RL4ReAl: Reinforcement Learning for Register Allocation

S. VenkataKeerthy  
IIT Hyderabad  
India

Siddharth Jain  
IIT Hyderabad  
India

Anilava Kundu  
IIT Hyderabad  
India

Rohit Aggarwal  
IIT Hyderabad  
India

Albert Cohen  
Google  
France

Ramakrishna Upadrasta  
IIT Hyderabad  
India

Abstract
We aim to automate decades of research and experience in register allocation, leveraging machine learning. We tackle this problem by embedding a multi-agent reinforcement learning algorithm within LLVM, training it with the state of the art techniques. We formalize the constraints that precisely define the problem for a given instruction-set architecture, while ensuring that the generated code preserves semantic correctness. We also develop a gRPC based framework providing a modular and efficient compiler interface for training and inference. Our approach is architecture independent: we show experimental results targeting Intel x86 and ARM AArch64. Our results match or out-perform the heavily tuned, production-grade register allocators of LLVM.

CCS Concepts: • Software and its engineering → Compilers: • Computing methodologies → Multi-agent reinforcement learning; Multi-agent systems; • Mathematics of computing → Graph coloring.

Keywords: Register Allocation, Reinforcement Learning

1 Introduction
Register allocation is one of the well-studied and important compiler optimization problems. It involves assigning a finite set of registers to an unbounded set of variables. Its decision problem is reducible to graph coloring, which is one of the classical NP-Complete problems [8, 22]. Register allocation as an optimization involves additional sub-tasks, more than graph coloring itself [8]. Several formulations have been proposed that return exact, or heuristic-based solutions. Broadly, solutions are often formulated as constraint-based optimizations [34, 38], ILP [3, 5, 12, 42], PBQP [31], game-theoretic approaches [45], and are fed to a variety of solvers. In general, these approaches are known to have scalability issues. On the other hand, heuristic-based approaches have been widely used owing to their scalability: reasonable solutions for practical benchmarks in near linear time. However, developing good heuristics is highly non-trivial and requires specialized domain expertise, on compiler construction as well as on hardware architecture. Various heuristics have been proposed over the past 40 years [9, 11, 15], extending to recent times [13]. They are often fine-tuned for a particular architecture and yield non-optimal performance.

Recently, with the wide range of successes of Machine Learning (ML), ML-based approaches are being proposed to solve compiler optimization problems that have been known to be computationally expensive. These include classical optimizations like phase ordering [4, 21, 28], vectorization [25], function inlining [49], throughput prediction [40, 51]. However, the applicability and effectiveness of ML methods to compiler optimizations under hard semantic constraints remains poorly understood. Focusing on register allocation (regalloc), we identified some of the main reasons.

- Regalloc is a complex problem, composed of multiple sub-tasks, including splitting, coalescing, spilling. These sub-tasks have to be considered in addition to modeling hardware complexities.
- ML-based allocation schemes should ensure correctness: no two variables in the same live range be assigned to the same register, and the register types should be respected. Such semantic constraints should not suffer any approximation, unlike forgetful optimizations like function inlining.
- On a practical note, it is hard to integrate ML models and Reinforcement Learning (RL) algorithms in Python with compiler frameworks in C++ that are among the most complex pieces of software engineering.
Some initial attempts at addressing these challenges include Das et al. [20] proposing a partially ML based solution, Kim et al. [32] leveraging RL to reduce the search space of PBQP, and the infrastructural Compiler-Gym [18] approach to ease the RL-training process.

We propose a retargetable Reinforcement Learning (RL) approach to the Register Allocation (REAL) problem. We formulate a multi-agent hierarchical reinforcement learning optimization considering program-specific information: (1) to model the sub-tasks of register allocation like coloring, live range splitting and spilling, and (2) to encode the correctness constraints for preserving the semantics and hardware-compatible register assignments. The legality of the register allocations and assignments is preserved by imposing constraints on the action space, or outcome of each agent. As register allocation is a combinatorial problem, establishing the ground truth is hard, making RL a natural choice. It also facilitates the imposition of correctness constraints.

We leverage the LLVM infrastructure [35] to build the first end-to-end RL framework addressing the above-mentioned challenges. The interference graph of a function is extracted from the Machine Intermediate Representation (MIR) of LLVM. Instructions within each node are represented as vectors using representation learning methods. For this purpose, we propose MIR2Vec embeddings to represent MIR entities. These embeddings represent vertices of the interference graph that is traversed by RL agents. MIR2Vec embeddings are application-independent and may be used for other backend applications in the future. Finally, we propose LLVM-gRPC, a generic framework to facilitate communication between the RL model and the compiler during training and deployment. Our approach is portable: we show results on both Intel x86 and ARM AArch64.

