Irregular Register Allocation for Translation of Test-pattern Programs

MINSU KIM, JEONG-KEUN PARK, and SOO-MOOK MOON, Seoul National University, South Korea

Test-pattern programs are for testing DRAM memory chips. They run on a special embedded system called *automated test equipment* (ATE). Each ATE manufacturer provides its own programming language, which is mostly low level, thus accessing the registers in the ATE directly. The register structure of each ATE is quite different and highly irregular. Since DRAM chipmakers are often equipped with diverse ATEs from different manufacturers, they employ automatic translation of a program developed for one ATE to a program for different ATEs. This raises an *irregular register allocation* problem during translation. This article proposes a solution based on partitioned Boolean quadratic programming (PBQP). PBQP has been used for a number of compiler optimizations, including *paired register allocation*, which our ATE register allocation also requires. Moreover, the *interleaved* processing in ATE incurs complex register constraints, which we could also formulate elegantly with PBQP. The original PBQP solver is not quite appropriate to use, though, since ATE register allocation does not allow spills, so we devised a more elaborate PBQP solver that trades off the allocation time and allocation search space, to find a solution in a reasonable amount of time. Our experimental results with product-level pattern programs show that the proposed register allocator successfully finds valid solutions in all cases, in the order of tenths of seconds.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computer systems organization** → **Embedded systems**; • **Hardware** → Test-pattern generation and fault simulation;

Additional Key Words and Phrases: Irregular register allocation, automated test equipment, partitioned Boolean quadratic programming, memory testing, test-pattern programs

ACM Reference format:

Minsu Kim, Jeong-Keun Park, and Soo-Mook Moon. 2020. Irregular Register Allocation for Translation of Test-pattern Programs. *ACM Trans. Archit. Code Optim.* 18, 1, Article 5 (December 2020), 23 pages. <https://doi.org/10.1145/3427378>

1 INTRODUCTION

The manufacturing process of DRAM memory chips includes a final step to validate their correct behavior. For this, we perform automated tests on every chip by using an automated test equipment (ATE), a special embedded system equipped with the dedicated processing units called algorithmic pattern generator (ALPG) and the registers. During the test, ATE physically places a chip on a slot

New article, not an extension of a conference paper.

This work was supported in part by Samsung Electronics Co., Ltd.

Authors' addresses: M. Kim, J.-K. Park, and S.-M. Moon (corresponding author), Room 851, Building 301, Seoul National University, 1 Gwanak-ro, Gwanak-gu, Seoul, 08826 Republic of Korea; emails: {mskim, erebus}@altair.snu.ac.kr, smoon@snu.ac.kr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Association for Computing Machinery.

1544-3566/2020/12-ART5

<https://doi.org/10.1145/3427378>

inside it, sends a pattern of signals (bit vectors) to the pins of the chip in each clock, and decides pass/fail depending on its behavior. ATE is programmable, and the programs running on the ATE to generate the bit vectors are called *test-pattern programs*.

A DRAM chipmaker often buys ATEs from more than one manufacturer for price competitiveness (an ATE costs around 3–5 million US dollars), and tests a DRAM product using any ATE that can generate the patterns with the same clock speed of the DRAM. One issue is that each ATE manufacturer provides its own programming language, mostly composed of assembly-like syntax using the commands of the ALPG and accessing the registers directly. This makes pattern programming not only complex and time-consuming but also *not portable* among the ATEs of different manufacturers. Furthermore, even different models of the same manufacturer have a different hardware configuration (e.g., number of ALPGs and registers) even with different language primitives, so they also have a portability issue.

To solve this compatibility issue, open standards and interfaces have been proposed, which allow the test engineers to develop a program using a universal high-level or low-level language, independent of the ATE manufacturers or ATE models [8, 16, 23]. Then, the universal program is translated into a pattern program for a specific target ATE. Unfortunately, these approaches are not widely accepted by the DRAM industry as far as we know, since the major ATE manufacturers do not support the standards and it is hard for the test engineers to learn a new interface [28].

However, the chipmakers do need portability, so they often employ another solution, *program translation*. For a given device under test (DUT), test engineers develop and verify a pattern program for one ATE, then translate it to a program for a different ATE that can test the DUT. Since there are many diverse ATEs in the ATE farm of a chipmaker, and since any ATE can be a source or a target of translation, we need an efficient and aggressive translation system. For this, we need to redefine some of the important compilation issues such as intermediate representation (IR), register allocation, and instruction scheduling. For example, we proposed an output-based IR where the output of a pattern program serves as an IR [18, 25], instead of conventional IRs such as the LLVM bitcode, since it is hard to define an universal IR for all ATEs while the output delivers the exact intention of the test engineer. In this article, we deal with the register allocation issue, which is quite challenging due to the highly irregular nature of the ATE registers.

ATE registers are often wired in a complex way, which leads to highly irregular *register pairing*. For example, we can add two registers A and B but cannot add two registers A and C. Another irregularity comes from ATE's multiple ALPG units, *interleaved* to provide concurrency to generate patterns faster than the clock speed of each ALPG. For example, an ATE with four ALPG units provides 4-way interleaving, requiring the number of statements in a loop to be a multiple of four, so that a bundle of four statements constitutes a *major cycle*. Within a major cycle, a register cannot be written more than once and cannot be read ahead of a write. Finally and most importantly, there are no memory load/store commands in the ATE, meaning that register spills are impossible. This requires an allocation strategy different from conventional register allocators, because allocation failure means translation failure.

There are many irregular register allocation techniques proposed for CPU or DSP compilation [3, 13, 19, 20, 27, 29]. Among those techniques, we chose an approach based on partitioned Boolean quadratic programming (PBQP) [29], a graph optimization problem, since the PBQP graph can easily model the register pairing requirement as well as the interference. Moreover, the bundling constraints of ATE registers can be elegantly formulated with PBQP. Unfortunately, the original PBQP solver based on the removal of “easy” nodes, followed by approximation of “hard” nodes, is not quite appropriate to use, since our PBQP graphs rarely have easy nodes. Also, approximation-based allocation might give up too early and fail to find a valid solution even if one exists, thus failing the translation. So we devise a more elaborate PBQP solver focusing on the

enumeration of those hard-to-allocate nodes, which trades off the allocation time and the allocation search space, to find a solution in a reasonable amount of time. Our experimental results with product-level pattern programs show that the proposed irregular register allocator successfully finds valid solutions in all cases, in the order of tenths of seconds. The elapsed time is acceptable, since the programs are translated in an offline manner.

The primary contributions of this article are as follows:

- We identified the irregular register allocation problem when translating the test-pattern programs, where we categorize the register constraints, some of which were not explored previously in irregular register allocation.
- We successfully formulated the ATE register allocation restrictions as a PBQP problem.
- We proposed a modified PBQP solver that can provide a better tradeoff between optimality and time-complexity, particularly for the cases of test-pattern programs.
- We demonstrate that our register allocation module works well for product-level test-pattern programs written to test DDR4 SDRAMs.

The rest of the article is organized as follows. Section 2 provides a brief background on memory testing, test-pattern programs, and the previous irregular register allocation using PBQP. Section 3 categorizes the constraints caused by irregular register architecture of ATE. Section 4 overviews our translation platform of test-pattern programs and explains how the proposed register allocation unit works in the platform. Section 5 shows how to represent the constraints as a PBQP graph and explains how the PBQP solver works. The evaluation result is in Section 6. Section 7 discusses related work. Section 8 shows the summary and future work.

