True-JIT – Learning and Prediction of Compilation Sequences in a Centralized JIT Compiler

Anonymous Author(s)
Submission Id: 755

Abstract

In recent years we have observed the development of centralized Just-in-Time (JIT) compilers serving many compiler-less Language Virtual Machine (LVM) clients at once. While centralized JIT compilation reduces CPU and memory overhead due to global code caching and code reuse, it introduces additional latency between LVM clients and the central JIT compilation server. In this paper we investigate opportunities for hiding this latency. We explore how sequences of JIT compilation requests can be learned by the centralized JIT compiler and predicted. We train a long short-term memory (LSTM) network with JIT compilation requests and, during deployment, we use this to proactively compile and deliver native code to the LVM clients in a true just-in-time fashion, i.e. with minimal latency for the LVM clients. We have implemented our novel scheme in a distributed WebAssembly environment, and evaluated it against several benchmark suites including real-world applications from various application domains. We show that learning of JIT compilation sequences is feasible, and predictive speculation is effective in hiding JIT compilation latency. We demonstrate that centralized JIT compilation assisted by code caching, predictive compilation and code delivery reduces JIT compilation latency by, on average, 1-2 orders of magnitude for complex and timing-critical workloads.

1 Introduction

JIT compilation promises to deliver application portability [17] and improved runtime performance [33] through dynamic code generation and optimization. Used widely in the implementation of programming languages such as JavaScript, Java, Kotlin and C#, JIT compilation has found application from small embedded systems [29] to High-Performance Computing (HPC) [34], web browsers [9] and mobile devices [30]. A more recent trend includes the use of JIT compiled languages and JIT compilers in Cloud settings [24, 25].

However, it is well known that JIT compilation introduces startup delays [16] and compilation overhead during runtime [27]. Both of these factors are particularly undesirable in Cloud environments, where high responsiveness [10, 39] and cost of utilized compute and memory resources [15] are major considerations. Redundant JIT is particularly expensive for users who pay for resources as and when they use them, e.g. in Function-as-a-Service (FaaS) settings.

Against this background, centralized JIT compilation approaches have emerged where a single JIT server produces native code for *multiple* LVM clients [8, 25]. For example, IBM's *JITServer* [25] compiler for the Java Virtual Machine (JVM) makes extensive use of global code caching, aiming to avoid repeated compilation wherever possible. While this improves data center resource utilization, network communication between the LVM clients and the centralized JIT compiler introduces *additional* delay.

1.1 Key Ideas

In this paper we explore two key ideas to *minimize* JIT compilation latency in such a centralized JIT compilation scheme. First, we *speculatively compile* code in the JIT compiler and warm up its centralized code cache. Second, we *speculatively deliver* code to client LVMs over the network. The aim of combined speculative compilation and code delivery is to ensure that client LVMs never have to wait for code generation, but can continuously execute compiled code from their local code caches.

Central to achieving our goals is the observation that sequences of compilation requests issued to the central JIT compilation service are not random, but follow the generally well-behaved control flow structure of the target application. Modelling these sequences as sentence structures we train an LSTM network, which can then be used to predict the most likely sequence of future compilation requests. We use this to accomplish truly Just-in-Time (JIT) code management, where waiting times for JIT compiled code are largely eliminated.

We explore the novel concepts of this work in a disaggregated WebAssembly JIT compiler developed from scratch. We observe that as WebAssembly applications grow in complexity, users are beginning to experience increased file sizes and substantial increases in startup time due to compilation latency [36]. However, our work is not limited to WebAssembly, but could be adapted to other languages and JIT compiler frameworks.

1.2 Contributions

In this paper we make following contributions:

1. We develop a scalable, centralized JIT compilation infrastructure serving multiple WebAssembly client LVMs,
2. we present a novel method for minimizing JIT compilation latency based on speculative compilation and delivery of code using an LSTM network, and

Benchmark	#Runs	Avg. JIT Seq. Length	Similarity Score		
			Min	Max	Avg
gcc	218	4403	0.34	1.0	0.6
mcf	7	50	1.0	1.0	1.0
cactuBSSN	11	907	1.0	1.0	1.0
povray	10	519	0.67	1.0	0.78
lbm	27	29	0.93	1.0	0.97
xalancbmk	8	1756	0.77	0.92	0.84
x264	12	296	0.44	1.0	0.87
deepsjeng	12	101	0.77	0.93	0.87
leela	12	216	0.92	0.99	0.95
nab	13	96	0.99	1.0	1.0
exchange2	13	42	0.98	1.0	0.99
xz	51	153	0.25	1.0	0.92

Table 1. Similarity scores (1.0: identical; 0.0: nothing in common) for JIT compilation sequences for the SPEC CPU2017 benchmarks across the Alberta workloads: Each benchmark is run with multiple workloads, for which we record the sequences of JIT compiled functions. We then compare the JIT compilation sequences for each benchmark and observe high similarity despite variation of input data.

3. we systematically evaluate our LLVM based True-JIT system against a baseline representing the concepts developed in IBM's OpenJ9 centralized, caching *JIT-Server*.