Contributions. The following are our contributions:

- The first end-to-end application of RL for solving the register allocation problem.
- Formalizing the constraints to restrict the action space and preserve semantic correctness.
- Proposal of MIR2Vec to encode ML representations at Machine IR (MIR) level.
- Design and implementation of LLVM-gRPC for compiler integration with ML models.
- Experimental evaluation targeting x86-64 and AArch64 on SPEC CPU 06 and 17 benchmark suites.

2 Background and Mathematical Model

We formulate the register allocation problem by defining different constraints. We also give an overview of LLVM register allocators and multi-agent RL.

2.1 Register constraints

Optimizing compilers convert the source code into an Intermediate Representation (IR) where most target-independent optimizations take place. In the backend compiler, this IR is incrementally lowered to a machine-specific form. This representation in LLVM is called as Machine IR (MIR). MIR at the stage of register allocation is very close to machine instructions, as instruction selection and other low-level optimizations have already been performed. After instruction selection, certain physical registers that are mandated by the architecture are immediately assigned. For instance, x86 processors mandate the output of 32-bit division to be stored only in $eax$ and $edx$ registers.

As shown by the example in Fig. 1(a), DIV32 instruction divides the contents of $eax$ and $edx$ by $x$ and stores the result in $eax$. Such mandatory assignments including calling conventions are made. Regalloc can now be reduced to assigning physical registers to the other left-out virtual registers ($V$) while respecting the following constraints.

Type constraint. The register file ($R$) of a machine consists of a collection of registers $R^t$ belonging to different types ($t$): $R = \bigcup_t R^t$. Assigning a physical register $r$ to a virtual register $v$ of type $t$, $v^t \to r$, should satisfy the register type constraint $\chi^t(v^t) = \{v^t \to r : r \in R^t\}$. In Fig. 1(b), each virtual register is associated with a particular register type. For instance, $\%$x is of gr 32 type, which means that it belongs to a 32-bit wide general-purpose register type. Meaning, only registers belonging to that type (like $eax$) can be assigned.

Congruence constraint. Real-world instruction set architectures like x86 and AArch64 have a hierarchy of register classes. For instance, 32-bit type of registers (like $eax$, $ebx$) are physically part of the 64 bit ones (like $rax$, $rbx$). We consider the registers that adhere to this part of relation as a congruence class $C(r)$. For example, registers $%al$, $%ah$, $%ax$, $%eax$, $%rax$ of x86, which are “chunks” of the same physical register belong to the same congruence class, satisfying $%al \subseteq %ax \subseteq %eax \subseteq %rax$. So, the assignments for virtual register $v$ should be among the set of registers that satisfy the following congruence constraint $\forall r' \in C(r)$:

$$\chi^C(v^t) = \{v^t_1 \to r : \forall u_i, u_j \in V, u_i \neq u_j, \exists u_j \to r' \in L(u_i)\}$$

Here $L(v)$ corresponds to the live range of variable $v$, and is computed as $L(v) = [p^{def}_v, p^{end}_v]$; the definition of $v$ occurs at program point $p^{def}_v$, and its last use is in $p^{end}_v$. Fig. 1(a) gives an example. The live ranges of the corresponding variables are shown in Fig. 2(a).

Interference constraint. Register allocation has been modeled as a graph coloring problem [11]. For each function in the program, it involves creating an Interference graph $G(V, E)$ defined as follows: the vertices of the graph are mapped to virtual registers ($v$) or physical registers ($R_e$), meaning $V \in (V \cup R_e)$; the edges $E$ are computed as $\{(u_i, u_j) : v_i, v_j \in V \land L(u_i) \cap L(u_j)\}$. The interference graph corresponding to the example in Fig. 1 is shown in Fig. 2(b). The interference constraint says that no two adjacent nodes in
The LLVM compiler currently has four register allocators: Fast, Basic, Greedy, and PBQP, ranked according to implementation complexity. They are implemented as passes and operate on one function at a time.

Fast is an improved version of the linear scan algorithm [43] and operates at the basic block level. Basic is an improved variation of Fast, and operates at the function level [54]. Greedy was developed [29] to address the shortcomings of Basic; it iteratively combines four strategies: splitting, spilling, coalescing (merging of live ranges), eviction (de-allocating the already-allocated physical register). Each of these strategies is driven by greedy heuristics. Greedy is a complex, highly tuned, default regalloc at -O3 optimization level. It iterates over the virtual registers in multiple rounds and obtains a legal physical allocation if possible. It includes highly tuned heuristics (i) for placing of proper spill code, (ii) to minimize the load-store instructions while favoring register moves by applying strategies like node-splitting/eviction/last-chance-recoloring, etc. PBQP is the only solver based mechanism in LLVM, and models it as a Partitioned Boolean Quadratic Problem to obtain allocations [26]. These allocators are a result of significant (man-decades) amount of engineering by expert compiler writers; and are continually improved to address the regressions on a case-by-case basis. Not all allocators implement all strategies; live range splitting only takes place in Greedy, whereas coalescing is present in Greedy and PBQP, and eviction is in iterative allocators like Greedy and not PBQP.

