JCST Journal of Computer Science & Technology

Indexed in: SCIE, EI, INSPEC, CBST, DBLP, etc.

Sponsored by: ICT, CAS & CCF

Automatic Target Description File Generation

Geng Hong-Na, Lyu Fang, Zhong Ming, Cui Hui-Min, Xue Jingling, Feng Xiao-Bing

View online: http://doi.org/10.1007/s11390-022-1919-x

JCST Official WeChat Account

JCST WeChat Service Account

JCST Homepage: https://jcst.ict.ac.cn
SPRINGER Homepage: https://www.springer.com/journal/11390
E-mail: jcst@ict.ac.cn
Online Submission: https://mc03.manuscriptcentral.com/jcst
Twitter: JCST_Journal
LinkedIn: Journal of Computer Science and Technology

Geng HN, Lyu F, Zhong M *et al.* Automatic target description file generation. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(6): 1339-1355 Nov. 2023. DOI: 10.1007/s11390-022-1919-x

Automatic Target Description File Generation

Hong-Na Geng[1, 2] (耿洪娜), *Student Member, CCF*, Fang Lyu[1, *] (吕　方), *Member, CCF*
Ming Zhong[1, 2] (钟　茗), *Student Member, CCF*, Hui-Min Cui[1, 2] (崔慧敏), *Member, CCF*
Jingling Xue[3], *Senior Member, IEEE*, and Xiao-Bing Feng[1, 2] (冯晓兵), *Senior Member, CCF*

[1] *State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190 China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia*

E-mail: genghongna@ict.ac.cn; flv@ict.ac.cn; zhongming21s@ict.ac.cn; cuihm@ict.ac.cn; jingling@cse.unsw.edu.au fxb@ict.ac.cn

Received September 17, 2021; accepted March 6, 2022.

Abstract Agile hardware design is gaining increasing momentum and bringing new chips in larger quantities to the market faster. However, it also takes new challenges for compiler developers to retarget existing compilers to these new chips in shorter time than ever before. Currently, retargeting a compiler backend, e.g., an LLVM backend to a new target, requires compiler developers to write manually a set of target description files (totalling 10 300+ lines of code (LOC) for RISC-V in LLVM), which is error-prone and time-consuming. In this paper, we introduce a new approach, Automatic Target Description File Generation (ATG), which accelerates the generation of a compiler backend for a new target by generating its target description files automatically. Given a new target, ATG proceeds in two stages. First, ATG synthesizes a small list of target-specific properties and a list of code-layout templates from the target description files of a set of existing targets with similar instruction set architectures (ISAs). Second, ATG requests compiler developers to fill in the information for each instruction in the new target in tabular form according to the list of target-specific properties synthesized and then generates its target description files automatically according to the list of code-layout templates synthesized. The first stage can often be reused by different new targets sharing similar ISAs. We evaluate ATG using nine RISC-V instruction sets drawn from a total of 1 029 instructions in LLVM 12.0. ATG enables compiler developers to generate compiler backends for these ISAs that emit the same assembly code as the existing compiler backends for RISC-V but with significantly less development effort (by specifying each instruction in terms of up to 61 target-specific properties only).

Keywords retargetability, compiler, target description, target backend, automatic generator

1 Introduction

Agile hardware design promises to bring a variety of new chips to the market faster[1–3], posing new challenges for compiler developers who are required to retarget existing compilers to these new chips in shorter time than ever before. Existing compilers leverage target-independent code generation to help construct a compiler backend for a new target architecture by requiring compiler developers to write a set of description files to specify its similar instruction set architecture (ISA) among others, e.g., target description (*.td) files in TableGen in LLVM or machine description (*.md) files in GCC. However, writing such description files can be error-prone and time-consuming[4]. Fig.1 highlights the efforts required in developing the compiler backends for 20 different architectures in LLVM 12.0.

Regular Paper

The work was supported by the Strategic Pilot Science and Technology Project of Chinese Academy of Sciences (Category C) under Grant No. XDC05000000, and the Youth Program of National Natural Science Foundation of China under Grant No. 61802368.

*Corresponding Author

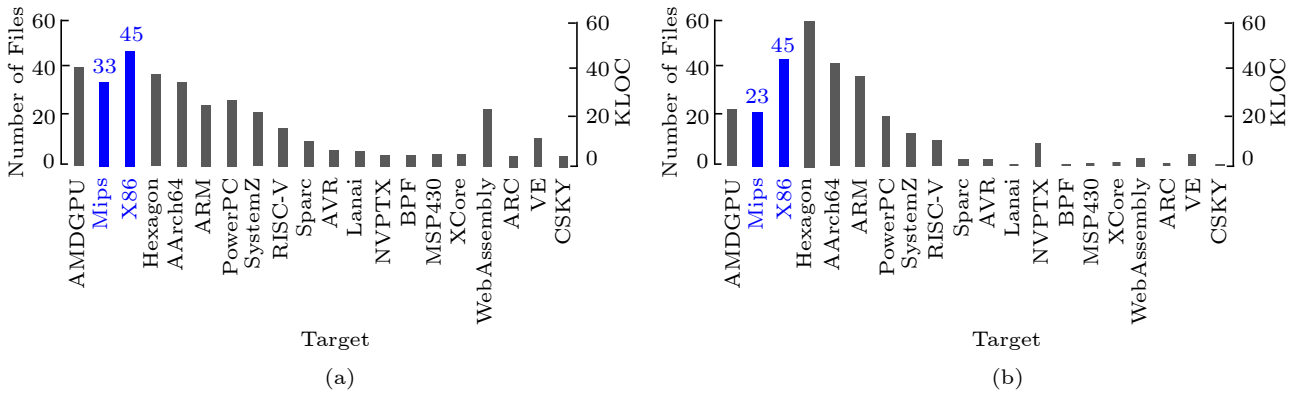

Fig.1. Heavy research and development (R&D) efforts in writing target description (.td) files for a new target in LLVM 12.0.

For MIPS (with 1 700+ instructions), LLVM requires 33 *.td files totalling about 23 KLOC. For x86 (with 15 000+ instructions), LLVM requires 45 files totalling about 45 KLOC.

In this paper, we are therefore motivated to develop a new approach, ATG (Automatic Target Description File Generation), which can generate a compiler backend for a new target more quickly by requiring compiler developers to fill in the information for each instruction in tabular form according to a small list of target-specific properties pre-synthesized and then generating the *.td files for the new target automatically according to a list of code layout templates pre-synthesized.

Our key observation behind the development of ATG is that while the .td files written for a new target can be tens of KLOC long, all the target-dependent tokens (used for describing target-specific properties) can be relatively small. For each of the two TableGen records given in Fig.2, only two tokens (highlighted in blue) are target-dependent and all the rest are target-independent. If ATG first asks compiler developers to fill in the information for such target-dependent tokens and then completes the rest automatically by itself, should not we enhance compiler retargetability quite substantially? If so, how do we determine the target-specific properties for a new target and how do we generate its .td files automatically (based on the information provided for these target-specific properties)?

ATG addresses these challenges by proceeding in two stages for a new target. In the first stage, ATG synthesizes a small list of target-specific properties (referred to as TSP-List), together with a list of code-layout templates (referred to as CLT-List), from the existing target description files of a set of targets with similar ISAs to the new target. TSP-List captures all aspects of the new target that must be known to the compiler, including instruction formats, registers, pipelines and calling conventions[4], such as "`jr16`" for an instruction name and "`0x0c`" for an opcode for MIPS as shown in Fig.2. In the second stage, ATG first asks compiler developers to fill in the information for each instruction in the new target in tabular form according to TSP-List and then generates automatically its target description files according to CLT-List. In practice, the first stage can often be reused by different targets sharing similar ISAs, so that the time for synthesizing TSP-List and CLT-List can be amortized.

So far, we have not been aware of any prior work on accelerating building compiler backends for new targets by generating their target description files automatically in this way. In summary, we make the following contributions.

• We present a new approach, ATG, to significantly accelerate building a compiler backend for a new target by requiring compiler developers to de-

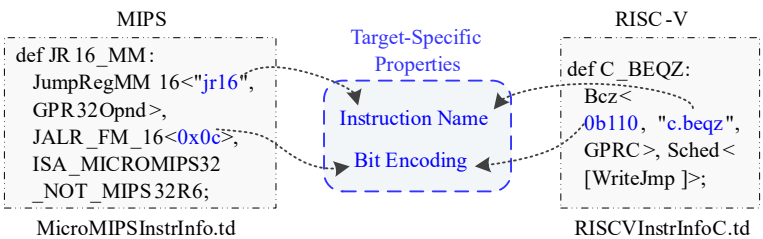

Fig.2. Reduced R&D efforts in specifying a few target-specific properties.

scribe it in terms of a few dozens of target-specific properties (in TSP-List) and then generating its target description files automatically according to a list of code layout templates (in CLT-List).

• We introduce a token-based classification approach to synthesize both TSP-List and CLT-List for a given target from the target description files of a set of existing targets sharing similar ISAs as the new target.

• We have implemented ATG in LLVM 12.0 and have evaluated it by using nine RISC-V ISAs comprising a total of 1 029 instructions (with 404 scalar instructions and 625 vector instructions from the RISC-V Vector extension (RVV)[①]). An experienced compiler developer in our group usually spends months on writing the *.td files in TableGen for each of these nine ISAs. In contrast, ATG can reduce these efforts quite substantially. ATG synthesizes TSP-List and CLT-List (only once) from the target description files of 10 MIPS ISAs in about four days. For the nine RISC-V ISAs (with 13–625 instructions), it takes only 1 day–7 days for the same compiler developer to specify each target according to TSP-List. Then ATG generates automatically the *.td files for the nine ISAs in about two minutes per target according to CLT-List. Although these ATG-generated *.td files are larger than the hand-written ones in LLVM by 2x–3x per target overall, the two compiler backends generated by LLVM (in more or less the same amount of time with nearly the same code size) from both sets of *.td files emit the same assembly code for the 16 C/C++ benchmarks of SPEC2017 and more than 15 600 LLVM regression test cases while incurring more or less the same amount of compilation time.