1.3 Motivating Observations

Similarity of Application Execution. As we seek to predict the most likely next function(s) needed by an LVM client we are interested in the similarity between runs of the same application with different inputs. For this we compute pairwise Ratcliff/Obershelp similarity scores [11] for each the SPEC CPU2017 benchmarks [14] using the Alberta workloads [7] as shown in Table 1. Using a JIT compiler which compiles functions as they are encountered, we generate the sequence of JIT compilation requests for each run of a program. We then compute the similarity scores across all compilation sequences for each benchmark. We observe that different inputs lead to different, but highly similar JIT compilation sequences with similarity scores in the range of 25-100%, and typically above 80%. For many benchmarks distinct inputs which lead to identical compilation sequences. The *GCC* benchmark is of special interest as more than 200 distinct workloads are available. While its minimal similarity score is just 34%, clusters of inputs result in highly similar application behaviors, i.e. with just five clusters JIT compilation sequences can be covered with intra-cluster similarities above 80%. We exploit this similarity for prediction based on prior learning.

Scope for hiding JIT compilation latency. Consider the graph in Figure 1. For the SPEC CPU2017 `x264` benchmark

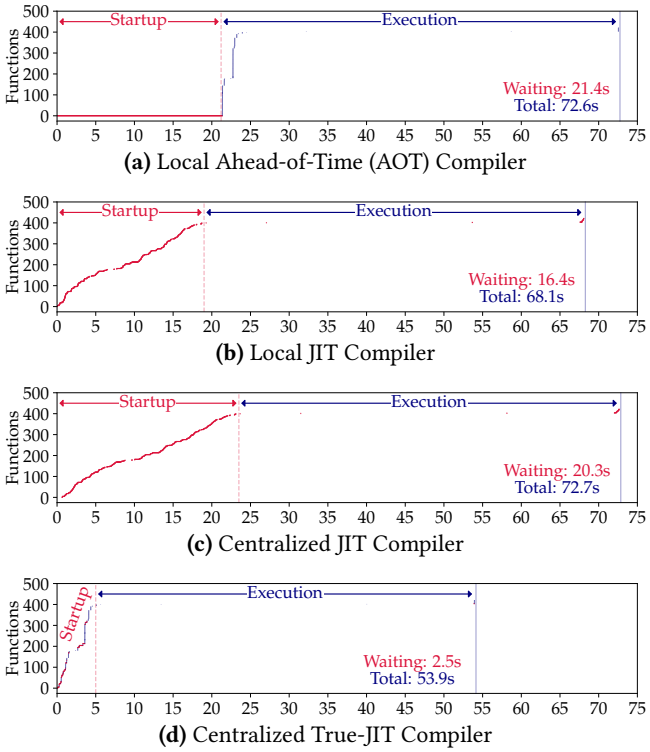


Figure 1. Execution traces with number of compiled functions over time for the SPEC CPU2017 x264 benchmark. Local AOT compilation (in (a)) shows a distinct startup phase before code execution starts, whereas local and centralized JIT compilation interleave compilation and execution during startup (in (b) and (c)). Centralized JIT compilation increases end-to-end execution time due to a prolonged startup phase. True-JIT (in (d)) almost eliminates the startup phase as native code becomes available very quickly and end-to-end execution time is being reduced.

we show execution traces with the number of compiled functions (on the y-axis) over time (on the x-axis) for four different compilation schemes: local AOT, local JIT, centralized JIT and our novel predictive, centralized True-JIT scheme. A local AOT scheme, i.e. an LVM with an integrated AOT compiler, is shown in Figure 1a. Upon launch of the benchmark the AOT compiler kicks in and it takes several seconds for native code to become available. Only after all functions have been compiled execution will start.. After this initial 21 second AOT compilation phase no further code is being compiled until the application terminates. Compare this to the JIT scheme in Figure 1b, which employs an integrated JIT compiler. Execution starts immediately, and functions are compiled by the local JIT compiler as soon as they are encountered in the startup phase. After around 16s the LVM settles and enters a stable execution phase, which ends after around 68s. Next we show a centralized JIT compilation scheme in Figure 1c, which is similar to e.g. [25]. We notice that centralized JIT compilation takes more time than