2.4 Multi-Agent Hierarchical RL

Reinforcement learning (RL) is a branch of machine learning that often tries to solve the problems where enumerating ground truth is either hard or infeasible. The learning happens with experience where the agent learns a policy to determine the best possible action based on its observation from the environment. Depending on the goodness of action, the environment gives positive or negative rewards so as to course-correct the learning. With the evolution of DL, methods like PPO [46] that use gradient based approaches to learn better policy have become prevalent.

Depending on the problem formulation, there can be single or multiple agents to solve the problem. When the problem is modeled with multiple agents, it is called Multi-Agent Reinforcement Learning (MARL). If all the agents work together towards a common goal, learning is said to be cooperative. If they compete against each other to achieve a goal, the learning is said to be competitive. In certain cases, there can be a mix of both of these. MARL is an active field of research that has accomplished huge success in gaming [48], robotics [30], navigation [1], and autonomous driving problems [33].
We formulate regalloc as a number of smaller subproblems using multiple agents to model node selection, task selection (among splitting, spilling, or coloring), splitting, and coloring. These agents work cooperatively to achieve a beneficial register allocation. Another categorization in the case of MARL problems is based on the schedule of the agents. If the agents act on the environment sequentially, it is the sequential variant of MARL. If the agents form a hierarchy, where the top-level agent determines how the agents at the lower level should act, the learning is said to be hierarchical. In RL4ReAl, task selection, splitting, and coloring are modeled in a hierarchical fashion.

3 Modeling RL4ReAl

We formulate register allocation as a Markov Decision Process (MDP) using hierarchical reinforcement learning (RL), modeling the sub-tasks of register allocation as lower level tasks controlled by multiple agents. Fig. 3 sketches the overall approach. It involves interactions between the LLVM compiler and the RL model for both training and inference.

3.1 Environment

We implemented a new MLRegAlloc pass in LLVM, to generate an interference graph (G), allocate, split and spill registers as predicted by the agents. This pass also generates a representation of G using MIR2Vec.

3.2 Agents

The task of allocating registers is split into multiple sub-tasks. Each of these tasks are modeled as agents \( \{\omega_v, \omega_r, \omega_p, \omega_{\ell}\} \in \Omega \), that learn their respective policies \( \pi_{\omega} \) to optimally solve the low level tasks. We formulate hierarchical agents for four sub-tasks as shown in Fig. 3:

- **Node selector** \( (\omega_v) \): Top level agent that learns to pick a node \( v \in G \).
- **Task selector** \( (\omega_r) \): Mid level agent that learns to select a task among \( \{\chi, \varphi\} \) on \( v \) picked by \( \omega_v \).
- **Splitter** \( (\omega_p) \): Low level agent that learns to identify a split point \( k \) for \( v \).
- **Coloring Agent** \( (\omega_{\ell}) \): Low level agent that learns to pick a valid color \( \xi \in \chi \) or spill \( \mu \).

As it can be seen, each high level agent invokes a low level agent while following the timeline: \( \omega_v < \omega_r < \{\omega_p, \omega_{\ell}\} \). Each agent \( \omega \) has its own state space \( S_{\omega} \), action space \( A_{\omega} \), and reward \( R_{\omega} \) to learn a policy \( \pi_{\omega} \).

**Coloring Agent** \( (\omega_{\ell}) \). In case of regular architectures like x86 and AArch64, all the registers of the same type/class have equal effect on the performance. However, the standard register allocators prefer some registers over others within the same class while considering several heuristics. One of the predominantly used heuristic is to give preference to the physical registers that are not (aliases of) callee saved registers. This is achieved by deriving an ordering of physical registers that can be allocated to a virtual register while satisfying the constraints described in Sec. 2.1. The goal of this ordering is to reduce the number of register moves. Hence, we design a simple model that can act as a coloring agent to learn the beneficial color assignments.

For a graph of \( V \) nodes and a set of registers \( \chi(v) \) available at the instant, the state space of \( \omega_{\ell} \) is given as a tuple \( (|v|, |\chi(v)|, |V_{nclr}|) \), where \( V_{nclr} = V \setminus V_{ncl} \) are the nodes to be colored, \( v \) is the node that is picked by \( \omega_v \), and \( |\cdot| \) denotes its embedding. Meaning, the coloring agent uses the following information to decide the register to be assigned: the embedding of the vertex \( v \), the number of registers satisfying the constraints (see Sec. 2.1), and the number of uncolored nodes in \( G \). If no registers are available, the coloring agent marks \( v \) for spilling. Hence the legal action space of \( \omega_{\ell} \) is:

\[
A(\omega_{\ell}) = \begin{cases} 
\chi(v), & |\chi(v)| > 0 \\
\mu(v), & \text{otherwise}
\end{cases}
\]

\( \chi(\varphi) \) gives the set of legal registers for \( \varphi \) (Sec. 2.1). To improve performance, the agent should maximize the use of registers for vertices with higher spill weight. And, spill weight roughly corresponds to the importance of the node \( v \). Hence the reward for the coloring agent is given as:

\[
R(\omega_{\ell}) = \begin{cases} 
+M(\varphi), & \text{if } \chi(\varphi) \\
-M(\varphi), & \text{if } \mu(\varphi)
\end{cases}
\]

In our experiments, spill weight \( M \) is estimated using the spill costs computed by LLVM.