The rest of this paper is organized as follows. Section 2 motivates our ATG approach with a simple example. Section 3 describes our ATG approach. Section 4 discusses its implementation. Section 5 evaluates its effectiveness. Section 6 discusses related work and Section 7 concludes the paper.

2 Motivation

Fig.3 shows how ATG is designed to improve prior work on building a compiler backend for a new target. Traditionally, as illustrated in Fig.3(a), a target-independent generator relies on a set of handle-written *.td files (totalling often tens of KLOC) to produce a compiler backend for the new target. In this work, as illustrated in Fig.3(b), ATG requires compiler developers to fill in only the information for each instruction requested in TSP-List (a pre-synthesized list with dozens of target-specific properties) and then generates the *.td files for the new target automatically according to CLT-List (a pre-synthesized list of code-layout templates). Let us consider the case of writing a compiler backend for MIPS (with more than 1 700 instructions) using TableGen in LLVM. Manu-

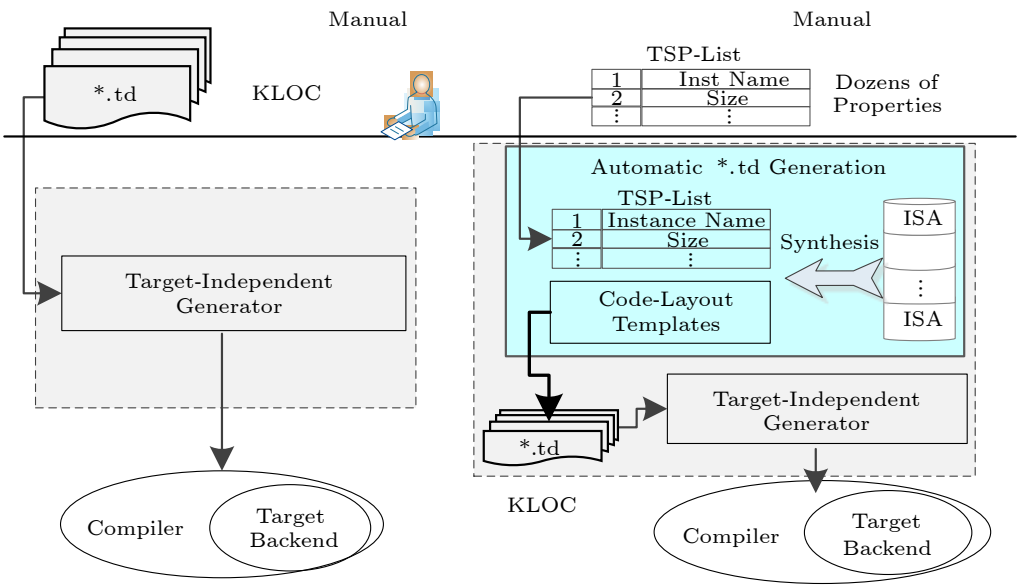

Fig.3. Building compiler backends by producing the *.td files for a new target manually as in (a) prior work and automatically as in (b) this work.

[①]https://github.com/riscv/riscv-v-spec, Sept. 2021.

ally, one would end up with writing 24 *.td files in about 19 KLOC for instruction support. With ATG, one would only need to fill in the information for each instruction according to 19–27 target-specific properties, i.e., tokens (Fig.2), where all the target-independent tokens will be generated automatically.

Given a new target, ATG proceeds in two stages. In the first stage (which can be reused for different new targets sharing similar ISAs), ATG synthesizes TSP-List and CLT-List from the *.td files of a set of existing targets with similar ISAs. Each template in CLT-List is identified uniquely by a feature vector characterized by its target-specific tokens included. In the second stage, ATG generates the *.td files for the new target by using the target-specific information requested in TSP-List and provided by compiler developers for each instruction (so that each instruction is also identified by a feature vector) to construct a description for the instruction in TableGen based on similarity-based matching (between the instruction and one of the code-layout templates in CLT-List).

In Subsection 2.1, we motivate our ATG approach by considering a single instruction example. In Subsection 2.2, we provide some further justifications for its practical feasibility.

2.1 A Motivating Example

Fig.4 illustrates how ATG generates a description of `c.jr` from RISC-V in TableGen based on some target-specific information provided by compiler developers. Fig.4(a) depicts its bit encoding, where its opcode "`0b10`" ranges in bits "`0`"–"`1`" for a "`16`"-bit instruction. For simplicity, we focus only on four of its target-specific properties.

Fig.4(b) gives a hand-written description taken from LLVM 12.0, coded declaratively as a sequence of TableGen records, where a `class` statement introduces an abstract record and a `def` statement introduces a concrete record. The four target-specific prop-

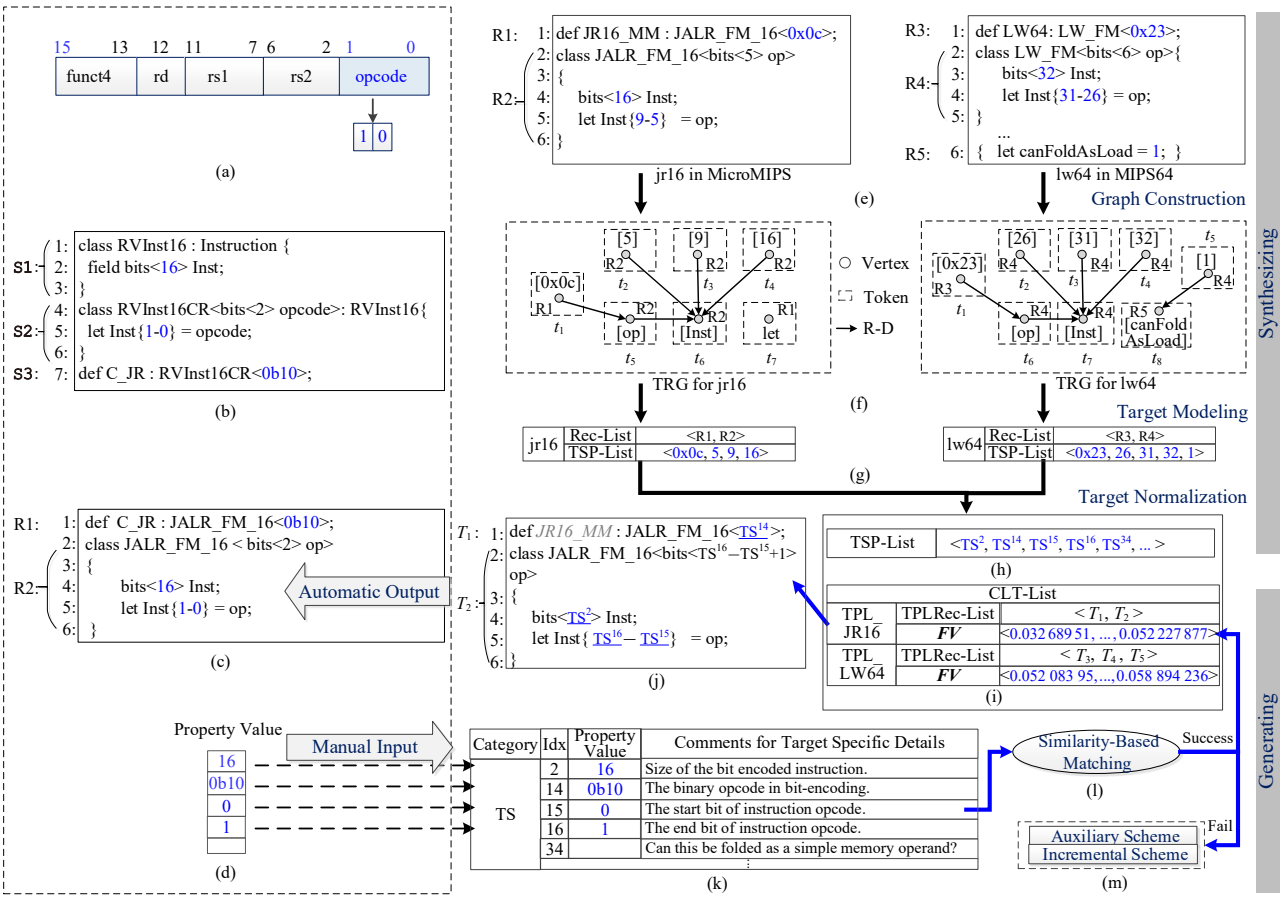

Fig.4. Motivating example for illustrating ATG. (a) Bit encoding of c.jr from RISC-V. (b) Hand-written description of c.jr taken from LLVM. (c) Description of c.jr generated by ATG automatically. (d) The only effort from compiler developers for c.jr. (e) Example records (MicroMIPS and MIPS64). (f) TRGs (MicroMIPS and MIPS64). (g) ISA models (MicroMIPS and MIPS64). (h) Target-specific property list. (i) Synthesized code-layout template list. (j) Synthesized TPLRec-List for c.jr. (k) Target-specific property list (TSP-List). (l) Similarity-based matching. (m) Schemes for failed matching. R*i*: record *i*. Idx: index.

erties considered here are specified by three records, S1–S3, where S1 defines the "16"-bit mode at line 2, S2 defines the bit range "0"–"1" for the opcode at line 5, and S3 specifies the opcode as "0b10".

Fig.4(c) gives a description of c.jr generated by ATG automatically according to the pre-synthesized code layout templates in CLT-List. This description is functionally equivalent to the hand-written one in Fig.4(b) but with a different code layout. As far as compilers developers are concerned, they need only to fill in the information related to the four target-specific properties, i.e., tokens (as requested in TSP-List) for c.jr, as shown in Fig.4(d).