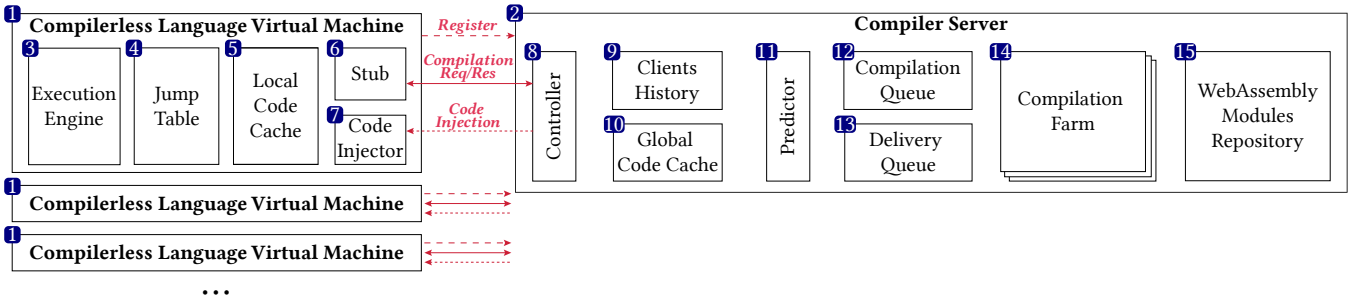
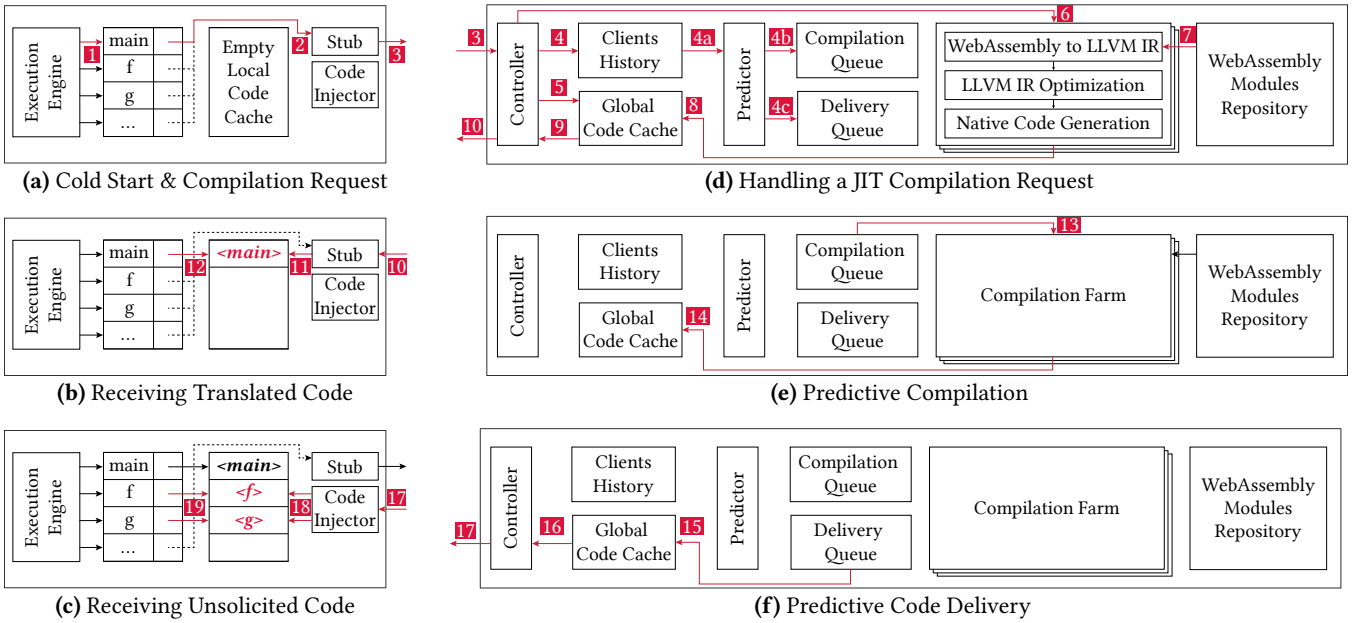


Figure 2. High-level overview of the True-JIT system architecture. A centralised JIT Compilation Server serves compilation requests issued by LVM clients, and using an LSTM predictively compiles and delivers code likely to be used in the near future.



(a) Cold Start & Compilation Request

(b) Receiving Translated Code

(c) Receiving Unsolicited Code

(d) Handling a JIT Compilation Request

(e) Predictive Compilation

(f) Predictive Code Delivery

Figure 3. Internal operation of the compiler-less LVM (a) – (c) and the central True-JIT Compilation Server (d) – (f).

the local JIT scheme. This is because Intermediate Representation (IR) and compiled code code are transferred over a network, which adds extra latency. In contrast, our centralized True-JIT compiler reduces the initial startup phase to less than 3s, resulting in an overall runtime of under 54s. This is due to True-JIT's ability to predict what functions are required next and to predictively compile and deliver them to the LVM before they are needed, thus preventing stalls resulting from misses in its local code cache.

We realize that a centralized JIT compiler provides ample scope for effective caching of generated code, there are opportunities for reuse of cached code across LVM clients, and predicting what functions LVM clients encounter next in their current execution is both feasible and worthwhile. In the remainder of the paper we show what is required to make such a predictive, centralized JIT system work effectively and efficiently.

2 True-JIT: System Architecture, Operation, Machine Learning Model and Training

In this section we introduce the system architecture of our True-JIT system as shown in Figure 2 and explain how its components (1-15) interact with each other (1-19) throughout the execution of an application in Figure 3. We show the operation of predictive JIT compilation and code delivery, and explain our approach to model training and deployment.

2.1 True-JIT System Architecture

At the top level as shown in Figure 2, True-JIT comprises a number of compiler-less LVMs 1 and a centralised JIT Compilation Server 2 with an integrated WebAssembly Module Repository 15 connected via a network.

Upon launching of an application the compiler-less Virtual Machine (VM) 1 registers itself with the Compilation Server 2. It communicates the application Uniform Resource Identifier (URI), i.e. its identifier in the WebAssembly Module

Repository 15, along with other client information such as its Instruction Set Architecture (ISA). Throughout execution the VM issues compilation requests for individual functions to the JIT Compiler Server. It receives responses from the central server comprising the code generated for the requested function. In addition to this request/response interaction between LVMs and the JIT compilation server, the server can also actively initiate code injection targeting any of the registered LVMs. We use this mechanism to proactively populate local code caches with native code, which is predicted by the server to be most likely needed by an LVM client in the near future.

2.2 LVM Client Operation

We narrow our focus to the operation of the LVM clients as shown in Figure 3(a)–(c). Consider the cold start scenario in Figure 3a, where the local code cache is empty and all pointers in the jump table point towards the compilation stub. The Execution Engine 3 inside the LVM then retrieves 1 a function pointer for the next function from the Jump Table 4, e.g. for function *main*, upon which we either locate the code for this function in the Local Code Cache 5, or request remote compilation 2,3 of this function via JIT client stub 6. Functions already contained in the local code cache can be invoked and executed immediately, whereas remotely compiled functions can only be executed once code has been delivered, linked, added to the local code cache and the pointer in the jump table has been rewired. The update of the jump table ensures that subsequent invocations of the same function are being directed to the code now available in the Local Code Cache.