2 BACKGROUND

2.1 Memory Structure and Operations

We briefly summarize the structure of DRAM memory. For example, x8 DDR4 SDRAM includes four banks, where each bank is a two-dimensional array of memory cells. A memory cell, which has 8-bit data, is addressed by a combination of a row address and a column address. So, an SDRAM chip includes 8-bit data pins and 17-bit address pins. It also has 5-bit command pins, used to trigger a state transition of the SDRAM, which operates as a finite state machine (FSM) (the typical states include *idle*, *activated*, *writing*, *reading*, and *pre-charging*). A bit vector sent through the command pins defines a command for each clock cycle. For example, a 5-bit vector (1, 0, 0, 1, 1) mapped to the command *write* (“WRT”) writes a data given through the data pins to the cell, whose column address is given through the address pins (the row address is provided by a separate command *activate* in advance).

2.2 Memory Testing Using ATE

ATE tests DRAM chips at high speed to detect one that has a behavioral fault [1, 24]. As shown in Figure 1(a), ATE moves a chip (or multiple chips at once) inside it using a physical handler. It then runs the executable compiled from a test-pattern program using the ALPGs and the registers. As a result, ATE generates and sends a bit-vector composed of command, address, and data bits to the corresponding pins of the chip at each clock. The pattern generation speed of the ATE should be the same as the operation speed of the chip, because, otherwise, the chip cannot operate as intended.

As the generated bit vectors are plugged into the chip, ATE compares the observed output with the expected output, which is also generated by executing the test-pattern program. If there is any mismatch, meaning that the chip fails, then ATE locates the unit in a proper failure bin categorized

Fig. 1. The overview of the automated memory testing process using ATE.

by its fault type; otherwise, the chip passed the test. Then, the ATE repeats the same test for the next chip. Since every chip should go through this test process before shipping, efficient yet thorough testing is important. So, the test engineers should write good-quality test-pattern programs that can detect all erroneous behaviors, based on the thorough understanding of the given DRAM architecture.

2.3 Test-pattern Programs

Typically, a test-pattern program traverses the memory cells (sequentially, diagonally, or in its own traversal algorithm) and repeatedly reads/writes data from/to the cells. It checks if the reads or writes work as expected, as well as if they do not affect the values of other (neighbor) cells. A test-pattern program is written in a domain-specific language of the target ATE. The programming language is mostly low level, expressing the hardware details of the ATE explicitly including the registers. Figure 1(b) shows a sketch of a test-pattern program for an ATE where each line is a statement, and A2, B1, C1, and D1 mean registers. Below, we list four hardware and language features related to register irregularity common to a few popular ATEs, which would complicate program translation.

Accessing physical registers. Test-pattern program often *directly* accesses physical registers without relying on compiler register allocation, unlike most general-purpose programs. So, the test engineers must understand the precise set of registers and their constraints. The constraints are more idiosyncratic than the ones for general CPU registers (see Section 3), and they are quite different among ATEs. This may complicate register allocation during translation.

Timing information. Each statement in the test-pattern program is associated with some *timing* information. In general programming languages, the order of the statements does not matter as long as the dependency is not broken. However, test-pattern program is intended to generate test-patterns at the exact timing described in the program, so it is not easy to reorder the statements or add additional statements, which can affect register allocation.

Interleaving. An ATE often includes multiple ALPGs that are *interleaved*, to generate patterns faster than the clock speed of ALPG. For example, the ATE in Figure 2 has four ALPGs. If each ALPG runs at a clock speed of 100 MHz, then such 4-way interleaving allows the ATE to run at a clock speed of 400 MHz by generating patterns alternately among the four ALPGs. This requires the pattern program to be *bundled* in a group of four statements, and the four statements will be

Fig. 2. How interleaving of ALPG units works. Even if the maximum speed each ALPG units generate signals is 100 MHz, for example, 4-way interleaving can lead to the output stream of 400 MHz.

executed concurrently (however, they still have sequential execution semantics). That is, the statements 1, 2, 3, and 4 are executed at the same time, each on a different ALPG. Similarly, statements 5, 6, 7, and 8 are executed at the same time. This requires the number of statements of a loop (even outer loop) to be a multiple of four as well. Such a statement bundle is called a *major cycle*, which causes many restrictions on register allocation (see Section 3).

No spill memory. There is no data memory other than the registers, so *spilling* is not an option during register allocation. Therefore, if register allocation during translation fails, it will fail the translation itself. This requires an allocation strategy different from conventional register allocation that can survive with spills even if allocation fails.

2.4 Partitioned Boolean Quadratic Programming for Register Allocation

A PBQP problem is a cost-minimizing allocation problem for given a PBQP graph [14, 29]. A PBQP graph $G(V, E, c)$ is an undirected graph with a collection of costs c , including a cost for each vertex $v \in V$ and a cost for each edge $(u, v) \in E$. Here, we denote a vertex cost by $\vec{c}_u$ for a node u and an edge cost by C_{uv} for an edge between a vertex u and a vertex v . As to the dimension of costs, $\vec{c}_u$ is an $m \times 1$ column vector and C_{uv} is an $m \times m$ matrix. Each element of a cost vector or matrix is in $\mathbb{R} \cup \{\infty\}$ (i.e., real numbers or infinity). Here, two vertices u and v are regarded as *disconnected* if and only if $C_{uv} = O$ (a matrix with all elements equal to zero).

Now, we introduce a selection vector $\vec{x}_u$, one for each vertex u (its dimension is also $m \times 1$). Here $\vec{x}_u \in \{0, 1\}^m$ and $\vec{x}_u \cdot \vec{1}^T = 1$ (for $\forall 1 \leq u \leq n$, if the number of vertices is n). In other words, each selection vector contains exactly a single value of one, meaning that the vertex selects that position, and zeros in the rest of the positions.

Finally, a *partitioned Boolean quadratic programming* problem is defined as to minimize the cost function f ,

$$f = \left[\sum_{1 \leq u \leq n} \vec{x}_u^T \cdot \vec{c}_u \right] + \left[\sum_{1 \leq u < v \leq n} \vec{x}_u^T \cdot C_{uv} \cdot \vec{x}_v \right], \quad (1)$$

where $n = |V|$ is the number of vertices.

For example, if the graph in Figure 3(a) is given as a PBQP graph, then the PBQP problem for this graph is to find three selection vectors (one for each vertex). The cost function f yields the sum of highlighted numbers in Figure 3(a), when the selection vectors are arbitrarily chosen as $\vec{x}_1^T = (0, 1)$, $\vec{x}_2^T = (0, 1)$, and $\vec{x}_3^T = (1, 0)$. The first term becomes $3 + 0 + 0 = 3$, and the second term becomes $4 + 5 + 9 = 18$, which yields $f = 3 + 18 = 21$. However, if we choose a different set of selection vectors, $\vec{x}_1^T = (1, 0)$, $\vec{x}_2^T = (0, 1)$, and $\vec{x}_3^T = (0, 1)$, then the cost function calculates $f = (4 + 0 + 1) + (2 + 3 + 2) = 12$, which happens to be the minimal as shown in Figure 3(b). As the search space of such selection vectors is as large as $O(m^n)$, the brute-force search for optimal

Fig. 3. An example of a PBQP graph with $n = 3$ and $m = 2$. Each column vector near the vertices represents $\vec{c}_i$, and each matrix near the edges represents C_{ij} ($i < j$).

solution would be intractable, so Scholz et al. proposed a PBQP solver based on graph reduction [14, 29], which will be explained later (see Section 5).