2.1.1 *Stage 1: Synthesizing TSP-List and CLT-List*

We obtain these two lists automatically for a new target by considering the *.td files from a set of existing targets with similar ISAs. For each existing target, we construct one single token reference graph (TRG) to capture a new kind of ref-def (R-D) relationship among the tokens that appear in all its *.td files. We distinguish three types of tokens (Table 1): 1) programmer-defined (PD) tokens (such as variables introduced by compiler developers), 2) language-specific (LS) tokens (such as operators and punctuation marks), and 3) target-specific (TS) tokens (such as instruction names and opcodes). We rely on the TRG for a target to separate type 3 from type 1 and type 2 in its *.td files. Fig.4(e) gives two records describing jr16 for MicroMIPS and three records describing lw64 for MIPS64. Fig.4(f) depicts a small portion of the TRG for MicroMIPS by including only a few tokens from Fig.4(e) for jr16 and a small portion of the TRG for MIPS64 by including only a few tokens from Fig.4(e) for lw64. Given two tokens t_1 and t_2 in a TRG, $t_1 \rightarrow t_2$ represents a ref-def relation such that t_1 is used in the statement where t_2 is defined. Let us consider Figs.4(e) and 4(f) by focusing on MicroMIPS only. From lines 4 and 5 in Figs.4(e), we have 0x0c $\rightarrow$ op, 5 $\rightarrow$ inst, 9 $\rightarrow$ inst, op $\rightarrow$ inst, and 16 $\rightarrow$ inst (since 16 as part of the type bits provided in line 4 is also considered to be used in line 5 where inst is defined). Given a $t_1 \rightarrow t_2$ edge in a TRG, t_1 cannot be an LS token and t_2 must be a PD token. Based on how the three different types of tokens are used, we can thus identify the four (five) tokens highlighted in blue as TS tokens for MicroMIPS (MIPS64) in Fig.4(e).

As illustrated in Fig.4(g), we now represent each instruction $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all instructions considered, as a 2-tuple $(Rec\text{-}List_i, TSP\text{-}List_i)$, where $Rec\text{-}List_i$ is a list of its associated records, and $TSP\text{-}List_i$ is a list of TS tokens identified in these records. This is illustrated for jr16 from MicroMIPS and lw64 from MIPS64.

Finally, as illustrated in Figs.4(h)–4(j), we obtain TSP-List and CLT-List by performing a simple target normalization process. For each instruction $i \in \mathcal{I}$, we have already obtained $(Rec\text{-}List_i, TSP\text{-}List_i)$ as illustrated in Fig.4(g). To obtain TSP-List (Fig.4(h)), we simply collect and combine the target-specific tokens in TRGs. To obtain CLT-List (Fig.4(i)), we consider every instruction $i \in \mathcal{I}$ in turn. We include a code-layout template $(TPLRec\text{-}List_i, \boldsymbol{FV}_i)$ in CLT-List, where $TPLRec\text{-}List_i$ is templatized from $Rec\text{-}List_i$ with normalization only on TS tokens, and $\boldsymbol{FV}_i$ is the feature vector obtained from $TSP\text{-}List_i$ for representing the instruction, such that $Rec\text{-}List_j$ is templatized identically as $TPLRec\text{-}List_i$, as illustrated in Figs.4(i) and 4(j). In each templatized record T_i (Fig.4(j)), TSP^i is the i-th entry in TSP-List (Fig.4(k)). For the two instructions considered, jr16 and lw64, two code layout templates, TPL_JR16 and TPL_LW64, are obtained (where T_3–T_5 are omitted).

Table 1. Three Categories of Tokens[②]

Token Category	Abbr.	Description
Language-specific	LS	Language-specific tokens such as reserved key words, operators, and punctuation, which are target-independent. This type is represented with LS. For example, 1) keywords: def, class, let; 2) bang operators: !eq, !add; 3) punctuation: ":", ",".
Programmer-defined	PD	Target-independent tokens defined by programmers. This type is represented with PD. For example, "inst", "op", ...
Target-specific	TS	Property tokens for machine-specific constraints, which are indispensable as to the correct target support in a compiler. This type is represented with TS. For example, "16" represents size of the bit encoded instruction. "0" represents the start bit of the instruction opcode and "1" represents the end bit.

[②]https://github.com/agilecompiler/agilecompiler, Jan. 2022.

2.1.2 Stage 2: Generating *.td Files

Given the target-specific information provided for `c.jr` from RISC-V in Fig.4(d), ATG will first turn this information into a feature vector. Afterwards, ATG will select TPL_JR16, based on similarity-based matching (Fig.4(l)), to generate a description for this instruction as shown in Fig.4(c). To completely specify this instruction, in practice, compiler developers need to fill in the information for 25 properties. In contrast, the hand-written description in LLVM consists of 46 LOC (with 336 tokens).

As demonstrated in Figs.4(l) and 4(m), in the rare cases when ATG fails to find a suitable code layout template to generate a description for a given instruction (e.g., a customized instruction), ATG provides two schemes to produce a description. The auxiliary scheme is proposed as a supplement for those failed instructions without new properties. For those failed instructions due to additional properties or due to more complex reference patterns on target-specific properties which are never captured in available ISAs, ATG should encourage developers to design a short piece of code manually which is referred to as the incremental scheme. These two schemes are presented with more details in Subsection 3.2.

2.2 Discussions

Let us discuss the practical feasibility of our ATG approach from four perspectives. First, when writing the *.td files manually for a new target, compiler developers often reuse code from existing targets[5]. This suggests that the code-layout templates in CLT-List can be automatically synthesized from the *.td files of a set of existing targets sharing similar ISAs as the new target. Second, the TS tokens are used differently from the LS and PD tokens in the *.td files. This suggests that the target-specific properties in TSP-List can also be automatically synthesized from the existing .td files in a similar way as CLT-List. Third, in the rare cases when ATG fails to generate a description for a certain instruction (e.g., a customized instruction in a domain-specific chip) in TableGen automatically, ATG provides means to call upon compiler developers to generate a description together. Finally, we can now obtain new compiler backends from ATG-generated *.td files instead of hand-written *.td files. In both cases, the correctness of a compiler backend can be verified in exactly the same way[6], often subject to comprehensive regression testing.

3 Automatic Target Description File Generation

We propose ATG to apply automatic generation for target description files. Fig.5 displays the two stages in ATG: 1) synthetic stage, analytically synthesizing TSP-List and CLT-List from similar ISAs, 2) generation stage, generating target description files for a new ISA with only a tabular form in terms of TSP-List.

In this section, we introduce the basic idea behind ATG. The synthetic stage includes in two steps, TRG construction for token analyses (Subsection 3.1.1) and target synthesizing (Subsections 3.1.2 and 3.1.3) for TSP-List and CLT-List. The generation stage for new targets with these two lists is then introduced (Subsection 3.2).

3.1 Synthetic Stage

The synthetic stage is to construct two reusable lists, TSP-List and CLT-List, from a set of existing targets with similar ISAs. TSP-List is a target-specific property list for generating a new instruction set which is required with specific values from developers. CLT-List manages a set of code-layout templates for automatic code generation in demand. These lists are refined from token-based analyses on a set of TRG graphs.

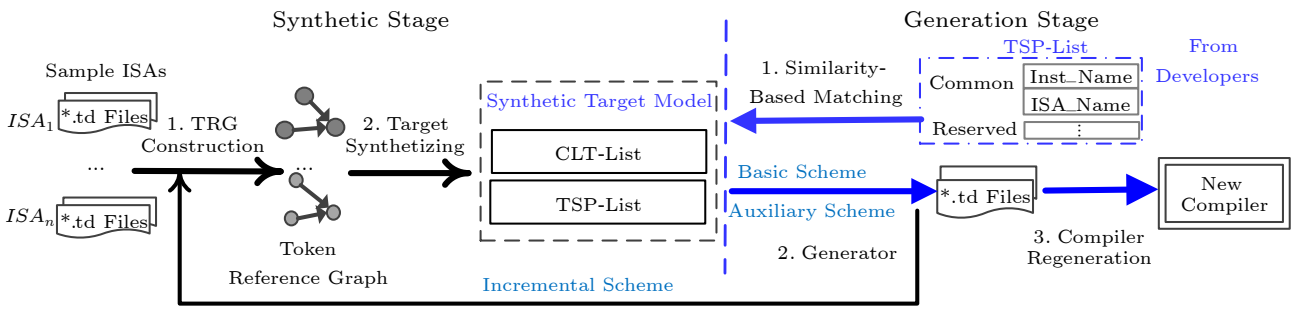

Fig.5. Framework of Automatic Target Description File Generation (ATG).

3.1.1 TRG Construction

A TRG is constructed on tokens from an ISA, which are parsed from records in *.td files. As lines 3–5 of Algorithm 1 show that the target-specific graph captures a new ref-def (R-D) relationship automatically among tokens. Basically, the graph relies on traditional def-use (D-U) techniques[7], but with a broader focus on usage, which is referred to as reference (ref). According to the definitions in Table 1, LS tokens and TS tokens are taken as references in our work. For each pair of tokens t_i and t_j in TRG, an R-D edge $t_i \rightarrow t_j$ is set up once t_i is used by t_j. The rules based on the R-D relation are summarized in lines 8–15 of Algorithm 1, which can make a distinction among PD, LS and TS tokens according to their distinct in-degree (IN-D) and out-degree (OUT-D) R-D edges. These rules illuminate the categories of t_1 and t_5 in Fig.4(f) for `jr16` where t_1 must be a TS token due to the only outgoing $t_1 \rightarrow t_5$ edge, and t_5 must be a PD token due to the in-coming edge. The token t_7 in the same TRG must be an LS token since it does not have any R-D chains. These three categories of tokens make it ready for later synthesizing process.