**Splitter** \( (\omega_p) \). Live range splitting \( \varphi(v, k) \) corresponding to a variable \( v \) involves inserting a move instruction at the split point \( k \) and creating two new live ranges \( L(\varphi') \) and \( L(\varphi'') \) in place of a single original live range \( L(\varphi) \) as explained in Sec. 2.2. Selecting an appropriate split point plays an important role in making effective spilling and coloring decisions. For instance, the Greedy in LLVM greedily selects the split points so as to carve out a region that can be allocated a register, while the other parts are spilled. In our model, the splitting agent predicts the split point in the live range among all the points of use.

Inserting the move instructions can be seen as a dataflow problem, that is analogous to the phi (copy) placement while creating (going out-of-) SSA form. We use dominance frontiers to place the move instructions appropriately to preserve the correctness. In Algorithm 1, we show how the move instructions are inserted. This algorithm directly builds from earlier works [8, 10, 24] and dominance property [19], so its soundness can easily be proved.

For predicting where to split the live range of a variable \( v \), the node splitter considers the spill weights at each use of the variable \( M(\varphi) = \{M(\omega_v) : \forall k \in K\} \), the distances between each successive use \( D_v = \{D(\omega_{\ell}(v, u_{v_i})), \forall i, i + 1 \in K\} \), and the embedding \( |v| \) of \( v \). The use distance is the number of program points between two uses of \( v \). Hence, the state
We set the tuple: \( \text{time}, \text{spill weight} \). Hence the state space is formulated as a tuple of available registers, the number of interferences, its lifetime, spill weight. We define \( k \) as a hyper-parameter. We set \( k = 2 \) (1 definition and 1 use) in our experiments. We model the reward for this agent based on the outcome of the low level tasks.

\[
R(\omega_p) = M(\omega) - \sum_{i \in \{p', \omega''\}} M(i)
\]

**Task Selector (\( \omega_r \)).** For selecting a task (\( \tau \)) among coloring and splitting, the agent \( \omega_r \) considers the parameters specific to each of the tasks: the representation of \( v \), the number of available registers, the number of interferences, its lifetime, spill weight. Hence the state space is formulated as the tuple: \( \langle v, |\chi(\omega)|, \delta(\omega), |K(\omega)|, M(\omega) \rangle \). The action space of \( \omega_r \) is defined as:

\[
A(\omega_r) = \begin{cases} 
\{\varphi, \chi\}, & |K(\omega)| \geq k \\
\chi, & \text{otherwise}
\end{cases}
\]

Here \( |K(\omega)| \geq k \) indicates that \( v \) should have at least \( k \) uses to be considered for splitting. We define \( k \) as a hyper-parameter. We set \( k = 2 \) (1 definition and 1 use) in our experiments. We defer from giving a reward as the goodness of the outcome is not known till a coloring decision is made.

**Node Selector (\( \omega_n \)).** It is well known that the order of picking a variable for allocation would highly affect the final outcome of register allocation. Usually, the iterative allocators use a priority queue to determine the order of allocation. The priority is derived from different heuristics including the spill cost of the variables, size of the live ranges, among others. We use a model \( \omega_n \) to figure out this order of allocation by predicting the variable/vertex to process (split/color) at every step of allocation after processing the previous vertex.

The state space of \( \omega_n \) comprises the embedding of each vertex in \( G \) obtained from a Gated Graph Neural Network (GGNN) as explained in Sec. 4. Along with these embeddings, the agent uses the spill weights of the nodes \( M \) to characterize the state. Hence, the state space is given as a tuple \( \langle |G|, M(\omega) \rangle \). Its legal action space is \( A(\omega_n) = V_{\text{new}}. \) The learned policy is deemed good based on the final coloring decision of the node. Hence, the reward for this agent is also modeled based on the rewards of the coloring agent \( \omega_c \):

\[
R(\omega_n) = \begin{cases} 
R(\omega_c), & \tau = \chi \\
0, & \tau = \varphi
\end{cases}
\]

**Global rewards.** In addition to providing rewards to the agents at each step, we derive a global reward based on the
throughput of the generated function estimated by LLVM-MCA [37]. The global reward is computed based on the difference between the throughputs of the code generated by RL4REAL ($Th_{RL4REAL}$) and Greedy ($Th_{Greedy}$) as:

$$R_G = \begin{cases} +10, & Th_{RL4REAL} \geq Th_{Greedy} \\ -10, & \text{Otherwise} \end{cases}$$

This way of using throughput helps capturing the overall impact of the allocation scheme in comparison with Greedy.