They also proposed how a register allocation problem can be modeled as a PBQP problem. It maps each virtual register in the program to one of n vertices. Each physical register is mapped to one of $m - 1$ positions, while the m_{th} position is reserved as a spill. Then, the *selection vector* of cardinality m can be interpreted as allocation to one of $m - 1$ physical registers or to the spill. Each entry of the cost vector or matrix can be either a finite number to represent the *allocation cost*, such as the spill cost or the execution cycle counts, or an *infinity* to represent that such a register allocation is not possible due to interference (i.e., a cost matrix C_{uv} whose diagonal values are all infinity means an interference between u and v). By using the cost vectors and matrices, the PBQP graph can express the register allocation constraints including the irregular ones more in detail than the conventional interference graph.

3 ATE REGISTER CONSTRAINTS

This section introduces and categorizes the register restrictions that the ATE hardware imposes on its test-pattern programs. The details are different depending on the ATE, but we covered many types of constraints found from our experience of translating product-level DRAM pattern programs of major ATE manufacturers. These restrictions are originated from the ATE architectures, not publicly available, so we inferred those constraints from the user manuals that contain a list of some legal and illegal use cases. We identified four types of constraints, which will be illustrated below. These constraints will be represented as the costs of the PBQP graph, as will be detailed in Section 5.

3.1 Interference and Class

The first category shows the common and well-known constraints: interference and class. Two virtual registers *interfere* if they are both live simultaneously at any program point. The *register class* represents the register type such as floating-point and integer in CPU. In ATE, there are two register classes: address and data. That is, address values can only be stored in address-class registers and data values can only be stored in data-class registers. This is so, since the signal lines connected to DUT's address pins are only muxed from address registers (the same for data pins and data registers).

For the program example in Figure 1, we assume that our ATE has five registers in total. Four of them are address registers: A1, A2, B1, C1, and one of them is a data register D1. Their names may be different in the real hardware and there are many more registers available, though; we abstracted the hardware as much as possible here for simplicity. Besides, the size of registers can

$x = y + z$

y

z

	A1	A2	B1	C1
A1	O	O	X	O
A2	O	O	X	O
B1	X	X	O	O
C1	X	X	X	X

Fig. 4. Allowed (O) and disallowed (X) register combinations for operands of an address addition $x = y + z$.

(a) Disallowed

```

1 A1 = ...
2 ... = A1
3 A2 = ...
4 ... = A2

```

(b) Allowed

```

1 A1 = ...
2 ... = A1
3 B1 = ...
4 ... = B1

```

Fig. 5. In a 4-way architecture, while the program (a) is disallowed, the program (b) is allowed.

also lead to irregularity (which we can be easily expressed in the PBQP graph), but for simplicity, we also assume they are long enough.

3.2 Operand Register Pairing

In some ATEs, some operations can use a limited set of registers as their operands. For example, Figure 4 illustrates that the addition of two address values can be done only between some registers only in some fixed order. That is, if the program contains an address addition $y + z$, then we can add A1 and C1 but not A1 and B1. Also, we can add B1 and C1 but not C1 and B1. This type of irregularity cannot be expressed by the traditional interference graph but can be easily modeled in the PBQP graph using the cost matrix (see Section 5.3).

3.3 Register Group

The *register group* is a constraint caused by the major cycle discussed in Section 2.3. Within a major cycle, it allows up to one definition of any physical registers in the same register group. This constraint is originated from the internal architectural design where some registers are shared by the interleaving ALPGs, so they cannot be written concurrently in the same major cycle. This also implies that a definition of any register can be placed only once in a major cycle.

For example, we assume that A1 and A2 are in the same register group. It means that a definition of A1 and a definition of A2 cannot occur within the same major cycle. Assume that each of registers B1, C1, and D1 is in its own register group. In Figure 5, if this program is running on a 4-way architecture where a major cycle consists of four statements, the program in Figure 5(a) is not allowed due to two definitions of A1 and A2 in the same register group within a major cycle boundary. However, Figure 5(b), is allowed, because A1 and B1 are in different register groups.

3.4 Definition-Use Ordering

This type of constraint is also due to the major cycle issue discussed in Section 2.3. A definition of a register cannot be located later than any *use* of the register within the same major cycle, even if the *use* is not reading the value defined later. This is also originated from the internal design

Fig. 6. In a 2-way architecture, while the program (a) is disallowed, the programs (b) and (c) are allowed.

Fig. 7. (a) The workflow of our translation system. (b) An example of the input for the register allocator. The program in Figure 1(b) is a desired output for this input, as it does not violate any register constraints of the target ATE in Section 3.

where the ALPGs write out values for definitions first and then fetch values for uses in a major cycle. Therefore, even if there is no data flow from a later definition to the use, all uses prior to the definition of the same register within a major cycle will be tainted. This is the reason why such use cases are prohibited from the source program level.

Figure 6 shows how this constraint affects programs for a 2-way interleaving ATE. The program (a) would have no problems at all in most general programming languages. However, it is not allowed as a test-pattern program, because a use of A1 in line 3 is in the same major cycle with a definition of A1 in line 4. However, the program (b) is permitted, because the use of A1 in line 3 does not get affected by the definition of A2 in line 4. The program (c) is also permitted, because now the major cycle boundaries are correctly aligned.

4 REGISTER ALLOCATION PASS

This section shows where the proposed register allocation module is located in our translation platform of test-pattern programs. Figure 7(a) illustrates the workflow of our translation system when translating an ATE B program to an ATE A program. We use the output-based IR, which we obtain by running (or simulating) the source test-pattern program (ATE B program in this case). From the output-based IR, we reconstruct the target program (ATE A program). The reason that we use the output as IR is that it is the simplest way to obtain a universal IR, not dependent on any ATE, yet delivering the programmer's exact intention [18]. That is, a pattern program has

no dynamic input and no conditional branches except for loop branches whose trip count is a constant, thus always generating the same output patterns.

When we reconstruct the target program, the registers are virtual yet (actually, they are still expressed using the source ATE's registers, which are different from the target ATE's registers). So they have to be allocated to the physical registers of the target ATE, complying with the ATE's register constraints. For the example pattern program in Figure 1, Figure 7(b) shows the reconstructed program, not yet register-allocated. Again, the real-life program looks different from it where we abstracted away ATE-specific details for simplicity.

The RA module also takes the *configuration file* written to describe details of the target ATE architecture required for register allocation. Specifically, the configuration file includes the number of physical registers available, the names, the classes, their characteristics, and the details of other constraints. The constraints are divided into four categories described in Section 3.1-3.4. We describe them in a JSON-like format that we defined. Consequently, our RA module can parameterize all target register configurations to flexibly cope with future changes of the ATE architecture.

Let us explain the program in Figure 7(b). Each line is a *statement*, a set of instructions that are executed exactly in one clock cycle. That is, an ALPG fetches and executes all instruction in line 1 at clock cycle 1, all instructions in line 2 at clock cycle 2, and so on. First, L1 (line 5) is the jump target label to which JMP L1 (line 12) jumps. Second, instructions like NOP, ACT, WRT and PRE, make the ATE generate corresponding *command* signals, which are then sent to DUT (see Section 2.1). For example, ACT encodes (1, 1, 0, 0, 0), which makes the DUT ready for upcoming read/write commands. Third, ADDR1 and ADDR2 indicate address-class *virtual* registers and DATA1 indicates a data-class *virtual* register. There are register operations such as ADDR3 = ADDR1 + ADDR2 (line 7). Finally, address signals are either X or Y and the data signal is D, and all of them use the ' $\leq$ ' notation to express *signal transmission* to DUT, using the values read from the virtual register in the right hand side. Both address signals and data signals are supposed to be transmitted at every clock cycle, so those lines with no explicit transmission code will transmit the values from the same virtual registers in their previous line (e.g., in line 7, Y and D get the same values as in line 6).

