Software-defined Software: A Perspective of Machine Learning-based Software Production

Rubao Lee, Hao Wang, and Xiaodong Zhang
Department of Computer Science and Engineering, The Ohio State University
{liru, wangha, zhang}@cse.ohio-state.edu

Abstract—As the Moore’s Law is ending, and continuous and increasingly high demand of software development in the human society, we are facing two serious challenges in the computing field. First, the general-purpose computing ecosystem that has been developed for more than 50 years will have to be changed by including many diverse devices for various specialties in high performance. Second, human-based software development is not sustainable to respond the requests from all the fields in the society. We envision that we will enter a time of developing high quality software by machines, and we name this as Software-defined Software (SDS). In this paper, we will elaborate our vision, the goals and its roadmap.

I. INTRODUCTION: TWO CRISIS OF FUTURE SOFTWARE PRODUCTION

Software production (i.e. programming) has started after the birth of modern electronic computers, both of which have created a large and powerful CPU-based computing ecosystem. However, the real driving force behind this ecosystem, the Moore’s Law [1] along with the Dennards Scaling Law [2], are ending due to the physical limit [3]. We are entering an era with a dramatic change to evolve into a new computing ecosystem where a variety of highly parallel, highly customized hardware accelerators co-exist with general-purpose processors, such as vector instruction hardware (e.g., Intel AVX) and NVIDIA GPUs. Unfortunately, the traditional programming approach in the CPU-based ecosystem are facing two fundamental challenges in the post Moore’s law era.

- The crisis of hardware heterogeneity: Developing software for high performance in computing and data processing on advanced hardware require increasingly sophisticated programming and optimization efforts in order to deliver highly optimized code, which is based on a deep human understanding of both underlying hardware architecture and application domain knowledge.

- The crisis of human resources: The human-based programming approach is not sustainable due to the shrinking of the talented software developers’ pool. Highly skilled programmers are rare and hard to be massively educated and trained. A parallel-programming expert typically needs 5-7 years of Ph.D. training, which is individual-based in low production, and cannot be in batch mode for a high production. In other words, one expert’s knowledge, skills and experiences cannot be automatically inherited by others.

The problems of low performance in machine execution and low production of software development in human resources have been hidden in the CPU-dominated computing era for two reasons. First, the Moore’s Law can automatically improve the execution performance by increasing the number of transistors in CPU chips to enhance the capabilities of on-chip caches and computing power. Thus, execution performance continues to be improved without major software modifications. Second, the development of CPU-dominated computing ecosystem for many years has created multilayer abstractions in a deep software stack, e.g. from IAS, to LLVM, to JAVA/C, and to JavaScript/Python. This software stack promotes software productivity by connecting programmers at different layers to co-work together.

However, in the past Moore’s Law era, the accelerator-oriented computing environment cannot afford such a multilayer software stack, which instead requires sophisticated programming to directly interact with hardware devices. Although recent research efforts have been made on developing an intermediate representation (IR) for code generation, such as Weld [4], we strongly argue to fundamentally address the two above mentioned crises by making the machines to develop software automatically. In this vision paper, we call this approach as Software-Defined Software (SDS).

II. THE SDS APPROACH

Motivated by recent advances of machine learning, which show its significant capabilities in solving image/voice/video recognition, natural language processing, recommendation systems, and etc, we propose the SDS approach in the programming area. The machine learning-based approach has shown promising problem cases that could only be solved by humans in the past. Furthermore, regarding playing the ancient Chinese Go game, which is believed that top professional players are usually human genius, the recent invention of DeepMind Alpha Go program [5] and the ultimate Alpha Zero program [6], has shown that machines can not only beat humans, but also given a strong evidence that self-learning machines can reinvent and enhance human experts knowledge by a huge speedup (from thousand years to a few days). Thus, we ask a question: Can a machine do the job of software production, even better than humans?

In this vision paper, we boldly believe the answer is YES. On one hand, the machine learning research domains have accumulated intensive experiences of solving various difficult problems defined by humans, with a number of new learning frameworks and models, such as Deep Learning [7]. Deep
Reinforcement Learning [8], and Transfer Learning [9], [10]. On the other hand, recent automatic programming research work, including DeepMind’s Neural Programmer-Interpreters [11] and Microsoft’s DeepCoder [12], have opened the door of using neural networks to generate computer programs, although it is still at the very primordial stage.

A. The Core of SDS: Automatic Programming by Deep Neural Networks

Like traditional software development by humans, the SDS approach also needs consider three factors: requirement, environment, and deployment. First, what are the software requirements (e.g., functionality, performance)? Second, under what environment will be the software developed (e.g., hardware platforms, software layers, programming languages)? Finally, how to deploy the software (e.g., integrating with other tools, end users, input/output)? Therefore, to make a machine write a program, the user has to tell the machine all the three factors. However, the telling can be either in traditional software requirement descriptions (e.g., UML) or by natural language interactions, considering the significant capability of AI for natural language processing and speech recognition.