4 Representing Interference Graphs

We represent nodes of the interference graph as embeddings obtained from LLVM’s MIR instructions. Such embeddings form the input to a Gated Graph Neural Network (GGNN) that learns to generate the representation of the state space.

Any deep learning model can accept only a numerical representation as input. When it comes to applying ML techniques on programs, there are two possible ways: (i) feature based representations [21], or (ii) distributed representation-based embeddings [6, 50, 53].

It is widely understood that program embedding techniques at IR-level automatically capture the semantic information that may be difficult to recover with only syntactic embeddings [53]. Further, the IR-based program embeddings generalize better across applications, while effectively requiring much less data to train.

We propose MIR2Vec, a learned embedding model for representing the MIR form of the program. The learned MIR2Vec representations are in the form of $n$-dimensional real-valued vectors, which can be passed to the model for learning a downstream optimization task like register allocation. The embeddings can be seen as the key means for facilitating the current optimization problem (viz. regalloc), but also a necessary means in which other backend problems (viz. instruction scheduling) can be easily modeled to obtain a (representation) learning based infrastructure for backend optimizations. We generate MIR2Vec representations by: creating triplets by forming relations between entities, training TransE [7] to obtain the seed embedding vocabulary, and using it to create instruction-level representations as shown in Fig. 3. As MIR is target-specific, the embeddings are also specific to the architecture.

MIR Entities. Opcodes and MIR instruction arguments form the entities. Arguments primarily include physical and virtual registers, and immediate values. We abstract these arguments with generic identifiers as a preprocessing step.

We create two different relations. (i) NextInst: Captures the relation between the current opcode and the next instruction opcode, (ii) Arg: Captures the relation between the opcode and the arguments of the instruction. Once the triplets are generated, we train the TransE model to obtain the embeddings for each of the entities.

Grouping of opcodes. MIR contains specialized opcodes, in terms of the operating width among other factors. MIR contains about 15.3K different possible opcodes in x86 and about 5.4K in AArch64. Obtaining a dataset to cover all such specialized operands would be highly infeasible, and in turn, would not generate good representations. Hence we mask out the opcodes based on their operating width, the source and destination locations (immediate, register, and memory) and group them together.

For example in x86, there are about 200 different MOV instructions operating on different bit width, sources, and destinations, like MOV32r0, MOVZX64rr16, MOVAPDrrr, etc. All such opcodes are grouped together as a generic MOV token while forming the triplets. The obtained triplets are fed to the TransE model to generate the embeddings for each entity, resulting in seed embedding vocabulary.

Representing instructions. For a given MIR instruction with opcode $O$ and $n$ arguments $A_1, A_2, \ldots, A_n$, its representation is computed as

$$W_o \cdot [O] + W_a \cdot ([A_1] + [A_2] + \cdots + [A_n]), W_o > W_a$$

where $W_o, W_a \in (0, 1)$ correspond to the weights of opcodes and arguments, $[\cdot]$ denotes the embedding of the entity from seed-embedding vocabulary, and operators $\cdot$ and $+$ denote multiplication and vector addition respectively.

Interference Graphs. As mentioned earlier, the MIR at the stage of register allocation contains partial physical register assignments and virtual registers. The physical registers are assigned for the instruction operands that have restrictions on the particular register to be used. Virtual registers are used in all other places. Consequently, we need to take into account the edges corresponding to both virtual and physical registers. Virtual registers are marked with the register class so that assignments can only be one among the physical registers in that class.

For computing $G$, considering the interferences between the (physical $\leftrightarrow$ virtual) and (virtual $\leftrightarrow$ virtual) registers is sufficient as $G$ is bidirectional and we do not need to worry about the physical registers that are already assigned.

We use a collection of instructions in the live-range of a variable to represent a vertex of the interference graph. Each instruction is represented in $\mathbb{R}^n$ using MIR2Vec embeddings. Consequently, a vertex $v$ is represented as a matrix of embeddings $[\cdot]_v$ in $\mathbb{R}^{m \times n}$, where $m$ denotes the number of instructions in its live range.

Gated Graph Neural Networks (GGNNs) are widely used in programming language modeling [17, 41] especially when the inputs are modelled as graphs. GGNNs involve message passing between the nodes of the graph. Information propagates across multiple nodes to arrive at the representation for a given node. Also, they allow annotating the nodes and edges based on their types and properties, and consider these while learning a representation. We use GGNNs to process
the embedded interference graph to get the final representation. This network transforms $\mathbb{R}^{m \times n} \rightarrow \mathbb{R}^d$. We set $d = n$ in our experiments and annotate the graph with the following node types, along with spill weights.