Now, the register allocation module aims to find an appropriate selection of a physical register for each virtual register in Figure 7(b). The program back in Figure 1(b) is a good example of a register-allocated output program, assuming that the target ATE is as in Figure 1(a) equipped with four interleaving units. The output program does not violate any constraints mentioned in Section 3 while using the physical registers existing in the target hardware. In the following sections, we explain the internal structure and algorithms of the RA module.

5 REGISTER ALLOCATION ALGORITHMS

5.1 Internal Structure of the Register Allocation Module

Figure 8 depicts the four-step workflow of the register allocation module. It shows the internal structure of the register allocator, which is illustrated as the grey box in Figure 7(a). The first step is program analysis (Section 5.2), which conducts data flow analysis including liveness analysis to collect information on the program that may cause constraints. The second step is to build a PBQP graph (Section 5.3), based on the information collected from the program analysis step and the configuration file written for the target architecture. Once we have converted the register allocation problem into a PBQP graph, we can get a solution by running a PBQP solver. Because a PBQP graph itself is a data structure independent from a solver, we can use any solver such as the original solver by Scholz et al. [29]. However, due to the characteristic of the PBQP graphs built from the test-pattern programs, we devised our own solver by modifying and extending the

Fig. 8. The internal structure of the register allocation module.

Table 1. The *USE* Table for Each Command Type

CMD	signal X	signal Y	signal D	description
NOP				Stay idle
ACT	O			Activate a row
WRT		O	O	Write data at col.
READ		O	O	Read data at col.
PRE	partly	partly		Precharge

heuristics of the original solver (Section 5.4). Finally, the last step is to rewrite and generate the target program if the PBQP solver succeeds in finding a valid solution of register allocation.

5.2 Program Analysis

This step analyzes the input program with virtual registers (e.g., Figure 7(b)) to collect information including register liveness information [2]. Before the analysis, we first rename all virtual registers so that every definition of a virtual register has a distinct name, thus converting the program into a kind of SSA (static single assignment) form [9].

Then, we perform the standard liveness analysis, by implementing the iterative worklist algorithm [2]. The algorithm applies the transfer function $LIVE_{in}[s] = USE[s] \cup (LIVE_{out}[s] - DEF[s])$ for each statement s , where $LIVE_{in}$ is the set of live variables right before the statement, $LIVE_{out}$ is the set of live variables right after the statement, $USE[s]$ is the set of variables used in the statement, and finally $DEF[s]$ is the set of variables that are defined in the statement.

Statements for register operations such as addition can be interpreted as in general programming languages, but there are many implicit register representations in the pattern programs that we also need to consider. In Figure 7(b), even though we explicitly denoted the *signal transmission* with the “ $\leq$ ” notation, the actual source program does *not* include those instructions. They should be implicitly determined by the *command* type from its semantics. So, the command WRT, for example, sends signals to the DUT’s Y address and D data, after *reading* the address register and the data register specified ahead of the statement. To identify those implicit registers live at each command, we create the command *USE table* as shown in Table 1, pre-defined based on the DDR4 SDRAM specification [4]. In this way, we can correctly find the liveness information even in the context of the test-pattern programs. During the parsing of the program, we also collect other information such as what virtual registers are coupled in register operations such as addition. This information is then handed over to the PBQP graph builder to prepare for the register pairing constraints.

5.3 PBQP Graph Builder

Now, based on the collected information, we build a PBQP graph. The algorithm is depicted in Algorithm 1. Its flow handles each of the four constraints mentioned in the previous section one-by-one. It first initializes the graph with all the cost vectors and matrices set to zero (line 1). Again, a cost matrix with zero costs is regarded as being disconnected. In the following lines, the builder accumulates the costs with infinity, as it checks each constraint one-by-one.

Lines 2–4 check interference. If two virtual registers interfere with each other, then the cost of the edge between them must include the infinity diagonal so that they are not allocated to the same physical register. Lines 5–8 resolve the class constraint. Mismatching classes can be simply represented with infinity being set to the corresponding entry of the vertex cost (e.g., address virtual registers have infinity values in the positions of data physical registers).

Lines 9 and 10 deals with the register pairing of operands for the *add* instructions (other two-operand instructions are handled similarly). The *pairing matrix* means the parameter plugged in by the configuration file, and it is a matrix identical to the one in Figure 4 with the “X” symbols replaced by ∞ ’s and the “O” symbols replaced by 0’s. These infinity costs will inhibit any allocation that leads to illegal register pairing.

Lines 11–17 are for the register group constraints. It linearly scans the program to find if there is more than one definition of any virtual registers within a major cycle, and then apply the infinity value to the corresponding entries of the edge cost for all possible combinations of the physical registers in the same register group.

Finally, lines 18–27 consider the def-use ordering in a major cycle. This logic is similar to the one for the third constraint in lines 11–17, except that it adds the diagonal infinity matrix to the edge cost, because this constraint inhibits two virtual registers from being allocated to the same physical register.

Unlike the original PBQP graph, our PBQP graphs for pattern programs always have either 0 or ∞ as the cost values. Originally, PBQP problems are designed to work with real numbers to minimize the output value of the cost function f , as depicted in Section 2.4. However, we only have the two extreme cost values, because ATE does not allow spills (so the cost vector and matrix do not have a spill entry). This makes the register allocation result be in either a success (with $f = 0$) or a failure (with $f = \infty$).

As a side note, one thing to mention is that we can assign real-numbered costs for *readability-oriented translation* of test-pattern programs. That is, one interesting characteristic of the test-pattern program translation is that test engineers care for the readability of the target program to compare it with the source program, to confirm that it works correctly following the test engineers’ exact intention and programming conventions. For example, registers with “X” included in their name are mostly used to represent an X address, even though there are neither physical nor syntactic rules about it. If we identify all virtual registers used for X addresses and assign lower costs to the physical registers with “X” in their names than other physical registers, then the PBQP solver would try to match the preference. In the current article, we do not address this preference issue for simplicity.

5.4 PBQP Solver

Scholz et al. proposed an efficient PBQP solver [29]. Their solver employs a heuristic based on graph reduction that removes one vertex at a time, with the calculated costs (constraints) of the removed vertex being propagated to its adjacent vertices. The graph reduction repeats until there are only trivial vertices with no neighbors left. Then, we can allocate registers in the reverse order of the removal.

ALGORITHM 1: Build PBQP Graph

Data: $insts[1..l]$, list of source instructions
Data: $live[0..l]$, list of lists (virtual regs live at line l)
Data: $def[0..l]$, list of lists (virtual regs defined at line l)
Data: $use[0..l]$, list of list (virtual regs used at line l)
Data: $virRegs[1..n]$, list of virtual registers
Data: $phyRegs[1..m]$, list of physical registers
Data: k , the number of interleaving units
Data: Pairing Matrix, a constant depending on the target configuration
Data: Diagonal Infinity Matrix, a constant

$$\begin{pmatrix} \infty & 0 & \dots & 0 \\ 0 & \infty & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \infty \end{pmatrix}$$