Algorithm 1. TRG-Construction

Input: $*.tdFiles$
Output: $TRGs$: token reference graphs;
$Category$: three categories of all tokens;

```
1  Records ← PreProcess(*.tdFiles);
2  for R^k, R^t ∈ Records do
3      for t^i ∈ R^k, t^j ∈ R^t do
4          if HasR-DRelation(t^i, t^j) then
5              TRGs.Add_Edge(t^i, t^j, R → D);
6  for R^j ∈ Records do
7      for t^i ∈ R^j do
8          IN-D ← CalculateINDegree(t^i, TRGs);
9          OUT-D ← CalculateOUTDegree(t^i, TRGs);
10         if IN-D == 0 && OUT-D == 0 then
11             Category<LS>.Append(t^i, LS);
12         if IN-D == 0 && OUT-D ≠ 0 then
13             Category<TS>.Append(t^i, TS);
14         if IN-D ≠ 0 then
15             Category<PD>.Append(t^i, PD);
16 return TRGs, Category;
```

3.1.2 Synthesizing TSP-List

TSP-List is a list of target-specific properties required for a target support. TSP-List is derived from the R-D relationship on TS tokens as described in Algorithm 2. A TS token in TRG is only a value for a specific target-specific property, e.g., `0x0c` for MicroMIPS `jr16` is a value specified for the property of `opcode` in Fig.4(f). The exact semantics for this value should be worked out via the R-D relation. As for a TS token t_i in a TRG, t_j from the only $t_i \rightarrow t_j$ edge is the PD token (a variable introduced by compiler developers) using it in the definition, and broadcasting it to the compiler backend as a carrier which is refereed as *PR-PD*. Thus, we append a new entry for *PR-PD* which corresponds to a target-specific property without repetition. During the R-D analyses, different *PR-PD* can contribute to new entries in TSP-List. In this way, the completed TSP-List should be a super-set on target-specific properties of all ISAs.

Algorithm 2. Synthesizing TSP-List

Input: $ISAs$: all selected sample ISAs;
Output: $TSP\text{-}List$

```
1  idx ← 0;
2  PR_PDs, TSP-List ← ∅;
3  for ISA ∈ ISAs do
4      *.tdFiles ← PrepareInput(ISA);
5      TRGs, Category ← TRG-Construction(*.tdFiles);
6      TS_Tokens ← CollectISATS(TRGs, Category,
       ISA.inst_set);
7      for t^i ∈ TS_Tokens do
8          PR_PD ← GetPDByR-DAnalyse(TRG, t^i);
9          comments_i ← ExtractComment(PR_PD);
10         value_i ← null;
11         if PR_PD ∉ PR_PDs then
12             PR_PDs ← PR_PDs ∪ PR_PD;
13             TSP-List.Add((idx, value_i, comments_i));
14             idx++;
15 return TSP-List;
```

As the motivation example in Fig.4(f) shows, as for `0x0c` → `op` for MicroMIPS jr16 and `0x23` → `op` for MIPS64 lw64, the common *PR-PD* of `op` contributes to the 14th entry in TSP-List as shown in Figs.4(h) and 4(k). Each entry in TSP-List is represented with a triple $TSP^i(i, value_i, comments_i)$, which refers to a target-specific property indexed by i. As to the 14th entry referring to `op`, this PR-PD token is extracted as the comment in $comments_i$, which can be manually maintained with "`The binary opcode in bit-encoding`" for much clearer indication in practice. The value field asks for an explicit value for this specific property from an instruction. When a new ISA is required, TSP-List needs to be specified with a list of new values by compiler developers according to detailed comments as the list of new values demonstrated in Fig.4(d).

Fig.6 demonstrates the TSP-List synthesized from 11 MIPS ISAs in LLVM 12.0. Due to space limita-

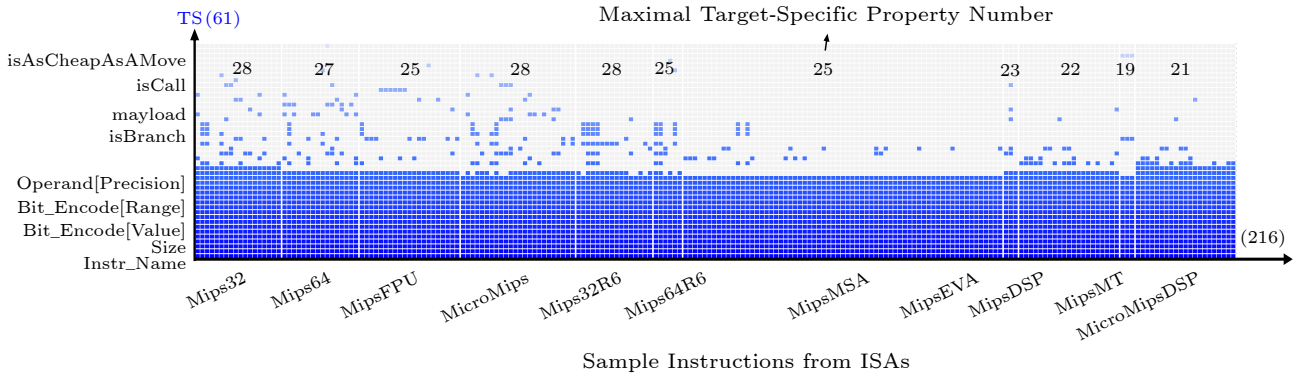

Fig.6. Target-specific properties distribution among different ISAs.

tion, only partial tokens are shown in the figure. The X-axis lists the 11 MIPS ISAs used in synthesizing TSP-List. For clarity, *PR-PD* tokens corresponding to TS tokens are demonstrated in Y-axis. These statistics on TS tokens certify that TRG is competent to make a distinction on target-specific properties. As for each of the ISAs in the X-axis, the properties in real need range in 17–28. However, there are some personalized or unique TS tokens for instructions of one ISA, or between different ISAs at the upper of the Y-axis. There are significant amount of TS tokens in common among ISAs at the lower end of the Y-axis. Therefore, a super set on all these properties is necessary to comprehensively cover these differences which can ensure both the correctness and the wider applicability of ATG. In our work, TSP-List with maximum 61 properties is competent enough for wider usage. Compared with the extensive code in target descriptions, it is trivial.

3.1.3 Synthesizing CLT-List

CLT-List manages a list of instruction models which are used for automatic code generation when a new instruction is required. As for each instruction $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all instructions in an ISA, a 2-tuple $(Rec\text{-}List_i, TSP\text{-}List_i)$ is used to represent the instruction, while $Rec\text{-}List_i$ collects associate records of i, and $TSP\text{-}List_i$ gathers those values for target-specific properties from associate records. CLT-List then uses a 2-tuple $(TPLRec\text{-}List_i, \boldsymbol{FV}_i)$ to represent the templatized instruction. $TPLRec\text{-}List_i$ collects code-layout templates for associated records in $Rec\text{-}List_i$ of i and $\boldsymbol{FV}_i$ is a unique real-vector representing i which can be synthesized from $TSP\text{-}List_i$.

Algorithm 3 demonstrates the collection on $Rec\text{-}List_i$ and $TSP\text{-}List_i$, which considers each instruction in turn. The 2-tuple representation for an instruction is acquired via TRG traversal. For instruction i, the algorithm starts from the property token which refers to the instruction name. The record using it is kept as the first associate record in $Rec\text{-}List_i$. As to each token of the record, tokens paired with it via R-D edges are figured out and records hosting these paired tokens can be added recursively to $Rec\text{-}List_i$. Then, all TS tokens in $Rec\text{-}List_i$ are collected into $TSP\text{-}List_i$. By repeating the above steps, the list of $Rec\text{-}List_i$ and $TSP\text{-}List_i$ can be perfected gradually.

Algorithm 3. Synthesizing CLT-List

Input: $TRGs$: token reference graphs;
$Category$: three categories of all tokens;
$\mathcal{I}$: inst_set of the sample ISA;
$ideal\_pw$: the result of tuning on pw;
Output: $CLT\text{-}List$

```
for i ∈ I do
    Rec-List_i ← GetRootRecord(i, TRGs);
    RecursiveAddByR-DEdge(Rec-List_i, TRGs);
    TSP-List_i ← CollectTS(Rec-List_i, Category);
    for R_j ∈ Rec-List_i do
        TPL-R_j ← Synthesizing(R_j, Category<TS>);
        TPLRec-List_i ← TPLRec-List_i ∪ TPL-R_j;
    FV_i ← Synthesize_FV(TSP-List_i, ideal_pw);
    add i(TPLRec-List_i, FV_i) into CLT-List;
return CLT-List;
```

After obtaining $(Rec\text{-}List_i, TSP\text{-}List_i)$, $TPLRec\text{-}List_i$ is templatized by substituting each of TS token t_i with TSP^k, while TSP^k is the corresponding entry to the *PR-PD* of t_i. As the example shown in Fig.4(i), the template can produce code for a new instruction which is similar to i in the later target generation stage.

$$\boldsymbol{FV}_i^z = \sum_{t=1}^{L} (\boldsymbol{tpv}_i^t \langle tp_z \rangle \times pw^t). \tag{1}$$

As the unique real-vector for representing instruc-

tion i, $\boldsymbol{FV}_i$ is designed as a Z-dimensional vector that is synthesized from all properties in the list of $TSP\text{-}List_i$. Supposing $TSP\text{-}List_i$ has L properties, $TSP\text{-}List_i^t$ is the t-th property value for instruction i. $TSP\text{-}List_i^t$ can be quantified into a Z-dimension (Z-dim) (Z=10) real vector $\boldsymbol{tpv}_i^t = (tp_1, tp_2, \ldots, tp_Z)$ by word2vec[8], while tp_z is the z-th ($1 \leqslant z \leqslant Z$) real element in the vector, i.e., $\boldsymbol{tpv}_i^t\langle tp_z\rangle$. As for $\boldsymbol{FV}_i^z$ ($1 \leqslant z \leqslant Z$) which is the z-th real element of $\boldsymbol{FV}_i$, it is synthesized by integrating each tp_z of $\boldsymbol{tpv}_i^t$ (from $\boldsymbol{tpv}_i^1$ to $\boldsymbol{tpv}_i^L$) with a corresponding personalized weight pw for $\boldsymbol{tpv}_i^t$ according to (1)[9]. Therefore, it can be seen that $\boldsymbol{FV}$ for instructions still relies on a group of ideal personalized weight pw for TSP-List to complete.