- **Not visited** — nodes that are not visited yet.
- **Spill** — nodes that are marked as spill.
- **Colored** — nodes that are assigned a register.

Such node representations are propagated through a GGNN by means of message passing. Messages received from adjacent nodes are aggregated and passed through a Gated Recurrent Unit [14] to yield a final representation.

5 Compiler Integration

The integration of DL/RL models into optimizing compilers can be a hurdle, for both research and practical deployment. Problems like vectorization [25] or phase ordering [28] are adeptly controlled through optimization flags, carrying information from the predictions of the model to the compiler. Register allocation however poses an inherent difficulty as the ML-decisions and the optimization algorithm are deeply intertwined; the model predicts the final allocation based on the compiler generated code from its splitting decisions. One naive approach is to rely on Python bindings for integration; however, this involves large overhead.

We propose LLVM-gRPC, a novel infrastructure for efficient communication between the Python model and C++ compiler to support both training and inference. LLVM-gRPC involves gRPC calls [23] with the LLVM toolchain, leveraging its modular structure as an LLVM library. This gives the end-user the option of designing custom RPC calls that can operate on any of the module, basic-block, loop or function of the input program. LLVM-gRPC allows bi-directional communication between ML models and the compiler, during both training and inference. To our knowledge, this facility is not available in other frameworks.

The splitting decision by the model is communicated to the compiler via LLVM-gRPC, which then applies it and responds back with the update containing new interferences and live ranges. The model then updates the interference graph using the received information and continues the traversal. After processing all vertices of G, all the coloring decisions are communicated to the compiler as a color map.

### 6 Experimental Evaluation

We first discuss the experimental setup, followed by a characterization of the benchmarks. Then, we report the results on x86, followed by the results on AArch64 and an explanation of the results. Finally, we report improvements on the regression cases using policy improvements.

6.1 Setup

For training MIR2Vec representations, we randomly select 2K source files from SPEC CPU 2017 benchmarks and C++ Boost library. MIR triplets are generated by applying -O3 optimization flag. The seed embedding vocabulary is obtained by training a TransE model [7] on generated triplets, by running an SGD optimizer over 1000 epochs to obtain an embedding vector of 100 dimensions. We obtain 1 Billion MIR triplets from which {675, 315} entities and {25, 17} relations are generated for {x86, AArch64} respectively.

We target a complex x86 (Intel Xeon SkyLake W2133, 6 cores, 32GB RAM), and a simpler mobile AArch64 (ARM Cortex A72, 2 cores, 8GB RAM) processors. We consider allocations of general purpose, vector, floating point registers for both x86 and AArch64 (listed in Tab. 1); other registers like eflags are pre-assigned before any regalloc.

Our framework is implemented as a pass MLRegAlloc in LLVM 10.0.1, using gRPC v1.34. We train the RL models using the PPO policy with the standard set of hyperparameters on the training set of functions selected from SPEC CPU 2017, until convergence of reward graph. There were about 11K and 30K functions (with 120–500 vertices) in SPEC CPU 2017 benchmark suite in x86 and AArch64 respectively, out of which we choose 5K functions at random (from SPEC 2017) for training. Training was done on a 32GB Tesla V100 GPU and sampling was done using 12 threads of a server with Intel Xeon Platinum 8168 processors.

**Model Architecture.** Each agent learns a policy depending on its model architecture. For GGNN, we use two fully connected (FC) layers to normalize the input, followed by a RNN layer for message passing. For the agents, we use simple neural networks with FC layers: node selector and splitter use four FC layers each, while task selector and coloring agents use three FC layers each with batch normalization. Everywhere, ReLU is used as the activation function.

**Benchmarks.** In our experiments on x86, we consider C/C++ benchmarks with less than 1 MLOC (1 Million Lines of Code) from SPEC CPU 2006 and 2017. This constitutes (8 Int + 5 FP) 13 benchmarks in SPEC CPU 2006, and (8 Int + 5 FP) 13 benchmarks in SPEC CPU 2017 benchmark suites. We were able to successfully compile the benchmarks listed in Tab. 3. We observed 4 compilation errors, and 4 runtime errors due to the engineering and integration issues.

6.2 Characterization of Benchmarks

Let us now characterize the SPEC CPU 2017 benchmarks for choosing the graphs for experimentation. We study the 45 hot functions (profiled with the perf tool [36]) that take at least 5% of the total execution time of a benchmark. The
number of vertices in the interference graphs of these hot functions along with the number of interferences and register pressure is shown in Fig. 4. We compute register pressure as the maximum number of overlapping live ranges across all program points of a function.

It can be observed that out of 45 hot functions, 44 have more than 120 vertices in their interference graphs, while 31 (the majority) of them have between 120 and 500 vertices. It can also be seen that the graph size has a strong correlation with the number of interferences and the register pressure. Meaning, the vertices and edges show linear correlation (not quadratic), and the vertices and register-pressure also show near linear correlation (not quadratic) \[8\].