The response of the central JIT compiler is shown in Figure 3b. When the compiler responds 10 to a previously issued compilation request, compiled code is added 11 to the code cache and the jump table is being updated 12.

A code injection scenario is shown in Figure 3c. Unsolicited code injection requests 17 from the central JIT compiler are routed via a code injector, which is responsible for adding the delivered code to the local code cache 18 and jump table 19 as before.

2.3 Central JIT Compiler Operation

We visualise the operation of the central JIT compiler in Figure 3(d)–(e) and cover handling of incoming JIT compilation requests (in Figure 3d), predictive compilation (in Figure 3e), and predictive code delivery (in Figure 3f).

2.3.1 Serving JIT Compilation Requests.

Upon receiving a compilation request 3 from an LVM, the JIT Compilation Controller 8 adds 4a this request to the client's history 9. It then checks 5 its Global Code Cache 10. If available, compiled code stored in the Global Code Cache is returned 9,10 to the client. Otherwise, the JIT Compilation Controller instructs 6 the LLVM-based JIT Compilation Task Farm 14 (similar to [12]) to fetch 7 the relevant WebAssembly function from the Module Repository 15 for compilation. Once compiled, compiled code is registered 8 in the Global Code Cache and further delivered to the client 9,19.

Incoming compilation requests trigger 4a the predictor 11, which relies on the client's request history to determine which functions to add 4b,4c to the Compilation Queue 12 or Delivery Queue 13.

We distinguish between two predictive scenarios in True-JIT: (a) Predictive JIT compilation, and (b) predictive code delivery. These two scenarios are illustrated in Figure 3e and Figure 3f. Both scenarios involve the JIT compilation server as an active entity.

2.3.2 Predictive JIT Compilation.

The goal of predictive JIT compilation is to ensure that as many as possible requests to the JIT compilation server can be served from the global code cache. This is achieved by speculatively compiling WebAssembly functions to native code before these have been requested by any client.

True-JIT's predictor captures a history window 8 of N recent compilation requests for each client, which form the basis for its prediction. The predictor then predicts a sequence of M functions most likely required by a client in the near future. Both N and M are configurable, and we discuss the impact of varying N and M in section 3. In True-JIT we employ an LSTM for this prediction task, and provide details concerning its configuration, training and deployment in subsection 2.5 and subsection 2.6.

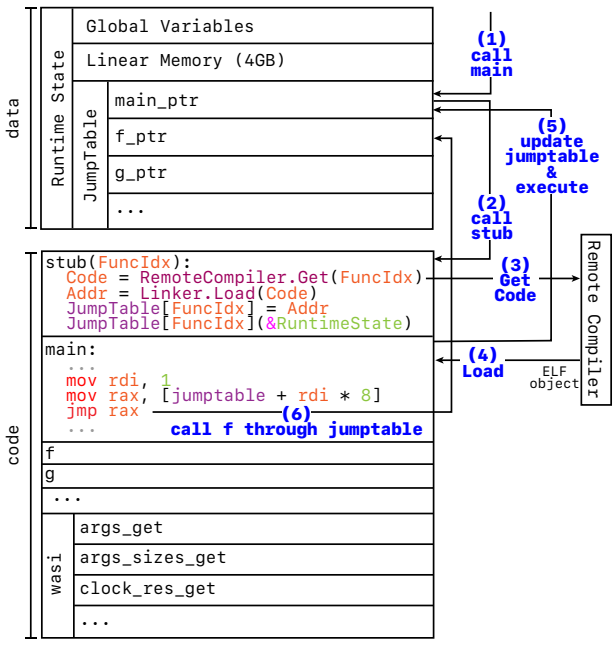


Figure 4. Client LVM address space layout and function invocation strategy utilising a dynamically updated jump table.

Functions predicted to be required in the near future are added to the Compilation Queue, which feeds 13 to the server's multi-threaded compilation task farm 14, which fetches the required WebAssembly functions from the module repository before compiled code is added 14 to the global code cache.

2.3.3 Predictive Code Delivery.

The goal of predictive code delivery is to ensure that requests to the central JIT compilation server are minimised, and compiled code is available in the local code caches by the time it is needed by the LVM clients.

To achieve this goal the Controller 8 inspects the Delivery Queue 13 and retrieves 15,16 compiled code from the global code cache 10. It then actively pushes 17 these compiled functions to an LVM client, ready for injection into its local code cache.

2.4 Virtual Address Space Layout and Function Invocation Strategy

The centralised JIT compiler needs to generate code that is compatible with execution model of the LVM clients. This affects, in particular, the organisation of the virtual address space and approach to function invocation. Figure 4 shows how we are solving both of these problems.

Each LVM client maintains a data and a (native) code region: In the data region we maintain the client's runtime state, including WebAssembly global variables and 4GB of linear memory addressable by WebAssembly. In addition, we maintain a jump table with function pointers to functions stored in the code region. This implies that all calls to JIT generated functions are indirect calls, which must be resolved through this jump table. In our evaluation in section 3 we will see that the performance impact of this indirection is negligible. Upon launch of an application this function jump table is initialised for all pointers to point to a stub function, which initiates remote compilation for the called function. Once the code has been generated, delivered and linked, we update the jump table to point to the native code in the code region, thus making sure that subsequent calls to the same function will execute directly from the local code cache in the client's code region. We additionally provide code wrappers for WebAssembly System Interface (WASI) functions, which provide convenient abstractions to system functions, e.g. libc library functionality.