Tuning pw for TSP-List. Each personalized weight pw_x for TSP^x in TSP-List is a significant factor to indicate its significance instruction representation. A tuning process is designed to obtain a group of ideal pw from all selected sample ISAs. The tuning is based on similarity evaluation with feature vectors for instructions by the Euclidean distance[③].

$$SD(i,j) = \sqrt{\sum_{z=1}^{Z}(\boldsymbol{FV}_i^z - \boldsymbol{FV}_j^z)^2}. \tag{2}$$

As for instructions i and j, the similarity distance $SD(i,j)$ is calculated with $\boldsymbol{FV}_i$ and $\boldsymbol{FV}_j$ with (2), while $\boldsymbol{FV}_i^z$ and $\boldsymbol{FV}_j^z$ are the z-dimension real number of these two vectors respectively. We use qualified $SD(i,j)$ for pair of similar instruction (i, j) (once $SD(i,j) \leqslant$ QUALIFIED_SD), while the threshold QUALIFIED_SD is set to 0.2 in our work. The tuning process aims for a group of ideal pw for TSP-List which can result in maximum qualified SD pairs.

As the tuning process in Algorithm 4 shows, it takes three steps to obtain a group of ideal pw for TSP-List. First, it groups peer instructions from all sample ISAs as shown in line 5 under two conditions: 1) the instructions have similar instruction names, such as `jr16` and `jr32`, `add` and `add.d`; 2) the instructions have $TSP\text{-}List$ which maps to exactly the same entries in TSP-List. These peer instructions are observed to have highly similar $Rec\text{-}List$, i.e., similar code-layout. Then, under these two conditions, peer instructions are clustered into strong connected subgraphs (SCSG). All SCSGs are connected to form a global strong connected graph (SCG). A basic principle underlined is that peer instructions should be similar enough, or should have qualified SD between each other. After the SCG is constructed, the tuning process undergoes approximately 50 000 iterations with different sets of parameters (pw) and makes statistics on instruction pairs with qualified SD. It keeps the one as an ideal group which can obtain maximum qualified SD for TSP-List.

Algorithm 4. Tuning on pw

Input: $ISAs$: all selected sample ISAs
Output: $ideal\_pw$: a group of pw
1 # QUALIFIED_SD 0.2; (threshold for SD)
2 $pw\_set \leftarrow$ generate a number of pw combinations in preset range;
3 **for** isa_i, $isa_j \in ISAs$ **do**
4 **for** inst $i \in isa_i$, $j \in isa_j$ **do**
5 $inst\_group$, $SCSG \leftarrow GroupPeerInsts(i,j)$;
6 add ($inst\_group$, $SCSG$) into $PeerInsts$
7 ($inst\_groups$, SCG) $\leftarrow ConstructSCG(PeerInsts)$;
8 **for** $pw \in pw\_set$ **do**
9 **for** $group \in inst\_groups$ **do**
10 **for** inst i, $j \in group$ **do**
11 $\boldsymbol{FV}_i \leftarrow Synthesize\_FV(GetTSP\text{-}List(i), pw)$;
12 $\boldsymbol{FV}_j \leftarrow Synthesize\_FV(GetTSP\text{-}List(j), pw)$;
13 $SD \leftarrow Eucilidean\_Distance(\boldsymbol{FV}_i, \boldsymbol{FV}_j)$;
14 **if** $SD <$ QUALIFIED_SD **then**
15 $qualified\_pairs$++;
16 **if** $qualified\_pairs > ideal\_pairs$ **then**
17 $ideal\_pairs \leftarrow qualified\_pairs$;
18 $ideal\_pw \leftarrow pw$;
19 **return** $ideal\_pw$;

Initially, the group of pw in tuning is set with empirical value total up to 1.0. According to our studies on MIPS ISAs, by experience, pw for the instruction name, the instruction size and their function (branch or memory) are initialized relatively higher (= 0.3) due to their significance in highlighting the difference in instructions, and are tuned in [0.2, 0.3]. The more frequently used properties in the middle end of the Y-axis of Fig.6 are initialized with 0.2 and tuned in [0.05, 0.2], and the rest properties are relatively unique and are initialized with 0.05, and tuned in [0, 0.05]. Then, more than 50 000 groups of pw are randomly generated under these constraints. The tuning process now takes four days which could be optimized in the future[10, 11].

After the tuning process on pw, the collection on an instruction representation is completed. After acquiring all instruction information, the construction on CLT-List is completed, and ATG is ready for code generation.

[③] https://en.wikipedia.org/wiki/Euclidean_distance, Sept. 2021.

3.2 Generation Stage

When a new ISA is required, the developers are required to fill in TSP-List with proper property values for every instruction. A new instruction t is first quantified with a feature vector $\boldsymbol{FV}_t$ with (1). And then it tries to match with available instruction models in CLT-List via SD evaluation with (2) and returns an instruction m which has the minimum qualified SD as a success match. New code for t can be generated automatically from $TSP\text{-}List_m$ by substituting each TSP^i with corresponding values offered by users. This is the basic scheme in ATG. However, ATG still may fail to match a new instruction. The auxiliary scheme is thus proposed as a supplement. For those failed instructions without new properties, we provide with an auxiliary template (referred to as AUX-TPL) shown in Fig.7, and it contains a group of `def` statements which can be defined with all property values in TSP-List offered by developers. The PD variables in `def` statements are original PD tokens derived from existing targets. Therefore, the AUX-TPL record can be interpreted by TableGen successfully, and PD variables can carry these values to the compiler backend correctly. This solution can work out this kind of failure successfully and it is evaluated in Section 5.

For those failed instructions due to additional properties or due to more complex reference patterns on target-specific properties which are never captured in available ISAs, ATG still cannot handle properly due to the lack of semantics. ATG needs to know more about the new instructions such as the property type, the reasonable reference pattern on properties. In this case, ATG should encourage developers to design a short piece of code manually which is refereed as the incremental scheme. Afterwards, ATG can make an incremental update on TSP-List and CLT-List, including adding new properties and re-tuning pw, and finally supplement CLT-List with new code-layout templates. Although this incremental update may take some time, it can benefit more similar instructions. In this way, the incremental scheme makes ATG competent for more practical scenarios.

3.3 Discussion on ATG's Generality

LLVM is a popular compiler framework among mainstream general-purpose compiler architectures. The prominent modularity and robustness make LLVM a preferred platform in the latest researches for both isomorphic and heterogeneous architectures[12, 13]. This work is a preliminary outcome towards a new efficient method for the automatic compiler design, which targets at a compiler developed via automatic code generation. This automatic mechanism asks for better modularity and regularity. From this perspective, LLVM is an ideal option as the base compiler framework. ATG can eventually lower the hardness in compiler development for both fast retargetability and optimization of compilers, and catch up with the fast speed of agile chip development in the future.

ATG is a general mechanism that can be applied

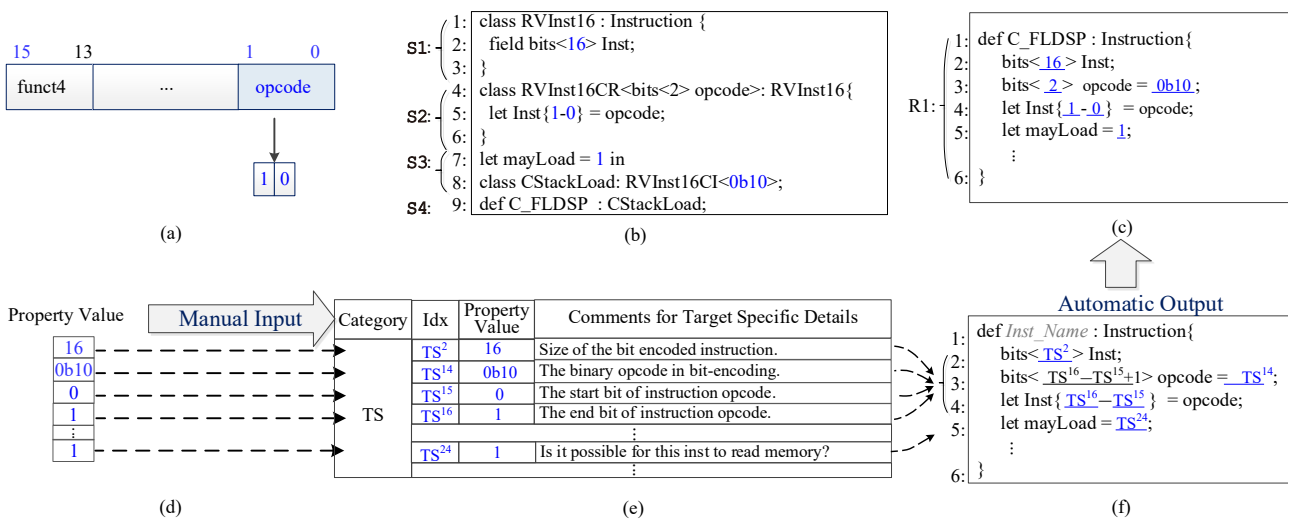

Fig.7. Code-layout template for failed match instructions. (a) Bit encoding of c.fldsp from RISC-V. (b) Hand-written description of c.fldsp taken from LLVM. (c) Automatically generated description of c.fldsp by ATG. (d) Only effort by developers for c.fldsp. (e) Target-specific properity list (TSP-List). (f) Auxiliary template (AUX-TPL) for failed match instructions.

to compiler architectures similar to LLVM. As introduced in Fig.3, general-purpose compilers such as GCC and LLVM are leveraged by using a certain language to manually describe hardware information (e.g., Tablegen Language for LLVM, Register Transfer Language[14] for GCC). ATG takes effect for these mainstream compilers by substituting the heavy manual efforts with a small TSP-List table. The framework in Fig.5 demonstrates the working stages of ATG. As to the above general-purpose compilers, ATG works with the same steps except the input files on which it is based. During the synthetic stage, it takes different types of files as input from various compilers before TRG-Construction, e.g., *.td files in LLVM, or *.md files in GCC. After that, ATG can turn out TSP-List and CLT-List in the same way. During the generation stage, with required input for TSP-List, ATG can produce corresponding hardware description files automatically.