The core of SDS is automatic programming for a given requirement and environment. We define three levels of automatic programming using the following example. Considering to sort four numbers \([4,2,3,1]\) into the ascending order \([1,2,3,4]\) using the C programming language. We here do not focus on how the machine understands the requirements. Instead, we focus on how the machine will finish this specific programming task.

**Level 1: Programming with Building Blocks:** The machine can assemble a program on the basis of a set of available standard library functions or primitives. In this case, the machine understands that the task is to sort an array. Therefore, it decides to choose the standard C library `qsort` function, and write the following program. We here ignore the compare function for `qsort`.

```c
int numbers = [4,3,2,1];
qsort(numbers, 4, sizeof(int), a_cmp_func);
```

**Level 2: Programming from Scratch:** The machine can write its own library function or building blocks without relying on an available function pool. In this case, the machine understands that the requirements can be divided into two steps. First, it should write a sort function. Then it should use the sort function to sort the input numbers. The machine will determine which sort algorithm should use, or a human hint is given to specify the sort algorithm. In this way, the machine will generate the code as follows.

```c
void machine_sort(int* input)
{
...
}
```

```c
int numbers = [4,3,2,1];
```

**Level 3: Application-Specific Programming:** The machine can write a very specific program for the requirements. For this sort example, there is no need to first create a general-purpose sort function and use it to process the input data. The machine can exploit its neural network models to execute the sort, in a way that cannot be explicitly expressed as a sort program. By removing the function calls caused by an independent sort function, the machine can generate specific code directly serving the application.

B. Human-defined Software vs Software-defined Software

The major differences between Human-defined Software (HDS) and Software-defined Software (SDS) are not only the quality of generated software, but the changes in the fundamental way of how software should be composed and utilized. We summarize three dramatic differences as follows.

1) **Difference 1: Write a Program vs Be the Program:** Although a person can write a program, he/she cannot be the program. Even if the human knows exactly how the sort program works, he/she cannot do the job of the sort program. However, a program that can write a program can make its own copy as the program it writes. This unique advantage makes it possible to achieve quickly re-programming and dynamic optimization, which are thought as challenging programming issues for human developers due to the long-latency and iterative programming based on observations and feedbacks from program executions.

2) **Difference 2: Explicit Algorithm Design vs Implicit Algorithm Design:** A key feature of a machine-generated program is that it may not use explicit and specific algorithms to execute a task. Instead, the program is based on a combination of multiple neural network models that are trained for mixed functionalities. This opens the door of self-growing software by re-training the underlying Deep Neural Network (DNN) models instead of current reduction-based software structure. A recent example of using a neural network model to replacing a general-purpose B-tree-like index is proposed in [13], which exploits implicit indexing on top of the underlying datasets.

3) **Difference 3: Pre-programming vs On-demand Field Programming:** Compared to the nature of pre-programming by human software development, another significant capability of SDS is that it makes field programming possible in an on-demand way. For example, a self-driving car could meet an unexpected situation that is out of the scope of any pre-defined rules or algorithms. For human-defined software, any exception handling mechanism can only handle well-defined exceptions. However, unexpected exceptions or runtime optimization opportunities can only be handled by software automatically and timely.

III. The SDS Goal: What Machines Can Do for Humans

We envision the SDS approach will pave the way for machines to execute software production in the following three ways, namely programming, optimization, and debugging.
each way, the machine can either be a totally independent worker or just an assistant to human software developers.

A. Programming

Programming is the core task of software production. According to the capability levels we defined in the above section, a set of programming work is highly possible that machines can do for humans. According to the three levels we illustrated in the above section, we present typical scenarios of each level’s capability.

The Level 1 programming is best suitable for two jobs. One is for application-level script programming that focuses on assembling a set of available script commands or well-defined primitives to satisfy a special application, such as Linux Bash Programming or HTML/JavaScript/CSS programming. Another workplace is for back-end compiler optimizations for code generations, which can work either at the level of LLVM or the underlying instruction sets. The commonplace of these two cases is that their core task is to search a solution based on the combinational possibilities on a set of basic instructions, which has been successfully proven that a machine can do much better than humans, as shown by Alpha Go.

The Level 2 programming is beyond assembling in a way that it can not only pick up existing building blocks (e.g., an instruction or a library function) but also creates new ones adaptively according to the programming requirements. We believe that the most possible and useful scenarios for this level programming is to write library functions or primitives for a new programming language/framework based on self-learning from both implementation examples from other language/frameworks or a set of programming primitives to satisfy a special application, such as Linux Bash Programming or HTML/JavaScript/CSS programming. Another workplace is for back-end compiler optimizations for code generations, which can work either at the level of LLVM or the underlying instruction sets. The commonplace of these two cases is that their core task is to search a solution based on the combinational possibilities on a set of basic instructions, which has been successfully proven that a machine can do much better than humans, as shown by Alpha Go.