2.5 Machine Learning Model

We approach the task of language prediction as a sequence-to-sequence task. Given a dataset $\mathcal{D}$ with N sequences

$$\{(\mathbf{s_n}, \mathbf{c_n}) \mid n \in \{0..N-1\}\}$$

where $\mathbf{s_n}$ is a sequence of function names and $\mathbf{c_n}$ is the corresponding next function name in the sequence, we aim to train a model T parameterised by $\boldsymbol{\theta}$ with maximum likelihood estimation:

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} \prod_{n=0}^{N-1} P\left(\mathbf{c_n} \mid \mathbf{s_n}; \boldsymbol{\theta}\right)$$

In practice, we minimise the negative log-likelihood, which is equivalent to cross-entropy (CE). For each mini-batch (i.e., group of $B << N$ examples in the dataset), we use the Adam optimiser to update the parameters of the model given the gradient of the cross-entropy loss function:

$$\boldsymbol{P}(\boldsymbol{\theta}) = \mathrm{T}\left(\boldsymbol{s}_n, \boldsymbol{c}_n \mid \boldsymbol{\theta}\right)$$

$$\mathrm{CE}(\boldsymbol{\theta}) = -\sum_{l=1}^{L-1} \log \boldsymbol{P}(\boldsymbol{\theta})\left[\boldsymbol{c_n}[l+1], l\right]$$

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \eta \cdot \nabla \mathrm{CE}(\boldsymbol{\theta})$$

where $\boldsymbol{P}(\boldsymbol{\theta})$ are the token probabilities predicted by the model, L is the sequence length, η is the learning rate hyperparameter, and $\boldsymbol{\theta}'$ are the updated parameters.

2.5.1 LSTM Model.

We use an LSTM model, a type of Recurrent Neural Network (RNN) that is capable of learning long-term dependencies. The LSTM model consists of an embedding layer, an LSTM layer, and a fully connected layer. The embedding layer converts the input function names into dense vectors of fixed size. The LSTM layer processes the sequence of function name vectors and computes a hidden state for each function name in the sequence. The fully connected layer takes the hidden state of the last function name in the sequence and outputs a probability distribution over the next function name in the sequence.

2.6 Training and Deployment

Successful prediction necessitates training with relevant training inputs, which result in representative application behaviours. In True-JIT we rely on offline training, i.e. training ahead of time. However, we are also aware that introducing an additional step in the software development workflow would be an impediment to the wider adoption of True-JIT. Hence, we leverage data collected during standard regression testing for model training.

2.6.1 Regression Testing and Model Training.

Consider Figure 5 for an overview of the code submission, regression testing, model training and deployment cycle for True-JIT. Developers push ❶ their application, e.g. as part of their CI/CD workflow, to a cloud based repository. This triggers compilation to WebAssembly ❷ and automated regression testing ❸ against predefined test cases that exercise a wide range of application behaviours, and in general, ensure good code coverage [18]. The True-JIT compiler is already being used for code generation during regression testing, however, without its predictive capabilities. Instead, the compiler traces JIT compilation requests ❹ used for training ❺ its application-specific LSTM model. After testing has

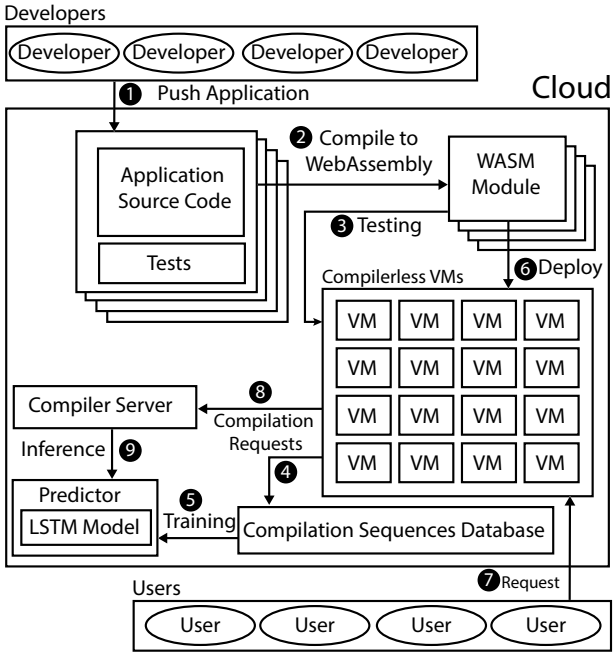


Figure 5. The True-JIT workflow: Model Training as part of regression testing in a Continuous Integration/Continuous Delivery (CI/CD) process.

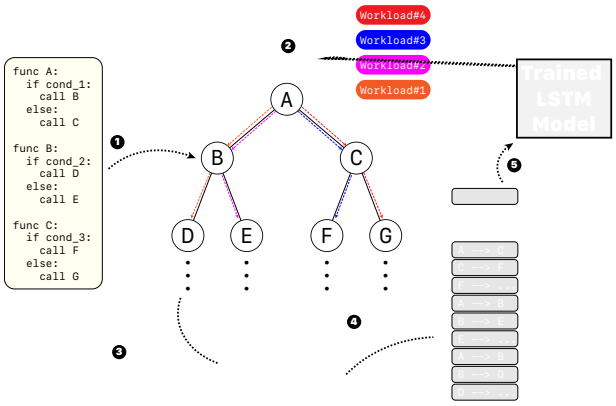


Figure 6. An example showing how training data is extracted from sequences of JIT compilation requests ("traces"), each representing an execution of the program with different input data.

completed successfully, the application is being deployed ❻. When this application is run ❼, compilation requests ❽ are being directed to the JIT compilation server, which then uses its predictor for inference ❾.