Result: a PBQP graph that represents all restrictions

```

1 Create an undirected graph  $G(V, E, c)$  where  $|V| = n$ ,  $|E| = m$ ,  $c$  is all set to zero;
2 for  $i \leftarrow 0$  to  $l$  do
3   for each pair  $(u, v)$  In  $live[i]$  do
4      $c(u, v) +=$  Diagonal Infinity Matrix
5 for  $i \leftarrow 1$  to  $n$  do
6   for  $j \leftarrow 1$  to  $m$  do
7     if  $virRegs[i].class \neq phyRegs[j].class$  then
8        $c(i)[j] \leftarrow \infty$ 
9 for  $(u, v)$  Of addition  $u + v$  In  $insts$  do
10    $c(u, v) +=$  Pairing Matrix;
11 for  $i \leftarrow 0$  to  $l$  Step By  $k$  do
12    $defsInMajorCycle \leftarrow$  empty list;
13   for  $j \leftarrow i$  to  $i + k - 1$  do
14      $defsInMajorCycle += def[j]$ ;
15   for  $(u, v)$  In  $defsInMajorCycle$  do
16     for  $(p, q)$  In all combinations of registers in the same register group do
17        $c(u, v)[p][q] \leftarrow \infty$ ;
18 for  $i \leftarrow 0$  to  $l$  Step By  $k$  do
19    $ordering \leftarrow$  empty list;
20    $usesInMajorCycle \leftarrow$  empty list;
21   for  $j \leftarrow i$  to  $i + k - 1$  do
22      $usesInMajorCycle += use[j]$ ;
23     for  $u$  In  $def[j]$  do
24       for  $v$  In  $usesInMajorCycle$  do
25          $ordering += (u, v)$ ;
26   for  $(u, v)$  In  $ordering$  do
27      $c(u, v) +=$  Diagonal Infinity Matrix
28 Return  $G$ ;
  
```

Fig. 9. The PBQP graph for the running example. Edge costs between DATA1 and others are omitted for simplicity.

One issue is that removing a vertex from the graph while achieving the optimal solution (global minimum of f) requires to explore the whole search space of size $O(m^d)$, where m is the number of physical registers and d is the degree (i.e., the number of adjacent vertices) of the removed vertex. When d is small (e.g., one or two), we can *enumerate* the whole search space without a big overhead. However, if the degree is more than two, then the overhead gets much higher, so we should *approximate* to remove a vertex, which might lead to a local minimum for f . Fortunately, Scholz et al. found that for most PBQP graphs of their irregular register allocation, this strategy of removing *easy* vertices first following the ascending order of the degrees, then removing *hard* vertices using an approximation when the smallest degree is more than two, leads to the global minimum.

Unfortunately, if we use the original PBQP solver to our PBQP graphs built for the test-pattern programs, then it cannot work well for two reasons. First, if we build a PBQP graph for the test-pattern program in Figure 7(b), then it will be as in Figure 9, where most of the edges do not have a zero matrix, meaning that most of the vertices tend to have a large degree. This is so because there are much more cases of irregularity than typical irregular register allocation only with register pairing and register class; they would fill many more entries of the edge cost matrix with infinity. This means that the original solver should approximate, thus easily falling into a local-minimum solution whose f is ∞ .

Second, a local-minimum solution with $f = \infty$ in ATE register allocation means a complete failure of translation. This is in sharp contrast to normal register allocation where a local-minimum allocation is acceptable, since we can still generate code using the spill code. So, if approximation used to remove a vertex in the original solver cannot find an optimal solution with $f = 0$, it is not acceptable for ATE register allocation. For example, Figure 10 is a PBQP graph where every vertex has a degree greater than two. So, the original solver approximates to remove the vertex ADDR1. In this case, allocating either A1 or A2 seems to have the same cost of zero, but if the approximation chooses A1 for ADDR1, the whole register allocation fails even though it could have succeeded if the heuristic were picking A2 instead.

Fig. 10. The PBQP graph that the original solver might fail. Assume there are some more vertices and edges omitted in the figure, so that the degree of the vertex ADDR1 is greater than 2.

To address this issue, we modify the PBQP heuristic with a completely different allocation strategy. Basically, our modified solver attempts to enumerate the search space aggressively to avoid hasty approximation that might lead to allocation failure. However, there should be a tradeoff between the enumeration time and the search space. Unlike the original solver that enumerates easy vertices first and then approximates hard vertices, we enumerate hard vertices first and then approximate easy vertices. The definitions of “hard” and “easy” are different as well. Actually, the highly restrictive register constraints of ATE make some vertices have many infinity values for its cost vector. That is, those vertices are *picky*, with only a few allocation options available, so they are particularly vulnerable to the negative effect of approximation. Also, since the number of allocation options for these picky vertices is scarce, it is worthwhile to explore all allocation options for them. These are the hard vertices for our solver. So, we define the “*liberty*” for each vertex as the number of finite entries in its vector cost (i.e., the number of allocation options for the vertex), and the search is performed in ascending order of liberty (from the hardest vertices first). The value of liberty can vary from 0 to m , and having any vertex with a liberty of 0 in the graph means the register allocation fails. For example, in the graph in Figure 10, the liberty of ADDR1, ADDR2, and ADDR3 is 2, 3, and 2, respectively.

Algorithm 2 shows the procedure of the modified heuristic. The algorithm takes two thresholds $n_t \leq n$ and $m_t \leq m$ such that it lists the vertices in an ascending order of liberty that is less than m_t , and then chooses the first n_t vertices, for which the algorithm performs full enumeration. For other vertices, we approximate as in the original solver (line 23). By adjusting these thresholds n_t and m_t in a time-affordable way, we can increase the chance of finding a valid solution. The thresholds should not be too large, because the time complexity for enumerations is $O(m_t^{n_t})$, but our empirical result shows that the required thresholds are small enough to find a solution for most cases (Section 6).

In lines 5–21, it iterates through all possible allocations in the increasing order of the liberty and the cost value. Because one allocation choice that has been explored should be skipped next time, line 10 removes the taken option. Then, lines 11–13 record the current decision and the execution

environment in case this decision leads to failure. The execution environment can be recovered later in such cases. In lines 14 and 15, each allocation decision on a vertex propagates along to all adjacent vertices to preserve the restrictions even after taking out the vertex from the graph. And, finally, in line 23, it applies the graph reduction based PBQP solver [29]. If the solver yields a valid solution, then we can add that to the previous decisions, *mapping*, which we return. Otherwise, we trace back and amend the former decisions in lines 24–26, until it ends up with a solution in lines 27–29, or eventually, the algorithm exhausts all combinations for n_t vertices in lines 18 and 19. In summary, the modified solver takes a hybrid strategy, where we make a decision by brute-force search for a small set of hard vertices and propagate it to other vertices as in the original solver.

ALGORITHM 2: The modified PBQP Heuristic

Data: $G\langle V, E, c \rangle$, a PBQP graph

Data: $\text{phyRegs}[1..m]$, list of physical registers

Data: n_t and m_t , thresholds

Result: virtual register $\Rightarrow$ physical register, or failure

```

1 mapping  $\leftarrow$  empty map;
2 history  $\leftarrow$  empty list;
3  $U \leftarrow$  list of first  $n_t$  vertices  $v \in V$  ascending sorted by the liberty  $l(v)$  where  $l(v) \leq m_t$ ;
4 level  $\leftarrow 1$ ;
5 while True do
6   if level  $< n_t$  then
7      $v \leftarrow U[\text{level}]$ ;
8      $p \leftarrow \text{argmin}(w(v))$ ;
9     if  $w(v)[p] < \infty$  then
10       $w(v).\text{remove}(p)$ ;
11      history[level] =  $G\langle V, E, c \rangle$ ;
12      mapping[v] = phyRegs[p];
13       $V.\text{remove}(v)$ ;
14      for  $(u \text{ adj. } v) \wedge (u \in V) \wedge (u \notin U[1..level])$  do
15         $w(u) += w(u, v)[p]$ ;
16      level = level + 1;
17   else
18     if level = 1 then
19       Return Allocation Failure;
20     level = level - 1;
21      $G\langle V, E, c \rangle \leftarrow \text{history}[\text{level}]$ ;
22   else
23     result  $\leftarrow$  PBQP-SOLVER ( $G$ );
24     if result = Allocation Failure then
25       level = level - 1;
26        $G\langle V, E, c \rangle \leftarrow \text{history}[\text{level}]$ ;
27     else
28       mapping += result;
29     Break;
30 Return mapping

```

Fig. 11. The tree traversal example of the register allocation for the graph in Figure 10.