The Level 3 programming is best suitable for applications that have well-defined behaviors but do not imply a clear way of how to do it. Typical AI-related applications, such as image/voice recognition, natural language processing, and language translations. In this case, automatic programming provides a possibility of end-to-end programming that directly serve the end-user requirements without a clear reduction of how each step is implemented. Specifically, when the software requirements are based on multiple functionalities (e.g., both voice recognition and voice synthesis), a Level 3 programming capability may provide the only solution that combines separate DNNs for the final requirements.

B. Optimization

Another useful possibility for automatic programming is that it can be used to optimize human-defined software. We think there are two possibilities for the machine-generated optimizations. The first one is automatic parallelization for a given sequential program written by a human programmer. For example, a human can simply write a quick sort program, which can be further automatically transformed into parallel programs executed on a variety of parallel hardware, such as Intel AVX instructions, GPU, or clusters. The second one is automatic optimization for a given architecture-independent program that can be optimized and re-implemented to be an architecture-dependent program with the considerations of all possible optimization opportunities, such as exploiting locality, prefetching, and utilizing hardware devices.

Optimization can also happen in a dynamic runtime way considering the feasibility of software-defined software that can make on-demand decisions during program executions. Such dynamic program re-optimizations are critically important for database query execution, datacenter optimizations, graph computations, where unexpected scenarios can happen without prior knowledge.

C. Debugging

Beside programming and optimizations, debugging is another important machine function for the approach of SDS. A well-trained machine can do the debugging job by detecting possible anomalies from executing traces of a set of given test cases. We believe two levels debugging work are possible for automatic programming machines. First, a machine can work as a regular software tester, who designs test cases against target programs. By understanding the software requirements, regardless how they are defined, e.g., by special instructions (e.g., UML) or by natural languages, a machine can automatically design test cases and behave like a human software tester.

The second level is that a machine can work beyond simple software tests but be able to discover hidden software bugs based on its capability of detecting anomalies. For example, the data race problem for parallel programming is notoriously hard to solve, but it is totally possible for a machine to detect the problem if its underlying DNN models are well-trained by a set of experiences and knowledges on a collection of data race problems. Simply speaking, because a machine can have a huge memory and a farly faster computation speed, it can certainly discover more software bugs than human testers, if the machine can understand how humans do the job of debugging.

IV. THE SDS ROADMAP: WHAT HUMANS CAN DO FOR MACHINES?

After we have described the SDS approach and its possible usage for software development, we are in a position to answer the ultimate question: How can we make it happen? In this section, we provide a R&D roadmap to turn the SDS approach into a reality in the near future, which essentially solves the problem of “what humans can do for machines”.

A. Building Learning Models with Logics

Unlike simple pattern recognitions in image/voice domains which are clearly suitable for deep learning-based approaches,
automatic programming is believed to be much harder, which requires reasoning and optimizations based on logics and mathematics proving. Therefore, we believe the first critical step for automatic programming is to combine the neural network based deep learning approach with the logic reasoning based formal approach into a unified framework that shrinks the possible search space for a programming requirement. In this direction, a research problem is how to embed a Z3 [14] or Coq [15]-like tool into a deep learning framework for the purpose of avoiding unnecessary search in a large space.

B. Building Learning Materials/curriculum

As we know, a supervised learning-based approach for a specific purpose must be based on some materials to be learned, which formed the ground truth knowledge. For example, ImageNet [16] is the key for the success of image recognition efforts based on its labelled data sets. However, there is no such a golden standard for automatic programming. To teach a machine efficiently learn how to write a program, even with the possibility of using reinforcement learning, humans should provide a clearly-defined example program set to be the curriculum for the learning programs. We believe different domains, such as GPU programming, distributed programming, and Web programming, must have their own specific curriculum. It is unclear and yet an open question whether it is possible to have a general-purpose program set that can be used for various programming tasks.

C. Building Software Frameworks for Machines

The foundation for automatic programming is that there exist a set of available building blocks that can either be basic units for a machine to use to assemble a program or learning examples to write similar blocks. However, current computing frameworks (e.g., Hadoop or Spark) or programming languages (e.g. C or Java) are designed for human developers which lack machine-understandable definitions for each function or primitive. For example, the C library function query can only be understood by a human programmer for its semantic meaning. To make automatic programming really happen, it is our human’s job to define clear specifications for machines.

D. Building Security/Trust Specification/Mechanisms

For a machine-generated program, the ultimate question is “Can we trust it”? It is easier if the program can be clearly expressed into a human-readable program, while much harder, if possible, when the program is expressed by a deep neural network model where the program behavior is hidden inside the model structure and weights of neural connections. Humans must design the security and trust specifications and corresponding enforcement mechanisms before deploying any software-defined software into real-world applications. The problem can be more complicated by the possibility that the check program is also generated by another program, which poses challenges for humans to design an architecture with defense walls that determine whether we should allow machine-generated programs to pass.

V. A Vision Conclusion

The number of world-class Go masters is much smaller than the number of highly skilled programmers in the world due to a very different intellectual level requirement. If machines can play the roles of highly intelligent Go masters, we believe that they can also become top programmers to deliver best quality code interacting with all kinds of hardware devices. This is the way of Software-defined Software (SDS), and our future way.

REFERENCES