4 Implementation

ATG is implemented as a standalone approach in about 8 000 lines of code in Python presently. It includes two separate stages, the synthetic stage for reusable TSP-List and CLT-List which takes relatively longer time, and a fast generating stage for a new ISA in minutes.

The synthetic stage is based on 11 MIPS ISAs from LLVM 12.0, which amount to about 1 700 instruction models. It takes *.td files from these ISAs as input. Then, TRG-Construction in Algorithm 1 sets up target-specific TRG for each of them and maintains the R-D relation for tokens. Next, TSP-List can be synthesized from TS tokens via token classification as shown in Algorithm 2. CLT-List is synthesized from instruction models. As shown in Algorithm 3, during this step, instructions are represented from three aspects including associate records, target-specific properties, and unique feature vectors which rely on a group of ideal pw on TSP-List with Algorithm 4. In our work, the tuning on pw is a time-consuming process. However, ATG is still an efficient approach compared with manual efforts.

The generation process is applied to nine RISC-V ISAs which amount to 1 029 instructions including 404 scalar instructions and 625 vector ones. We have taken one week to fill in TSP-List for these ISAs, which is the common overhead in the manual design. Then ATG can work for code generation on instruction combinations in minutes. ATG is evaluated via comparison with available RISC-V support in LLVM 12.0. The details for the ATG application is presented in Section 5.

5 Evaluations

In this section, we certify ATG as a qualified approach for an automatic target generator, significantly reducing efforts required.

5.1 Methodology

ATG is designed with a re-usable TSP-List and CLT-List, and then it can be reused by similar ISAs. We evaluate ATG from two aspects: 1) whether it is able to create *.td files for the selected ISA (reusability), and 2) whether the newly generated backend (LLVM_{ATG}) can create correct cases with the selected ISA (compilation ability). We select nine types of RISC-V ISA in RV32-mode and RV64-mode in the evaluation. As shown in Table 2, the standard ISA types include I, M, F, and D. The custom ISA types include A, C, B, FH, and RISC-V Vector extensions in V. Each ISA has various instructions in RV32-mode and RV64-mode, which amount to 906 and 1 029 instructions respectively.

Our compiler developer manually fills in TSP-List with properties for all the above instructions. While we provide TSP-List with 61 entries and most of these instructions only need 20 properties on average. The reusability of ATG is then evaluated with nine groups of ISA combinations in two modes as shown in Table 2 which provides flexible and comprehensive type coverage on the standard type (G1–G3), the custom type (G4–G8) and G9 for the complete ISA sets. These combinations can lead to 18 LLVM_{ATG} with different backends which are qualified to validate ATG's reusability (Subsection 5.2).

The compilation ability of ATG is evaluated via comparison between LLVM_{ATG} and LLVM with considerable test suites in Table 3. The LLVM regression test suite[④] includes IR files (.ll files), assembly codes (.asm files) and general functional tests, which amount to about 15 632 and 15 616 cases in RV32-mode and RV64-mode respectively. Moreover, 16

[④]https://llvm.org/docs/TestingGuide.html#regression-tests, Sept. 2021.

Table 2. Combinations on RISC-V ISA in RV32-Mode and RV64-Mode

ISA	Type	Description	Inst. Mode		Combination on ISA								
			RV32	RV64	G1	G2	G3	G4	G5	G6	G7	G8	G9
Standard	I	Base integer instructions	47	59	✓	✓	✓	✓	✓	✓	✓	✓	✓
	M	Standard extension for integer multiplication and division instructions	8	13		✓	✓	✓	✓	✓	✓	✓	✓
	F	Standard extension for single-precision floating-point instructions	26	30			✓	✓	✓	✓	✓	✓	✓
	D	Standard extension for double-precision floating-point instructions	26	32			✓	✓	✓	✓	✓	✓	✓
Custom	A	Standard extension for atomic instructions	44	88				✓					✓
	C	Standard extension for compressed instructions	37	48					✓				✓
	B	Standard extension for bit manipulation	63	100						✓			✓
	FH	Standard extension for half-precision floating-point instructions	30	34							✓		✓
	V	Standard extension for vector operations	625	625								✓	✓
Sum			906	1 029									

Table 3. Test Cases in the Evaluation

Test Suite	BenchMark Type	BenchMark Detail	Number of Validated Cases	
			RV32-Mode	RV64-Mode
SPEC 2017 CPU	C/C++	600.perlbench, 638.imagick, 605.mcf, 657.xz 623.xalancbmk, 511.povray, 619.lbm, 641.leela 526.blender, 508.namd, 510.parest, 602.gcc 631.deepsjeng, 620.omnetpp, 625.x264, 644.nab	16	16
LLVM regression test	LLVM IR	./llvm/test/CodeGen/RISCV/*.ll	452	408
		./llvm/test/DebugInfo/RISCV/*.ll	4	4
	Assembly	./llvm/test/MC/RISCV/*.s	137	165
		./llvm/test/DebugInfo/RISCV/*.s	1	1
	Others	General tests for LLVM backend	15 038	15 038
	Sum		15 632	15 616

C/C++ benchmarks in SPEC2017 are utilized (FORTRAN cases are not included due to unavailable toolchains on RISC-V). These test cases are compiled by LLVM_{ATG} and LLVM at options of "`-O0 -triple = riscv32`" for RV32 and "`-O0 -triple = riscv64`" for RV64. LLVM_{ATG} and LLVM are contrasted on assembly files, and also on the correct cases that can pass the built-in verification (Subsection 5.3).

Experiment Platform. ATG has no special requirements on the working environment. It now is implemented and evaluated on a server with Intel® Xeon® CPU E7-4809 v3 2.00 GHz and 256 GB of memory. The built-in verification for reliability, i.e., the correctness of test cases, is conducted on RISC-V QEMU Simulator.

5.2 Evaluation on Reusability

5.2.1 *Time Overhead in ATG*

The re-usablity of ATG relies on two re-usable lists of TSP-List and CLT-List which takes relatively higher time overhead. During the synthesizing process, tuning on ideal *pw* is the longest step which takes about four days based on 50 000 groups of attempts. This step could be sped up with recent researches on the training algorithm[10, 11]. The rest steps in the synthetic stage can be completed in minutes. When a new ISA is required, it takes 1 day–7 days for our developers to fill in TSP-List for these ISAs. This is a common overhead even in the manual design. After that, ATG can facilitate a fast code generating for a new ISA in minutes. In all, compared with traditional manual research and development, generating *.td files by ATG is still more efficient.

5.2.2 *Reusability of ATG*

The reusability of TSP-List and CLT-List is evaluated with combinations in Table 2. ATG can generate 18 sets of *.td files. In most of the combinations, the generated *.td files can fully cover the selected ISA by the basic scheme after being successfully matched in the standard ISA model. Only a few instructions fail to be generated due to mismatching. However, these failures can be solved by the auxiliary scheme which can supplement all the missing instructions in corresponding *.td files.

As summarized in Table 4, in the RV32-mode, the failed list by the basic scheme includes three C-type instructions in G5 and one B-type instruction in G6.

Table 4. Failed List in Instruction Matching

Combination	RV32-Mode	RV64-Mode
G4	-	amoxor.d/.d.aq/.d.rl/.d.aqrl
		amoand.d/.d.aq/.d.rl/.d.aqrl
G5	c.fldsp, c.fsd, c.fld	c.sd, c.sdsp, c.ldsp
		c.subw, c.fld, c.ld
		c.fldsp, c.addw, c.addiw, c.fsd
G6	sext.b	gorciw, crc32.d
		sext.b, gorcw, xperm.w
G9	All the above	All the above

They constitute the four failures in G9. Analogous results can be observed from experiments in RV64-mode. Despite of full instruction generation in most of the combinations, the mismatch for custom instructions leads to slight failures. There are eight A-type instructions, 10 C-type instructions, and five B-type instructions failing to be matched in G4, G5, and G6 respectively, which constitute 23 failures in G9.

The failures in similarity-based matching currently result from some unique combination of the input target-specific properties. These properties accompanied with available *pw* will turn out an ***FV*** with (1), which correspondingly results in a relatively significant distance in SD with (2) when matching with available instructions in CLT-List. This SD makes it hard to find a suitable code template for the new instruction in current reusable TSP-List. However, the auxiliary scheme is competent to solve these failures completely by generating corresponding code with our AUX-TPL in our work.

As the contrast shown in Table 5, the generated *.td files by both the basic scheme and the auxiliary scheme are larger than the hand-written ones in LLVM by 2x–3x per target. Among them, TS tokens account for about 6%–12% of all the tokens. TableGen interprets them into *.inc files which have the equivalent size but different code layout compared with the hand-written ones in LLVM. The differences mainly lie in different variables on the instruction name and the order of instructions in the code. After that, ATG can produce 18 backends which are exactly the same as those in LLVM. ATG does not result in noticeable time overhead on the compilation for LLVM_{ATG}.