Intuitively, Algorithm 2 can be viewed as a tree traversal as depicted in Figure 11. Each node of the tree is the state of the decisions. Level 0 means that no allocation decision is made. Then, the algorithm iteratively chooses a virtual register and tries to allocate a physical register available, one at a time. After making an allocation decision, the algorithm removes the vertex in the PBQP graph that corresponds to the virtual register. For each adjacent vertex, the corresponding edge cost is propagated to its vertex cost to update. Since the full tree traversal is intractably expensive, we limit both of the width and depth of the traversal. First, when choosing a virtual register to try for an allocation, we follow the ascending order of the vertex liberty. Second, when the current position is deep enough in the tree, we stop exploring down further and apply the original heuristic, which takes polynomial time. By limiting the width (m_t , the maximum liberty) and depth (n_t , the maximum level), we can control the time taken for allocation, while maximizing the chance of finding a valid solution.

As to the correctness, register allocation has to comply with all of the restriction rules of the ATE originated from their hardware design, as seen in Section 3. Since any allocation that might violate a restriction adds the infinity to the cost function, such allocation *cannot* be chosen by the PBQP solver (both the original one and our modified one).

6 EVALUATION

6.1 Experimental Setup

We evaluated our register allocator with eleven pattern programs written for an ATE, abstracted from a real-life ATE model that we will discuss in Section 6.4. It is equipped with 13 physical registers (i.e., $m = 13$) where there are two data registers and 11 address registers. It also has 8 inter-leaving units. Even though its real-life ATE model has many more registers (around 100 registers), most of them are reserved for special purposes, thus not important for this experiment, while the

Fig. 12. Elapsed time (log scale) for each configuration of solvers. Shaded bars indicate failing cases where the solver could not find a proper solution. The another y -axis is for vertex counts n of each benchmark program that is equal to the number of virtual registers used in the program.

programmer may use only a subset of data and address registers available (see Section 6.4). In fact, our input pattern programs were written only with those 13 registers, which is the reason that we experiment with an abstract ATE with 13 physical registers; more registers would make register allocation easier.

Our benchmarks include a toy pattern program (LAB1) and 10 product-level pattern programs (from PRO1 to PRO10). The programs describe some well-known patterns from the textbooks as well as the industrial patterns specific to some real DRAM products. We first perform live range renaming, so the real registers used in the programs are replaced by the virtual registers, then they are allocated to those 13 registers by our register allocator. It should be noted that we may allocate 13 registers differently from what the test engineer originally wrote for the pattern program, since our algorithm will perform its own register allocation, independent of the real registers used in the original program. The programs have about 1K lines of code on average, which generate 10 M–200 B cycles of patterns. After live range renaming, the number of virtual registers (i.e., n) ranges from the dozens up to nearly 250, making the register allocation non-trivial. We implemented the register allocator module using Python.

As Algorithm 2 states, our solver takes a threshold m_t as one of its inputs. Instead of taking another threshold n_t as well, it enumerates every node whose liberty is less than or equal to m_t for simplicity (since there are not too many such nodes). We experimented with four configurations of the *modified solver* (Algorithm 2), with its m_t ranging from 1 to 4. The *original* PBQP solver is experimented as well to provide the baseline.

When measuring the elapsed time of register allocation, we performed the experiment on a desktop computer equipped with Intel Core i7-9700K CPU @3.60 GHz, 64 GB RAM, and Ubuntu 18.04.2 LTS. The Python interpreter 3.6.9 is used to launch the programs and run most of the modules. For each configuration, we average the running time for 10 repetitive runs, after several warm-up runs.

6.2 Allocation Results

For demonstrating a typical translation scenario, we perform live range renaming for our program so that each live range has a unique virtual register number. Then we perform register allocation, targetting the same $m = 13$ registers. We found that out of the entire register allocation process, the PBQP solver takes from 66% to 91% of the total execution time. Figure 12 plots the solver runtime (log scale as seen in the left) using a bar for each configuration of solvers. Some of them failed to

find a valid solution (i.e., allocation failure), depicted by shaded bars. The line plot is to show the size of the corresponding PBQP graph, which equals to the number of renamed virtual registers n in each program (50–250 in the right).

Obviously, programs with a larger n generally take more time, because there are more vertices to traverse. Also, a larger m_t mostly leads to longer elapsed time, because m_t is the base of the worst-case time complexity. Furthermore, as m_t grows, the modified solver enumerates more vertices, meaning a larger n_t as well. The modified solver with $m_t = 4$ is from $5\times$ to $162\times$ slower than the baseline but is rewarded with more allocation successes.

As to allocation success, the modified solver with $m_t = 4$ finds valid solutions for all programs, whereas the original solver succeeds only in 2 of 11 programs. In every case, the success rate increases as m_t increases, meaning that the modified solver with $m_t = x + 1$ never fails what $m_t = x$ succeeded. That is true for any x (unless it times out due to too long running time), because the search space covered by $m_t = x + 1$ is a strict superset of the search space covered by $m_t = x$. Additionally, in the cases where the modified solver finds a valid solution with a smaller m_t , the solver time does not increase even with a higher m_t due to the deterministic implementation of the traversal order.

As to the memory footprint, we measured the size of the PBQP graphs, since it is the largest data structure in the program. We used Python's built-in data structure *dictionary* (a hash map) to implement matrices that represent the edge costs of the graphs. The biggest size of a PBQP graph was 194.2 MB when experimented with PRO10 ($n = 241$) and $m = 13$ registers. Even though the size is affordable, we expect to further reduce the size by encoding each cost value as 1-bit datum, since it is either 0 or ∞ .

Finally, the proposed method showed 100% of correctness and efficiency for the translated test, which is one of the most crucial factors from the standpoint of test engineering. To evaluate them, we compared the simulation results of the original program and the translated program based on the folded IR proposed by Kim et al. [18]. As a result, they produce the same output (correctness), which also means that their tests would complete within exactly the same time (efficiency), since the output is the stream of test signals with timestamps.

6.3 Stress Testing

Since our benchmark programs were originally written for an ATE with $m = 13$, the register allocation problem for the same ATE with $m = 13$ would not be too hard unless the solver cannot handle the irregular register constraints appropriately, as in the original PBQP solver. We performed a more challenging, stress test by repeatedly decreasing the number of registers m by one. We did this test by changing the configuration settings that are plugged into the register allocation module. Since two data registers are required at a minimum, we removed an address register each time.

Figure 13 is a plot of the elapsed time of each solver as a function of the number of physical registers, experimented with the program PRO9 where all solvers succeed in allocation for $m = 13$ (Figure 12). In the plot, "O" markers indicate a register allocation success, while "X" markers indicate a failure. The dashed vertical line at $m = 5$ shows the minimum of registers required to make the program (three address registers and two data registers).