As the number of vertices and the register-pressure are positively correlated with each other, in general the graphs with higher number of vertices are harder to allocate. Hence, we consider the functions that have at least 120 vertices for allocation through RL4ReAl. We limit the maximum number of vertices to be 500, as most of the hot functions are in this range (120–500), and also for ease of training. Functions that are not in this range are processed using Greedy.

### 6.3 Runtimes on x86

We study the SPEC CPU 2017&2006 benchmarks and compare the results with LLVM’s allocators. Basic, Greedy and PBQP generally outperform the Fast allocator. However, there is no single allocator among these three that perform the best for all programs. Hence, we compare our results with these three allocators.

In Tab. 3, we show the runtimes obtained by Basic and the improvements obtained over it by other allocators for each benchmark. These are obtained by taking the median of three executions. Positive (negative) numbers indicate speedups (slow-downs) over Basic. For RL4ReAl, we train two models: one trained only using local rewards (L) and another with local along with the global reward (G).

On average, RL4ReAl-L (RL4ReAl-G) yields about 19s (17s) improvement over Basic; Greedy results in an improvement of about 22s. RL4ReAl-L (RL4ReAl-G) shows speedup over Basic in 14 (11) out of 18 benchmarks. The highest and second highest improvements in runtimes over Basic are highlighted in Tab. 3. In particular, RL4ReAl, (L or G) results in highest or second highest improvements over Basic in 17 out of 18 benchmarks. As it can be seen, the runtimes obtained by our framework are very close to Greedy. In comparison, RL4ReAl-L (RL4ReAl-G) results in an improvement on 5 (6) benchmarks over Greedy. And those with slow-downs, runtimes of 12 (11) benchmarks are within 1% of Greedy, and only 1 show more than 4% slow-down.

To obtain confidence intervals, we ran benchmarks 8 times and observed that the noise was under 1% consistently across all benchmarks for a 95% confidence interval, except for 1libquantum, where the noise was 1.8% (1.2%) in RL4ReAl-L (RL4ReAl-G). We also have empirical results on numerical kernels. On PolyBench \[44\] benchmark, our results show similar performance: RL4ReAl obtains an average runtime of 43.5s (3.6s) in comparison 43.65s (3.6s) obtained by Greedy on Extra-Large (Large) input size.

### Analysis of Hot functions

We did a study at function level, focusing on the hot functions. There were 35 and 31 allocated hot functions in SPEC 2006 and 2017. In Tab. 2, we...
show the percentage difference in runtime improvements obtained by Greedy and RL4ReAl allocators in comparison to Basic. It can be seen that RL4ReAl results in improvements on largest number of functions, and minimum number of slow-downs over Basic.

On average, in SPEC 2017 RL4ReAl-L improves over Basic by 7%, while Greedy results in a similar improvement of about 6%. When it comes to SPEC 2006, to our surprise, Greedy did not show improvement on average runtime among the hot functions. It is mainly due to a 51.3% slow-down observed on B22_compressBlock from Bzip2 benchmark. In comparison, RL4ReAl-G results in a lesser slow-down of about 5% on this function.

In imagick benchmark, RL4ReAl (both L and G) obtains improvements over Greedy, on all the three allocated hot functions: GetVirtualPixelsFromNexus and SetPixelCacheNexusPixels functions, RL4ReAl improves by over 29% and 22%, whereas Greedy results in an improvement of about 8% and 5%; on MeanShiftImage function, Greedy and RL4ReAl show a similar improvement of about 44%. We list the top 5 hot functions that show highest % speedup and % slow-down in comparison to Greedy from both SPEC 2006 and 2017 in Tab. 4.

### 6.4 Runtimes on AArch64

In this section, we study the performance of SPEC CPU 2017&2006 benchmarks on AArch64. As mentioned earlier, the runtimes shown correspond to the median of three runs. For the cases where there were significant differences in runtimes (about 20s) across different runs, we take a median of five runs. We cross-compile the binaries from x86 targeting the AArch64 board by running the inference. We skipped the benchmarks like perlbench, h264ref, xalancbmk and sphinx3, as they are known to have cross-compilation issues, or they fail on compilation/execution with the standard register allocators of LLVM. The runtimes obtained by Basic, the improvements obtained over it by other allocators and RL4ReAl-G are listed in Tab. 5. On average, RL4ReAl achieves an improvement of 18s, whereas Greedy achieves an improvement of about 19s. Also, RL4ReAl achieves highest or second highest improvements in all the benchmarks except four. These results demonstrate that the learned heuristics from our model work well on different architectures.