We will see in section 3 that an LSTM trained for an application is relatively robust w.r.t. application changes. This means that can reuse a once trained model for e.g. a later version of the same application despite potentially substantial code changes. This is an advantage of True-JIT where a predictive model can be easily reused, but generated native code cannot.

2.6.2 Extraction of Training Data.

Functions are identified by their ID, and we are collecting sequences of function IDs ("traces") resulting from the execution of target applications with different input data sets in the centralised JIT compiler. A function ID is included in a trace the first time the function is invoked, i.e. when it triggers JIT compilation. Subsequent calls to the same function are not recorded in the trace as these later invocations will be served out of the client's code cache and do not require JIT compilation. Using compiler terminology, a trace is the result of an in-order traversal of the dynamic call tree where duplicates have been removed. Divergences in control flow as a result of different inputs of an application will result in multiple, different traces, which together are used to train an application-specific LSTM. We illustrate this with an example in Figure 6.

2.6.3 Model Architecture and Training Details.

The LSTM model has an embedding size of 32, a hidden size of 64, and an output size equal to the size of the vocabulary. The model is trained using the Adam optimiser [26] with a learning rate of 0.01. The loss function is cross-entropy, which measures the dissimilarity between the predicted probability distribution and the actual distribution. The model is trained for 100 epochs. The trained model is retained for later use by the JIT compiler server.

3 Evaluation

We evaluate the effectiveness of True-JIT in the CloudLab data centre environment, using a range of application benchmarks and real-world applications. We are mainly concerned with True-JIT's ability to reduce or eliminate JIT compilation latency, but also investigate its ability for making accurate predictions, its memory footprint and impact on network overhead, as well as throughput and scalability. We also compare True-JIT with other, non-distributed WebAssembly runtimes, and the distributed *JITServer* JVM.

3.1 Experimental Methodology

We evaluate our True-JIT approach against benchmarks from three different benchmark suites: SPEC CPU2017 [14], PolyBench/C [32], JetStream 2 [22], CoreMark [1], MiBench [21] and WABench [35], which includes whole applications from various application domains. From the SPEC CPU2017 benchmark suite[1] we use `lbm`, `mcf`, `namd`, `x264`, and `nab`, which we compiled to WebAssembly using the Clang/LLVM compiler. Similarly, we have compiled the PolyBench/C, MiBench and CoreMark codes to WebAssembly. From the JetStream 2 benchmark suite we use the `gcc-loops`, `float-mm`, `HashSet`,

[1]Other SPEC CPU2017 benchmarks fail to run in any of the tested Webssembly environments due to partially unsupported WASI calls.

Node	CloudLab Clemson r650
CPU	2×36-core Intel Xeon 8360Y, 2.4GHz
Memory	256GB (16×16GB DDR4 3200MHz ECC)
Storage	480GB SATA SSD
Network	2×100Gb Mellanox ConnectX-6, PCIe v4.0

Table 2. Specification of CloudLab nodes in our evaluation.

Parameter	Values	
Cloud Provider	Google Cloud Platform, CloudLab	
Compiler Server	1 Physical Node 1 Process 144 Threads	
WebAssembly Virtual Machines	32 Physical Nodes 10000 Processes Single Thread	
Benchmarks	SPEC 2017, PolyBench/C, NPB, MiBench, JetStream2, CoreMark, WABench	
Global Code Cache	**Initial State**	**Size**
	Cold, Warm	Unbounded
Compilation Strategy	AOT, JIT, TrueJIT	
Predictor Architecture	LSTM	
Predictions	**History**	**Depth**
	1, 2, ..., 10	1, 2, ..., 200
LVMs Release Strategy	Simultaneous	
	Staggered (1, ..., 1000ms)	
Network	**Latency**	**Bandwidth**
	1µs, ..., 10ms	10, ..., 100Gbps

Table 3. Experimental parameter space of our evaluation.

`quicksort`, and `tsf` benchmarks, which are provided as WebAssembly applications.

We have been running our experiments in a CloudLab environment using virtual machines according to the specifications in Table 2.

We provide a full overview of our benchmarking environment and configuration space in Table 3. In individual experimental settings we vary certain parameters to evaluate aspects of the system's behaviour. Across all experiments, both local and global code caches are unbounded for the purpose of our evaluation. In practice, though, we anticipate the use of size-limited caches with more sophisticated code cache management and replacement schemes, e.g. [23].

Throughout our evaluation we frequently refer to and compare against the *standard centralised JIT compilation scheme*, which is our re-implementation of the centralised caching concept of *JITServer* [25] in our own framework.

3.2 Key Results

We show the True-JIT's effectiveness in reducing JIT compilation latency in Figure 7, where we compare JIT compilation latency for three different schemes: Local JIT compilation embedded in the LVM client, centralised JIT like in *JITServer*, and True-JIT using an LSTM predictor trained prior to benchmarking. For this experiment we run each benchmark on 10000 client LVMs, supported by a single JIT compilation server for the two centralised compilation schemes. All code caches are initially cold. Clients are being released in an 'open loop', i.e. we launch a new client every 10ms. This scenario models the rapid up-scaling of a workload across many client LVMs. We report our measurements of the total JIT latency incurred by the LVM clients as speedup relative to the local JIT compilation baseline.

We observe that all benchmarks benefit from centralised JIT compilation, where total time spent on compilation as experienced by LVM client is reduced. In fact, the standard centralised JIT compilation scheme speeds up JIT compilation by a factor of 7.6 due to the benefit of global code caching. This is in line with the findings in [25]. True-JIT further improves the JIT latency and we measure an average speedup of 55.1 over local JIT compilation. This is strong evidence for the efficacy of True-JIT's ability to hide startup JIT latency when up-scaling workloads.