Of 13 physical registers, 2 data registers do not affect how our solver performs, since they are required all the time. Therefore, we can think of the effective number of physical registers, as $m_e = m - 2$. When $m_t < m_e$, the number of hard vertices whose liberty is smaller than m_t increases as m decreases, since the number of infinity elements remains the same while m decreases. This leads to a longer elapsed time, because our solver enumerates for more vertices. When $m_t \geq m_e$, the solver must perform enumeration for all vertices, but the decreased m itself reduces the search

Fig. 13. Elapsed time for each configuration of solvers. The x-axis indicates the size of a randomly chosen subset of physical registers. Markers “O” and “X,” respectively, indicate whether the solver succeeds or fails to find a valid solution in each case.

space while the number of hard vertices remains the same, so the elapsed time decreases as m decreases. This is the reason that the plot in Figure 13 increases then decreases when moving from right to left.

Obviously, the allocation difficulty would increase as m decreases, since decreased m means a smaller liberty in each node and a mistaken approximation can lead to its allocation failure more easily. The search strategy of our modified solver is effective in this viewpoint, since, naturally, it enumerates more aggressively when the allocation gets more difficult. Interestingly, in the case of the solver with $m_t = 3$, the solver fails within the interval $[7, 9]$ of m . Compared to the interval of $m > 9$, the allocation is too difficult within this interval with $m_t = 3$. With $m_t = 4$, however, our solver could find a solution at the cost of a more elapsed time. However, $m_t = 3$ succeeds in References [5, 6] due to the reduced search space. So, m_t would better be decided depending on the value of m . In our experiments, $m_t = 4$ was enough to find solutions with a reasonable amount of elapsed time. It took 81.8 s at most, which seems to be acceptable considering that the programs are translated off-line and then executed many times. However, an exhaustive search would be intractable due to the exponential time complexity (e.g., asymptotically it can be as large as 200^{250}).

6.4 Case Study

While experiments above were performed for a simplified version of ATE, this section illustrates a case study with a more realistic setup. It is based on a real-world translation scenario from ATE B to ATE A in Figure 7, where the source program is written for ATE B and the target program is generated for ATE A. Both are from the same ATE manufacturer, but ATE B is a more advanced model equipped with about 200 registers, while ATE A has about 100 registers (in fact, the simplified ATE in the previous experiments is abstracted from ATE A).

Although there are many registers, about 60% of registers in both ATE A and ATE B are dedicated to special purposes, such as for setting the memory mode, the max value indicator, and the generation speed indicator, and so on. Only the remaining registers (~40 for ATE A and ~80 for ATE B) are available as address and data registers. Register allocation for the translation of ATE B to ATE A is challenging (i.e., cannot be easily done manually by the test engineer), since some registers used in the source program do not exist in the target ATE, so a naïve one-to-one mapping would not work, nor would the traditional graph coloring that cannot easily handle the irregularities of the ATE registers.

Fig. 14. An example code fragment of the source program and the target program for a realistic case study.

Other than the number of available registers, there are little differences between ATE A and ATE B. That is, the ~40 registers of ATE A is a subset of the ~80 registers of ATE B. Also, they have similar restrictions including the register groups, addition pairing, and so on. Finally, both ATEs employ the same eight interleaving ways. For this experiment, we used a real-world program PRO11 written for ATE B as a benchmark.

We run the proposed register allocation algorithm, using a realistic configuration file that depicts the register restrictions of the target ATE A. It describes the names, the characteristics, and the restrictions (categorized as in Section 3) of the ~100 registers in ATE A. Since the dedicated registers cannot be used for register allocation for the virtual registers of the program, only ~40 registers need to be described in the configuration file.

Figure 14 describes an example code fragment of the source program (ATE B program) and the translated program (ATE A program). This example only focuses on register definitions and uses, ignoring the difference of the language primitives for simplicity. Figure 14(a) shows a code fragment for the source program, which uses registers such as A1, A2, A3, B1, B2, B3, C2, and D3. However, ATE A does not have registers A3, B3, C2, and D3, which requires the translator to perform proper register allocation.

Figure 14(b) shows the result of our register allocation, which successfully avoids all invalid register uses while using only the available registers in the target ATE A. For example, in line 2, A3 cannot be replaced by A1, because line 1, which is in the same major cycle with line 2 includes a use of A1. We found that our register allocator works correctly in this realistic translation scenario, handling all restrictions presented throughout the article, and produce the correct target program, which we verified as in the previous experiments.

7 RELATED WORK

Register allocation, in general, has been essential to compilers, therefore very well-studied for decades. Traditionally the register allocation problem is considered as a graph coloring algorithm with additional techniques of coalescing and re-materialization [5, 6]. Due to fast execution time and competitive performance, to these days, linear scan based global register allocation approaches also have been widely used [26].

However, as Scholz et al. [29] points out, a more sophisticated method is needed to represent unconventional register constraints found in irregular architectures, than the graph coloring and linear scan methods. In an effort to provide such a new method, Scholz et al. [29] proposes PBQP, as well as a PBQP solver. Based on this previous work, our article shows how to represent irregular constraints found in test-pattern programs as vertex and edge costs in a PBQP graph. The constraints include new types of irregularities, particularly related to interleaving units. Furthermore, this article suggests a modified PBQP solver to better suit our register allocation

problem. Meanwhile, there are mainly three other approaches that also have the potential to correctly represent the register allocation problem in irregular architectures.

The first one is integer linear programming– (ILP) based register allocation. Many researchers have delved into register allocation using ILP [3, 13, 20]. ILP is a mathematical optimization problem for an objective function while being subject to constraints that consist of linear equations and inequalities of integer variables. They formulate the register allocation problem as an ILP problem by assigning a *decision variable* for each symbolic register (or live-range), producing linear constraints based on the generated variables and modeling the cost function of spill overhead as the objective function. Along with the PBQP-based register allocator, the ILP-based register allocator is often regarded as one of the best methods to get a better-quality code when one can afford a longer compile time, particularly for irregular architectures such as X86. Besides, a long compile time is not much of a trouble for our translation scenario, considering that it is an offline task in real-world cases. The second approach to correctly represent register allocation problems for irregular architectures is suggested by Koes et al. It converts the problem into an multiple commodity network flow (MCNF) problem [19]. In this approach, each virtual register is a commodity of which source is at its definition and its sink is at its last use. Then, the register allocation problem is to find a possible flow for every commodity when each program point is constituted of vertices, each represents one physical register or a spill. The third approach converts the problem into puzzle-solving [27]. In this approach, each physical register is considered as a part of a puzzle board, of which shape differs depending on its characteristics. Additionally, for each program point, they consider live virtual registers as puzzle pieces. Then, the register allocation problem is to solve the puzzle by perfectly fitting all puzzle pieces into the board.

Even though this article could have exploited the other three approaches mentioned above (ILP, MCNF, or puzzle solving), we opt to represent our problem into a PBQP problem, because it was considered the most natural way to represent our constraints. There are several reasons as follows.

- First, due to the characteristics of the test-pattern programs (see Section 2.3), the code quality of the allocated program is always the same regardless of the register allocator. Therefore, the choice of the approaches does not matter much in terms of the performance of the generated code.
- Then, it becomes relatively more important to ensure to find a solution without spilling when at least one exists. As shown in Algorithm 2, this article presents a way by extending the existing heuristic of the PBQP solver.
- Compared to the other approaches, PBQP provides better flexibility when it is applied to other irregularities [14]. For example, to represent the constraints introduced in Section 3 as an ILP problem, we would have had to devise transformation rules (from the control flow graph to the instruction graph) that produce corresponding constraint equations. In contrast, PBQP manages the constraints as a form of cost values. As a result, we only need to devise the algorithm such as Algorithm 1 to fill the cost vectors and matrices with the infinity values at proper positions, which is much simpler.