### 6.5 Policy Improvement on Regression Cases

In traditional compilers, heuristic tuning for optimization is an iterative process: human experts identify regression cases, and heuristics are tuned to identify cases of regression. In this section, we attempt to tune the learned policy to evaluate if the regression cases can be improved. For this purpose, we identified poorly performing benchmarks from each configuration: mlc on RL4ReAl-L, hmmer and xz on RL4ReAl-G. The learned model is then retrained (fine-tuned) on the hot functions of these benchmarks. Upon training, we observe a positive improvement on all three regression cases.
and $39$ on mlc, $10$s and $6$s on mmmer and xz benchmarks. This experiment makes a strong case for online or continuous learning [2], where the learning continues during deployment for betterment of policy.

6.6 Discussion
We demonstrate performance results on par with the best allocators currently available in LLVM: RL4ReAl is most frequently the best or second best allocator, and there is no single allocator that performs best across all benchmarks (see Tables 3 and 5). It is well known that register allocation is one of the hard compiler optimization problems, and the baseline heuristics achieve excellent results that cannot be easily improved upon in terms of wall-clock time. For example, the studies of Pereira et al. [45], Shin et al. [47], Kim et al. [32], and the report on the PBQP solver [16] were not able to significantly outperform baseline heuristics across benchmarks, and they report performance numbers in the same ballpark as the ones that we obtain.

In this work, we consider the major sub-tasks/strategies of register allocation: coloring, splitting and spilling, with a focus on building the first end-to-end RL model integrated with LLVM. As mentioned earlier, Greedy also admits register coalescing as one of its strategies. We consider coalescing along with other possible strategies like multi-allocation, register packing, spilling to vector registers, as possible incremental extensions for a future work. These additional strategies could result in further runtime improvements. It could however be noted that even without admitting these additional strategies—and just relying on the ones that are available in LLVM—RL4ReAl gives competitive numbers vs. the state-of-the-art regallocs in LLVM.

It can be noted that these results have been obtained fully automatically, against production-grade allocators tuned over many man-decades of experience and effort.

7 Related Work
Recently, several ML-supported compilers were proposed, leveraging representation learning techniques for compiler optimizations [25, 28, 39, 40]. These works use learned embeddings like inst2vec [6], IR2Vec [53], Flow2Vec [50] for representing the input programs to the ML model. We model a complex register allocation problem using RL and propose MIR2Vec to represent programs in MIR form.

An initial attempt to solve regalloc using ML models by Das et al. [20] uses an LSTM to come up with an initial coloring scheme; it undergoes a correction phase to rectify the inconsistency in coloring interferences. Their work focuses on the graph coloring problem, and to our understanding, the solution was not integrated to obtain the final register assignments. Another recent work by Kim et al. [32] proposes an RL-based solution inspired from AlphaZero [27] for solving PBQP constraints by reducing the search space; they use Monte Carlo Decision Trees to simulate solving the PBQP graphs. Their model focuses on an irregular and custom architecture for Automated Test Equipments. RL4ReAl is the first end-to-end application of RL for solving the generic regalloc problem; does not need a separate correction phase, and is integrated as a MLRegAlloc pass in LLVM, and reports results that are comparable to the regallocs in LLVM.

Compiler-Gym [18] is a recent approach designed to leverage Python libraries for solving compiler optimization problems; it exposes RL environments and datasets for training. However it currently does not support integrating these trained models in the compiler pipeline for both training or deployment. Another framework, MLGO [52] integrates trained ML/RL models within the LLVM compiler. For this purpose, the compiler loads a trained model and accesses it via C++ APIs of Tensorflow or ahead-of-time generated code (release mode). The framework is used in production, with improved decisions for inlining for size, and live-range eviction (in regalloc) when compared to the compiler’s default heuristics. MLGO addresses a narrower space of the register allocation problem, but its deep integration of a precompiled ML model into LLVM hints at a path for further integration of deployment of our approach in a production compiler.

8 Conclusion
We propose a target-independent Reinforcement Learning approach to the Register Allocation problem. We use a multi-agent hierarchical algorithm to learn a policy for three of the main sub-tasks of register allocation, including coloring, live range splitting, and spilling. Semantic correctness is ensured by the constraints encoded as the action masks for the agents. Our method often exhibits better allocations and generally perform on-par with the standard register allocators of LLVM. RL4ReAl opens up new opportunities for research on regalloc and on other backend compilation problems. Source code and the related artifacts are available in https://compilers.cse.iit.ac.in/research/rl4real.

Acknowledgments
We are grateful to Govindarajan Ramaswamy, Dibyendu Das, Yundi Qian, Mircea Troffin, Yanqi Zhou and Nilesh Shah for valuable discussions and feedback. We would like to thank Utpal Bora, Sayan Dey, Shikhar Jain, Soumya Banerjee and Raj Ambekar for their fruitful comments and help in validating and improving the artifacts of this work. We acknowledge National Supercomputing Mission (NSM) for providing PARAM Seva computing resources at IIT Hyderabad and HPCE, IIT Madras. This work is partially funded by a Google PhD fellowship, an NSM research grant (MeiTy/R&D/HPC/2(1)/2014), and a faculty research grant from AMD.
References