Across benchmark suites there is relatively little variation and we see True-JIT deliver consistent JIT compilation speedups for kernel and application benchmarks, even where the standard centralised JIT approach without prediction experiences greater variation, e.g. across some of the WABench benchmarks.

3.3 Impact of Network Latency

How does True-JIT perform under conditions of increased network latency? To answer this question we introduce additional network latency between the LVM clients and the JIT server and observe the impact on JIT latency from the clients' perspective. Ranging between 0-10ms an extra delay is added to each network transfer to model the effect of increased network latency. We show our findings in Figure 8, where we compare JIT compilation slowdown as a function of additional network latency for both the standard centralised JIT scheme and True-JIT. In line with our previous results in Figure 7 we see that even without additional latency, True-JIT outperforms the standard centralised scheme in terms of compilation latency by almost 8x. As we increase the network latency we see compilation latency increasing for both schemes. However, we observe that True-JIT is more effective in its ability to hide this increase in network latency. At 10ms extra latency, True-JIT experiences a slowdown of 1.2 over the standard scheme at 0ms extra latency, whereas the non-predictive standard scheme experiences a slowdown of 25x. True-JIT's ability to gracefully degrade in the face of network latency can be ascribed to its predictive code delivery mechanism, which ensures that local code caches are more effective and experience fewer misses, resulting in fewer latency-inducing JIT compilation requests.

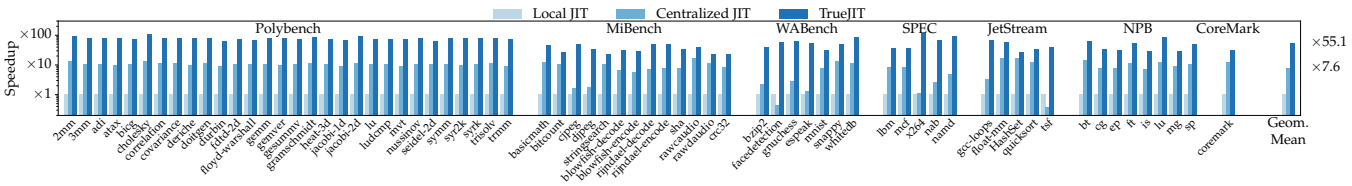


Figure 7. Speedup of standard centralised JIT compilation without prediction and True-JIT over local JIT scheme. We consider JIT compilation times only for our set of benchmarks. The standard centralised scheme already delivers a 7.6 speedup for JIT compilation due to global code caching, while True-JIT results in an average speedup of 55.1 over local JIT compilation thanks to its predictive JIT compilation and code delivery strategy (10000 clients, staggered release separated by 10ms).

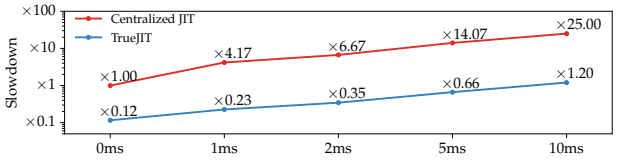


Figure 8. Impact of network latency on JIT compilation performance. We introduce additional network latency between 0-10ms and observe JIT compilation slowdown. Where the standard centralized scheme suffers from increased network latency, True-JIT's performance degrades gracefully.

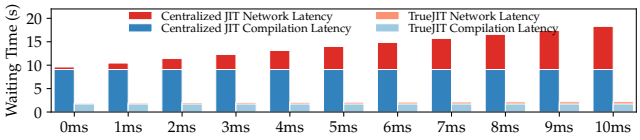


Figure 9. Breakdown of total client waiting time into (a) JIT compilation latency and (b) network latency. Client waiting time is dominated by JIT compilation, whereas network latency only contributes less than 10% of the total waiting time. As we increase additionally introduced network latency, we see True-JIT's ability to hide this latency unlike the non-predictive centralised JIT compilation scheme.

3.4 Network Overhead

How much network overhead is introduced by True-JIT? Sending JIT compilation requests to the True-JIT compilation server and, especially, transmitting compiled code back to LVM clients introduces additional network traffic. In order to quantify the impact of this additional network overhead we analyse what fraction of waiting time (as experienced by a LVM client) can be attributed to actual JIT compilation as opposed to code requests and transmissions over the network. We show our findings in Figure 9.

For this experiment we use the *x264* application, which with a binary code size of around 1MB is one of the larger benchmarks used in our study. Execution of the benchmark results in 38 distinct JIT compilation requests over a period of 37s. We observe that network overhead contributes less than 10% of the overall waiting time experienced by LVM clients in either of the observed schemes, while the majority of waiting time is due to JIT compilation. As we increase additionally introduced network latency from 0 to 10ms as in our previous experiment, we see True-JIT's ability to hide this increased latency, while the standard centralised JIT scheme experiences a significant increase in client waiting time.

For the *x264* application with its 1MB code size used in this experiment we observed an average bandwidth utilisation of around 27kb/s for the transmission of code from the JIT compiler to the LVM client. Thus, a single 10Gbit/s Ethernet network connection would be capable of sustaining code delivery for up to 30000-40000 LVM clients.