Meanwhile, recent studies on register allocation mainly focus on faster algorithms. Krause [21] presents polynomial-time optimal register allocation via the tree decomposition technique for structured programs. Even though test-pattern programs are also structured programs, the algorithm is inappropriate to apply to our problems, particularly with various irregularities. This is due to the algorithm only considering interference, leaving us with no possibilities to represent such irregularities. Chen et al. [7] suggests a register allocation method for Intel processor graphics, which is derived from the traditional graph coloring method to reflect the specificity of the target architecture, such as partial writes, arbitrary-sized variables, and bank conflicts issues.

However, the proposed methods are not general enough to be applied to ATE, despite their efficient application to their target architecture. Eisl et al. [12] develops the linear scan-based register allocation technique that assigns different register allocation for each trace in Just-In-Time (JIT) compilers. It allows higher quality register allocation for frequently executed paths. This work is also irrelevant, because, first, we are not compiling code in the JIT manner, and second, linear scan methods cannot reflect irregularities. Some works such as Domagala et al. [11] attempt to combine register allocation pass and instruction scheduling pass, since these two passes affect each other's results. By considering both problems at the same time rather than repeatedly conducting one pass after another, there is a chance to solve both problems more effectively. However, in test-pattern programs, the location of each instruction has its own meaning and it comprises semantics, so the instruction scheduling problem itself must be defined differently. At least, in this article, the semantics of the source program are maintained during translation, therefore, such a problem is beyond the scope of our work.

8 SUMMARY AND FUTURE WORK

There is a portability issue for test-pattern programs running on ATE, and one practical solution used by the DRAM chipmakers is direct translation. Since ATE registers are highly irregular, we need irregular register allocation during translation. In addition to conventional irregular allocation issues to handle register pairings and register classes, ATE register allocation also requires to handle irregularity caused by interleaving. We modeled all of these irregular register constraints elegantly with PBQP. Since the original PBQP solver is not working well for ATE register allocation where no spill is allowed, we developed an elaborate PBQP solver, which can find valid allocations with a reasonable amount of time. We demonstrated this with the product-level test-pattern programs.

Although our modified PBQP solver could find a solution efficiently for our ATE benchmark programs, the PBQP problem itself is a hard problem, especially when no spill is allowed as in ATE register allocation. So, we might need an even more elaborate solver, and one promising approach would be to exploit machine learning based on deep neural networks [10, 15, 17, 22]. Also, when it is really impossible to allocate registers for a given program, some changes in the source program, such as loop unrolling, might improve the chance of register by loosening the constraints related to the major cycles, without affecting the program correctness. All of these research issues are left as a future work.

REFERENCES

- [1] R. Dean Adams. 2003. *High Performance Memory Testing: Design Principles, Fault Modeling and Self-Test (Frontiers in Electronic Testing)*.
- [2] Alfred V. Aho. 2003. *Compilers: Principles, Techniques and Tools (for Anna University), 2/e*. Pearson Education India.
- [3] Andrew W. Appel and Lal George. 2001. Optimal spilling for CISC machines with few registers. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 243–253.
- [4] JEDEC Solid State Technology Association. 2012. JEDEC Standard: DDR4 SDRAM: JESD79-4A. September 2012 (2012).
- [5] Preston Briggs, Keith D. Cooper, and Linda Torczon. 1992. Rematerialization. *ACM SIGPLAN Not.* 27, 7 (1992), 311–321.
- [6] Gregory J. Chaitin. 1982. Register allocation & spilling via graph coloring. In *ACM SIGPLAN Notices*, Vol. 17. ACM, 98–105.
- [7] Wei-Yu Chen, Guei-Yuan Lueh, Pratik Ashar, Kaiyu Chen, and Buqi Cheng. 2018. Register allocation for Intel processor graphics. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO'18)*. ACM Press, New York, New York, 352–364. DOI: <https://doi.org/10.1145/3168806>
- [8] Dennis R. Conti. 2002. Mission impossible? Open architecture ATE. In *Proceedings of the International Test Conference*. IEEE, 1207.
- [9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (1991), 451–490.

- [10] Dibyendu Das, Shahid Asghar Ahmad, and Kumar Venkataramanan. 2019. Deep learning-based hybrid graph-coloring algorithm for register allocation. *arXiv:1912.03700*. Retrieved from <https://arxiv.org/abs/1912.03700>.
- [11] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and Ponuswamy Sadayappan. 2016. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the 25th International Conference on Compiler Construction*. 143–151.
- [12] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based register allocation in a JIT compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–11.
- [13] David W. Goodwin and Kent D. Wilken. 1996. Optimal and near-optimal global register allocation using 0–1 integer programming. *Softw.: Pract. Exp.* 26, 8 (1996), 929–965.
- [14] Lang Hames and Bernhard Scholz. 2006. Nearly optimal register allocation with PBQP. In *Proceedings of the Joint Modular Languages Conference*. Springer, 346–361.
- [15] Jiayi Huang, Mostofa Patwary, and Gregory Diamos. 2019. Coloring big graphs with alphagozero. *arXiv:1902.10162*. Retrieved from <https://arxiv.org/abs/1902.10162>.
- [16] IEEE Computer Society. 2006. *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language*. Vol. 1999. DOI: <https://doi.org/10.1109/IEEESTD.1999.90563>
- [17] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.
- [18] Minsu Kim, Jeong-Keun Park, Sungyeol Kim, Insu Yang, Hyunsoo Jung, and Soo-Mook Moon. 2019. Output-based intermediate representation for translation of test-pattern program. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'19)*.
- [19] David Koes and Seth Copen Goldstein. 2005. A progressive register allocator for irregular architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 269–280.
- [20] Timothy Kong and Kent D. Wilken. 1998. Precise register allocation for irregular architectures. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 297–307.
- [21] Philipp Klaus Krause. 2013. Optimal register allocation in polynomial time. In *Proceedings of the International Conference on Compiler Construction*. Springer, 1–20.
- [22] Henrique Lemos, Marcelo Prates, Pedro Avelar, and Luis Lamb. 2019. Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems. In *Proceedings of the 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI'19)*. IEEE, 879–885.
- [23] Stephen McGinty, Daniel Hadad, Chris Nappi, and Brian Caquelin. 2015. Developing a modern platform for test engineering - Introducing the Origen semiconductor developer's kit. In *Proceedings of the International Test Conference*. 1–10. DOI: <https://doi.org/10.1109/TEST.2015.7342393>
- [24] P. D. T. O'Connor. 1992. Testing semiconductor memories: Theory and practice, A.J. van de Goor, Wiley, 1991. Number of pages: 512. Price: £34.95. *Qual. Reliabil. Eng.* 8, 2 (1992), 156–157. DOI: <https://doi.org/10.1002/qre.4680080216>
- [25] Jeong-Keun Park, Minsu Kim, Sungyeol Kim, Insu Yang, Hyunsoo Jung, and Soo-Mook Moon. 2019. PaTran: Translation platform for test pattern program. In *Proceedings of the IEEE European Test Symposium (ETS'19)*. DOI: <https://doi.org/10.1109/ETS.2019.8791506>
- [26] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 895–913.
- [27] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. *ACM SIGPLAN Not.* 43, 6 (M 2008), 216. DOI: <https://doi.org/10.1145/1379022.1375609>
- [28] Jeff Rearick. 2019. The test industry roadmap: Ask the experts! In *Proceedings of the 24th IEEE European Test Symposium (ETS'19)*.
- [29] Bernhard Scholz and Erik Eckstein. 2002. Register allocation for irregular architectures. *ACM SIGPLAN Not.* 37, 7 (July 2002), 139. DOI: <https://doi.org/10.1145/566225.513854>

Received June 2020; revised September 2020; accepted September 2020