5.3 Evaluation on the Compilation Ability

5.3.1 *Evaluation on Compiling*

LLVM_{ATG} does not show noticeable impact on the compiling process for all the cases in the evaluation. As for SPEC2017, only five instruction types including I, M, F, D and A are utilized by the benchmarks so that we only apply G1–G4 with SPEC2017. As we explained in Subsection 5.2, in these combinations, LLVM_{ATG} by both the basic scheme and the auxiliary scheme has the same backends with that in LLVM. Therefore, LLVM_{ATG} can produce the same assemble files and binaries as LLVM in the RV32-mode and RV64-mode.

As for the 15 600 cases in LLVM regression test-suite, G1–G9 combinations are conducted as shown in Fig.8. With the basic scheme, there are 6, 4, and 10 cases that fail to be generated in G5, G6, and G9 respectively in the RV32-mode due to failures in instruction support summarized in Table 4. Similarly, for RV64, despite of the full instruction generations in most of the combinations, the mismatch for previous mentioned custom instructions results in 3, 7, 8, and 18 failures in G4, G5, G6 and G9 respectively. Compared with the quantity of success cases, the failed ratio (< 0.1%) is acceptable. All the failures can be completely corrected by the auxiliary scheme, which can produce the same assemble files and binaries as LLVM in all combinations.

Table 5. Successfully Generated Instructions in RV32-Mode and RV64-Mode

Comb.	RV32-Mode						RV64-Mode					
	LLVM	Basic Scheme	Auxiliary Scheme	Failed	File Size Ratio (x)	TS Proportion (%)	LLVM	Basic Scheme	Auxiliary Scheme	Failed	File Size Ratio (x)	TS Proportion (%)
G1	47	47	0	0	3.0	6	59	59	0	0	2.7	7
G2	55	55	0	0	3.0	6	72	72	0	0	2.7	7
G3	107	107	0	0	2.8	6	134	134	0	0	2.5	7
G4	151	151	0	0	2.7	6	222	214	8	0	2.6	8
G5	144	141	3	0	2.5	6	181	171	10	0	2.3	7
G6	170	169	1	0	3.1	6	234	229	5	0	2.9	7
G7	137	137	0	0	2.7	6	168	168	0	0	2.4	7
G8	732	732	0	0	2.5	12	759	759	0	0	2.4	12
G9	906	902	4	0	2.5	10	1 029	1 006	23	0	2.5	11

Note: Comb.: combination.

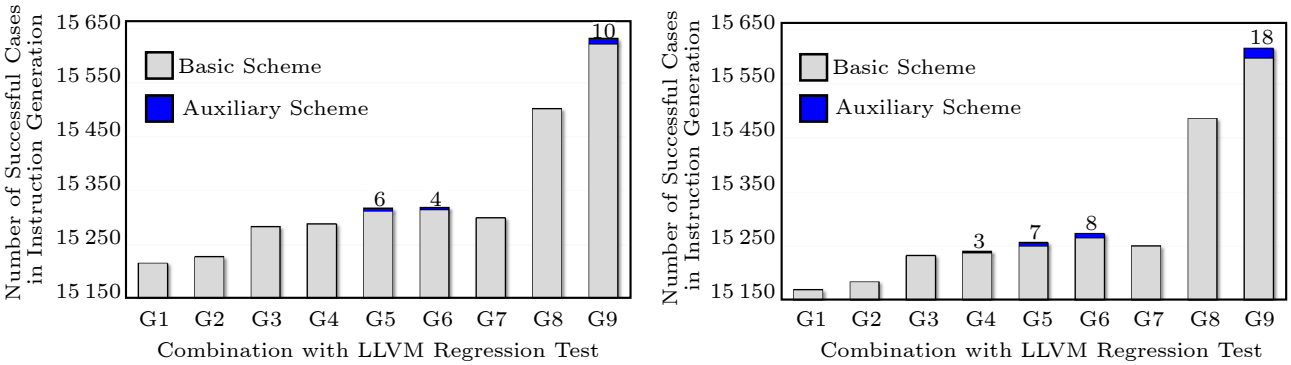

Fig.8. Experiments on instruction generation with LLVM regression tests. (a) RV32-mode. (b) RV64-mode.

5.3.2 *Validation on Running*

As we explained in Subsection 5.2, the backend in LLVM_{ATG} is the same with that in LLVM. Therefore, the cases generated by LLVM_{ATG} are the same with those by LLVM, and they can pass the built-in validation of both the LLVM regression test and SPEC2017. LLVM_{ATG} does not show any performance side-effects on these cases.

5.4 Summary

Experiential results in Subsection 5.2, especially with custom instruction types including the hot RVV set, certify ATG as a feasible mechanism for a much improved efficiency in manual development.

5.5 Discussion on Limitations

Despite the reusability, ATG is still incomplete to generate files in languages other than TableGen. Efforts aiming at C or C++ code are now undergoing for a more comprehensive target support. Moreover, further efforts about extending ATG to a wider scope such as the instruction scheduling in the compiler are undergoing. Currently, the evaluation of ATG still depends on the available LLVM infrastructure such as instruction selection and instruction emission with some manual assistance. Therefore, as we explained in Subection 3.2, ATG does not support customized instructions with complex mode or properties (such as pattern, immediate operand definition) due to the lack of relevant knowledge from existing ISAs. However, this more intellectual auto-designing approach is hopeful to further lower the threshold for compiler development fundamentally.

6 Related Work

The new trend brought forth by Agile chip design[1–3] is beckoning innovations on efficient compiler research and development. Latest researches[1, 15–18] foresee a vision of enhanced quality and improved efficiency in both hardware and software.

Modularity/Template is an efficient solution due to easier upstanding, easier reuse and scaling, but still with redundancy hard to overlook[4, 19–24]. Modularity is extensively used in domains including compilers such as LLVM, GCC, and KEQ[25]. It is also an attractive topic in agile design[26–28], such as TABLA[26], Chipkit[27] and OpenFPGA[28], which are applied in FPGA design or reusable SoC subsystems for tape-outs.

Generators are becoming more pervasive in the domain of compilers and specialized hardware accelerators for reduction on uncaught bugs, and lowering the demands in background knowledge[25, 29–31]. VEGEN[29] can generate a group of vectorization patterns automatically. CLGen[30] suggests an OpenCL generator for better runtime performance. KEQ[25] generates an equivalence checker automatically which proves equivalence for transformation from LLVM IR to X86_64. There are many outstanding studies on accelerators for hardware[26, 28, 32–34], which overcome bottlenecks of specific applications. ATG is a black box generator. With the only input on the TSP-List, it can generate *.td files.

Intermediate language/representation has been tackled for decades for both hardware generation and compilers. There are IRs for HLS[35], HDL[36] or compilers such as LLVM IR[37], GCC's internal IR[38], and earlier WHIRL[39]. They work as links between major components in a compiler which makes optimizations easier to re-configure. Calyx[40] implements a high lev-

el intermediate language for facilitating the design for custom hardware accelerators. MLIR[16] improves compilation performances for domain-specific hardware. ValueGraph[41, 42] is for validating the optimizations of the LLVM pipeline and thus improving runtime performance. BoogieIR[43, 44] is for taming the complexity of the program verification. Halide IR[45] can accelerate image processing process. These simpler techniques can speed up the efficiency in development. However, developers still need accumulated knowledge to better master new skills. ATG hides implementation details from users and thus requires no penetration.

7 Conclusions

We proposed ATG which can generate target description files from a simple TSP-List for a new target support in a compiler. ATG models sample ISAs (instruction set architectures) based on analyses on tokens, and normalize them into a standard target model including TSP-List for full specification on machine-specific constraints, and a standard ISA model for normalized record format. ATG can automatically generate target description files and greatly accelerate the generation of compiler backends over nine RISC-V ISAs, which can produce accurate code for 16 C/C++ SPEC2017 benchmarks and about 15 600 LLVM regression tests. ATG works in a black-box way. Except a small amount of indispensable properties, it does not ask for manual intervention on the whole process. Further exploration on available modularity and regularity inside compilers is still under work. ATG focuses on making radical changes in the retargetability of custom chips, greatly reducing the need for manual effort.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Bao Y G, Carlson T E. Agile and open-source hardware. *IEEE Micro*, 2020, 40(4): 6–9. DOI: 10.1109/MM.2020.3002606.

[2] Bahr R, Barrett C, Bhagdikar N, Carsello A, Daly R, Donovick C, Durst D, Fatahalian K, Feng K, Hanrahan P, Hofstee T, Horowitz M, Huff D, Kjolstad F, Kong T, Liu Q Y, Mann M, Melchert J, Nayak A, Niemetz A, Nyengele G, Raina P, Richardson S, Setaluri R, Setter J, Sreedhar K, Strange M, Thomas J, Torng C, Truong L, Tsiskaridze N, Zhang K Y. Creating an agile hardware design flow. In *Proc. the 57th ACM/IEEE Design Automation Conference*, July 2020, Article No. 142. DOI: 10.1109/DAC18072.2020.9218553.

[3] Fuchs A, Wentzlaff D. The accelerator wall: Limits of chip specialization. In *Proc. the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2019. DOI: 10.1109/HPCA.2019.00023.

[4] Collberg C S. Automatic derivation of compiler machine descriptions. *ACM Trans. Programming Languages and Systems*, 2002, 24(4): 369–408. DOI: 10.1145/567097.567100.

[5] Lopes B C, Auler R. Getting Started with LLVM Core Libraries. Packt Publishing Ltd, 2014.

[6] Leroy X. Formally verifying a compiler: Why? How? How far? In *Proc. the 9th International Symposium on Code Generation and Optimization*, Apr. 2011. DOI: 10.1109/CGO.2011.5764668.

