IEEE Computer Architecture Letters* (2024).

[36] Snehasish Kumar, Nick Sumner, Vijayalakshmi Srinivasan, Steve Margerm, and Arrvindh Shriraman. 2017. Needle: Leveraging program analysis to analyze and extract accelerators from whole programs. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 565–576.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[38] Jan Laukemann, Julian Hammer, Georg Hager, and Gerhard Wellein. 2019. Automatic throughput and critical path analysis of x86 and arm assembly kernels. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 1–6.

[39] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 330–335.

[40] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*. 469–480.

[41] Tao Li, Wu Jigang, Siew-Kei Lam, Thambipillai Srikanthan, and Xicheng Lu. 2010. Selecting profitable custom instructions for reconfigurable processors. *Journal of Systems Architecture* 56, 8 (2010), 340–351.

[42] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 163–174.

[43] Nahri Moreano, Guido Araujo, Zhining Huang, and Sharad Malik. 2002. Datapath merging and interconnection sharing for reconfigurable architectures. In *Proceedings of the 15th international symposium on*

System Synthesis. 38–43.

[44] Nahri Moreano, Edson Borin, Cid De Souza, and Guido Araujo. 2005. Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 7 (2005), 969–980.

[45] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 298–310.

[46] Tony Nowatzki, Venkatraman Govindaraju, and Karthikeyan Sankaralingam. 2015. A graph-based program representation for analyzing hardware specialization approaches. *IEEE Computer Architecture Letters* 14, 2 (2015), 94–98.

[47] NVIDIA Corporation. 2017. NVIDIA Tesla V100 GPU Architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[48] Nagaraju Pothineni, Anshul Kumar, and Kolin Paul. 2007. Application specific datapath extension with distributed i/o functional units. In *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. IEEE, 551–558.

[49] Laura Pozzi, Kubilay Atasu, and Paolo Ienne. 2006. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1209–1229.

[50] Laura Pozzi and Paolo Ienne. 2005. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. 2–10.

[51] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function merging by sequence alignment. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 149–163.

[52] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 97–108.

[53] Amirali Sharifian, Snehasish Kumar, Apala Guha, and Arrvindh Shriraman. 2016. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.

[54] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGPLAN Notices* 37, 10 (2002), 45–57.

[55] Shaojie Tan, Qingcai Jiang, Zhenwei Cao, Xiaoyu Hao, Junshi Chen, and Hong An. 2024. Uncovering the performance bottleneck of modern HPC processor with static code analyzer: a case study on Kunpeng 920. *CCF Transactions on High Performance Computing* 6, 3 (2024), 343–364.

[56] David Trilla, John-David Wellman, Alper Buyuktosunoglu, and Pradip Bose. 2021. Novia: A framework for discovering non-conventional inline accelerators. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 507–521.

[57] Ajay K Verma, Philip Brisk, and Paolo Ienne. 2007. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. 125–134.

[58] Shanshan Wang and Chenglong Xiao. 2023. Reinforcement Learning for Selecting Custom Instructions under Area Constraint. *IEEE Transactions on Artificial Intelligence* (2023).

[59] Shanshan Wang, Chenglong Xiao, Wanjun Liu, and Emmanuel Casseau. 2016. A comparison of heuristic algorithms for custom instruction selection. *Microprocessors and Microsystems* 45 (2016), 176–186.

[60] Chenglong Xiao and Emmanuel Casseau. 2012. Exact custom instruction enumeration for extensible processors. *Integration* 45, 3 (2012), 263–270.

[61] Chenglong Xiao, Emmanuel Casseau, Shanshan Wang, and Wanjun Liu. 2014. Automatic custom instruction identification for application-specific instruction set processors. *Microprocessors and Microsystems* 38, 8 (2014), 1012–1024.

[62] Georgios Zacharopoulos, Lorenzo Ferretti, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. 2019. Compiler-assisted selection of hardware acceleration candidates from application source code. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 129–137.

[63] Bo Zhao, Wei Gao, Rongcai Zhao, Lin Han, Huihui Sun, and Yingying Li. 2015. Performance evaluation of NPB and SPEC CPU2006 on various SIMD extensions. In *Big Data Computing and Communications: First International Conference, BigCom 2015, Taiyuan, China, August 1-3, 2015, Proceedings 1*. Springer, 257–272.

[64] Marcela Zuluaga and Nigel Topham. 2008. Resource sharing in custom instruction set extensions. In *2008 Symposium on Application Specific Processors*. IEEE, 7–13.

[65] Marcela Zuluaga and Nigel Topham. 2009. Design-space exploration of resource-sharing solutions for custom instruction set extensions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1788–1801.