[7] Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques, and Tools. Addison-Wesley Longman Publishing Co., Inc., 1986.

[8] Goldberg Y, Levy O. word2vec Explained: Deriving Mikolov *et al.*'s negative-sampling word-embedding method. arXiv: 1402.3722, 2014. https://arxiv.org/abs/1402.3722, Nov. 2023.

[9] Arora S, Liang Y Y, Ma T Y. A simple but tough-to-beat baseline for sentence embeddings. In *Proc. the 5th International Conference on Learning Representations*, Apr. 2017.

[10] Simos T E, Famelis I T. A neural network training algorithm for singular perturbation boundary value problems. *Neural Computing and Applications*, 2022, 34(1): 607–615. DOI: 10.1007/s00521-021-06364-1.

[11] Vural N M, Ergüt S, Kozat S S. An efficient and effective second-order training algorithm for LSTM-based adaptive learning. *IEEE Trans. Signal Processing*, 2021, 69: 2541–2554. DOI: 10.1109/TSP.2021.3071566.

[12] Haidl M, Moll S, Klein L, Sun H H, Hack S, Gorlatch S. PACXXv2 + RV: An LLVM-based portable high-performance programming model. In *Proc. the 4th Workshop on the LLVM Compiler Infrastructure in HPC*, Nov. 2017, Article No. 7. DOI: 10.1145/3148173.3148185.

[13] Barchi F, Urgese G, Macii E, Acquaviva A. Code mapping in heterogeneous platforms using deep learning and LLVM-IR. In *Proc. the 56th ACM/IEEE Design Automation Conference (DAC)*, Jun. 2019, Article No. 170. DOI: 10.1145/3316781.3317789.

[14] Davidson J W, Fraser C W. The design and application of a retargetable peephole optimizer. *ACM Trans. Programming Languages and Systems*, 1980, 2(2): 191–202. DOI: 10.1145/357094.357098.

[15] Lattner C. The golden age of compiler design in an era of HW/SW co-design. Technical Report, 2021. https://asplos-conference.org/asplos2021/index.html%3Fp=2355.html, November 2023.

[16] Lattner C, Tatiana S. MLIR: Multi-level intermediate

representation compiler infrastructure. Technical Report, 2020. https://cgo-conference.github.io/cgo2020/keynotes, November 2023.

[17] Cai B M, Ashwathnarayan S, Shafiq F, Eltantawy A, Azimi R, Gao Y Q. Exploring agile hardware/software co-design methodology. Technical Report, 2020. https://jnamaral.github.io/SSHAW/program.html, November 2023.

[18] Graf A. Compiler backend generation using the VADL processor description language [Ph.D. Thesis]. Technische Universität Wien, Wien, 2021.

[19] Sullivan K J, Griswold W G, Cai Y F, Hallen B. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 2001, 26(5): 99–108. DOI: 10.1145/503271.503224.

[20] Baldwin C Y, Clark K B. Design Rules: The Power of Modularity. MIT Press, 2000. DOI: 10.7551/mitpress/2366.001.0001.

[21] Tsantalis N, Mazinanian D, Krishnan G P. Assessing the refactorability of software clones. *IEEE Trans. Software Engineering*, 2015, 41(11): 1055–1090. DOI: 10.1109/TSE.2015.2448531.

[22] Ganzinger H. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming*, 1983, 3(3): 223–278. DOI: 10.1016/0167-6423(83)90021-7.

[23] Kastens U, Waite W M. Modularity and reusability in attribute grammars. *Acta Informatica*, 1994, 31(7): 601–627. DOI: 10.1007/BF01177548.

[24] Harper R, Lillibridge M. A type-theoretic approach to higher-order modules with sharing. In *Proc. the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Feb. 1994, pp.123–137. DOI: 10.1145/174675.176927.

[25] Kasampalis T, Park D, Lin Z Y, Adve V S, Roşu G. Language-parametric compiler validation with application to LLVM. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2021, pp.1004–1019. DOI: 10.1145/3445814.3446751.

[26] Mahajan D, Park J, Amaro E, Sharma H, Yazdanbakhsh A, Kim J K, Esmaeilzadeh H. TABLA: A unified template-based framework for accelerating statistical machine learning. In *Proc. the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Mar. 2016, pp.14–26. DOI: 10.1109/HPCA.2016.7446050.

[27] Whatmough P N, Donato M, Ko G G, Lee S K, Brooks D, Wei G Y. CHIPKIT: An agile, reusable open-source framework for rapid test chip development. *IEEE Micro*, 2020, 40(4): 32–40. DOI: 10.1109/MM.2020.2995809.

[28] Tang X F, Giacomin E, Chauviere B, Alacchi A, Gaillardon P E. OpenFPGA: An open-source framework for agile prototyping customizable FPGAs. *IEEE Micro*, 2020, 40(4): 41–48. DOI: 10.1109/MM.2020.2995854.

[29] Chen Y S, Mendis C, Carbin M, Amarasinghe S. VeGen: A vectorizer generator for SIMD and beyond. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2021, pp.902–914. DOI: 10.1145/3445814.3446692.

[30] Cummins C, Petoumenos P, Wang Z, Leather H. Synthesizing benchmarks for predictive modeling. In *Proc. the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2017, pp.86–99. DOI: 10.1109/CGO.2017.7863731.

[31] Lim J P, Nagarakatte S. Automatic equivalence checking for assembly implementations of cryptography libraries. In *Proc. the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2019, pp.37–49. DOI: 10.1109/CGO.2019.8661180.

[32] Liu D F, Chen T S, Liu S L, Zhou J H, Zhou S Y, Teman O, Feng X B, Zhou X H, Chen Y J. PuDianNao: A polyvalent machine learning accelerator. *ACM SIGPLAN Notices*, 2015, 50(4): 369–381. DOI: 10.1145/2775054.2694358.

[33] Chen T Q, Moreau T, Jiang Z H, Zheng L M, Yan E, Cowan M, Shen H C, Wang L Y, Hu Y W, Ceze L, Guestrin C, Krishnamurthy A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proc. the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI 18)*, Oct. 2018, pp.579–594.

[34] Ham T J, Wu L S, Sundaram N, Satish N, Martonosi M. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, Article No. 56. DOI: 10.1109/MICRO.2016.7783759.

[35] Coussy P, Morawiec A. High-Level Synthesis. Springer, 2008.

[36] Thomas D, Moorby P. The Verilog® Hardware Description Language. Springer, 2008.

[37] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. the 2004 International Symposium on Code Generation and Optimization*, Mar. 2004, pp.75–86. DOI: 10.1109/CGO.2004.1281665.

[38] Pilato C, Ferrandi F. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *Proc. the 23rd International Conference on Field Programmable Logic and Applications*, Sept. 2013. DOI: 10.1109/FPL.2013.6645550.

[39] Cohen W W. A demonstration of WHIRL (demonstration abstract). In *Proc. the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Aug. 1999, p.327. DOI: 10.1145/312624.312763.

[40] Tate R, Stepp M, Tatlock Z, Lerner S. Equality saturation: A new approach to optimization. In *Proc. the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2009, pp.264–276.

DOI: 10.1145/1480881.1480915.

[41] Tristan J B, Govereau P, Morrisett G. Evaluating value-graph translation validation for LLVM. In *Proc. the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 2011, pp.295–305. DOI: 10.1145/1993498.1993533.

[42] Hawblitzel C, Lahiri S K, Pawar K, Hashmi H, Gokbulut S, Fernando L, Detlefs D, Wadsworth S. Will you still compile me tomorrow? Static cross-version compiler validation. In *Proc. the 9th Joint Meeting on Foundations of Software Engineering*, Aug. 2013, pp.191–201. DOI: 10.1145/2491411.2491442.

[43] Barnett M, Chang B Y E, DeLine R, Jacobs B, Leino K R M. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. the 4th International Conference on Formal Methods for Components and Objects*, Nov. 2005, pp.364–387. DOI: 10.1007/11804192_1.

[44] Ragan-Kelley J, Barnes C, Adams A, Paris S, Durand F, Amarasinghe S. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 2013, 48(6): 519–530. DOI: 10.1145/2499370.2462176.

[45] Nigam R, Thomas S, Li Z J, Sampson A. A compiler infrastructure for accelerator generators. In *Proc. the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2021, pp.804–817. DOI: 10.1145/3445814.3446712.

Hong-Na Geng received her B.S. degree in computer science and technology from Wuhan University of Science and Technology, Wuhan, in 2017. She is currently a Ph.D. candidate in the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests are compiler optimization and agile compilation technology. She is a student member of CCF.

Fang Lyu is a senior engineer at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing. She received her Ph.D. degree in computer architecture from the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2014. Her main research interests include architecture-oriented performance analysis and optimization, compiler optimizations, etc. She is a member of CCF.

Ming Zhong received his B.S. degree in computer science and technology from Beijing University of Posts and Telecommunications, Beijing, in 2021. He is currently a Master student in the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interest is agile compilation technology. He is a student member of CCF.

Hui-Min Cui is a professor and Ph.D. supervisor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. She received her Ph.D. degree in computer architecture from the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2011. Her main research interests include programming framework for heterogeneous, parallel compilation, etc. She is a member of CCF.

Jingling Xue is a Scientia Professor in the School of Computer Science and Engineering at the University of New South Wales, Sydney, where he leads the Programming Languages and Compilers group. He received his Ph.D. degree in computer science and engineering from Edinburgh University, Edinburgh, in 1992. His research spans programming languages, compiler technology, and program analysis. He is a senior member of IEEE.

Xiao-Bing Feng is a professor and Ph.D. supervisor at Institute of Computing Technology, Chinese Academy of Sciences, Beijing. He received his Ph.D. degree in computer architecture from the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 1999. His main research interests include compiler optimization and binary translation. He is a senior member of CCF.