3.5 Cold Start vs Warm Start

Can True-JIT improve over the standard centralised JIT scheme once the global code cache has been warmed up? We set up an experiment where we compare three scenarios: (a) Standard centralised JIT compilation with a cold global code cache, (b) standard centralised JIT compilation with a warm global code cache, and (c) True-JIT with a warm global code cache. In Figure 10 we show our findings based on the *x264* benchmark and a single LVM. We treat the cold cache scenario (a) as our baseline. Warming up the code cache for scenario (b) results in a JIT compilation speedup of 11.3 – this is because native code can be served directly from the global code cache, which is faster than actual JIT compilation. For the True-JIT scenario we observe a further improvement, and from the client's perspective JIT compilation is sped up by another 18.5x. This is due to True-JIT's predictive code delivery, which does not wait until the LVM's local cache runs out of code, but proactively fills the local code cache and, thus, avoids most JIT compilation requests altogether. This confirms that True-JIT can indeed improve over centralised JIT compilation with its warmed up global code cache as a result of its proactive code delivery strategy.

3.6 Memory Footprint

What is the memory footprint of True-JIT? For this we compare local JIT compilation and standard centralised JIT compilation with True-JIT in Figure 11 for the SPEC CPU2017 *x264* benchmark. For each scheme we measure its complete

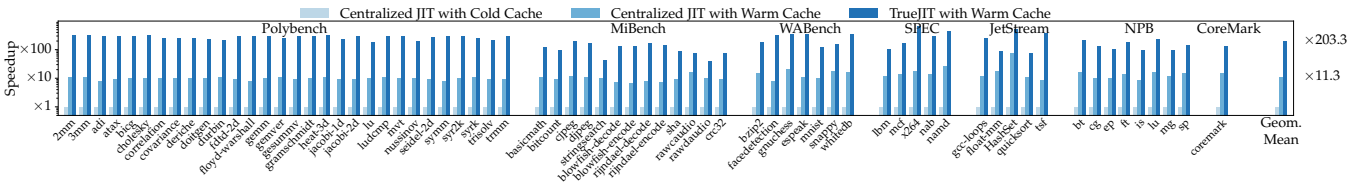


Figure 10. Comparison of the impact of warm global code caches for the centralised JIT scheme and True-JIT. While a warm global code cache substantially reduces waiting time until code becomes available to LVM clients, True-JIT's predictive code delivery results in substantial further improvements as local code caches are being proactively populated just-time-time.

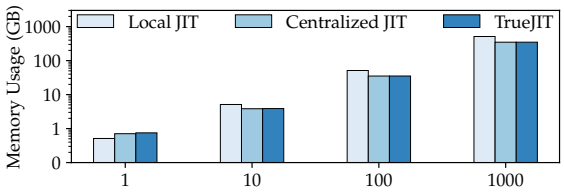


Figure 11. Aggregated client and server memory usage for local JIT, centralised JIT and True-JIT for 1-1000 LVM nodes running the SPEC CPU2017 *x264* benchmark.

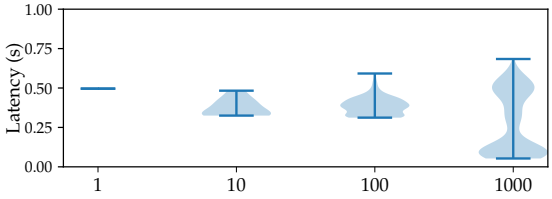


Figure 12. JIT latency distributions as the number of LVMs is increased from 1 to 1000.

footprint for configurations between 1-1000 LVM clients. For each configuration we include client and server memory usage accounting for each of their data and code areas, code caches, and in the case of True-JIT, its integrated LSTM predictor.

We observe that all three considered schemes scale linearly with the number of LVM nodes. For very small configurations the local JIT scheme has an advantage, e.g. in the 1-node configuration the local JIT scheme occupies about half as much memory as either of the two centralised schemes. This is due to the memory cost of the distributed infrastructure, e.g. communication libraries. However, already for 10 LVMs the centralised schemes consume less memory than the local scheme. The memory penalty for True-JIT's predictor is small, and for configurations with 10 or more clients the memory consumption is on par with that of the standard centralised scheme.

3.7 Throughput and Scalability

Are True-JIT's centralised JIT compiler and predictor performance bottlenecks? To answer this question we scale up the number of LVM clients for a single JIT compilation server and observe the JIT compilation latency for these clients. This is shown in Figure 12 for the *Polybench 2MM* benchmark and configurations involving 1, 10, 100, and 1000 LVM clients. We expect JIT compilation times to rise sharply once we saturate the JIT compilation server. While we observe that the range of observed latencies increases as we go from a single client to 1000 clients, the average latencies remain the same. The distribution widens as we scale up, possibly as a result of beginning contention. However, as the majority of requests to the centralised JIT server will be served from the global code cache, the effectiveness with which we can lookup and retrieve code from this software cache is the determining factor to True-JIT's scalability, and replication can be employed to increase its bandwidth [38]. We can conclude that in this configuration with 1000 LVM clients we have not yet encountered the end of True-JIT's scalability curve.

3.8 Accuracy and Depth of Predictions

How accurate are the predictions made by True-JIT and how far ahead can we predict? Consider Figure 14 in which we show prediction accuracy of our LSTM for the SPEC CPU2017 *gcc* in response to a selected prediction depth. Prediction depth here is the number of JIT compilation requests in the future that we expect to receive based on the current history of JIT requests issued to the centralised JIT compiler. In Figure 14 we plot this accuracy over depth as an average across each point of execution of the benchmark.

We observed an initially high prediction accuracy of over 95% for the immediately following JIT compilation request. As a general trend we also observe that prediction accuracy decreases as we increase the depth of prediction, i.e. prediction further ahead into the future become less and less reliable. Even at a prediction depth of 10 we still record a prediction accuracy of 80%, which we will see later is still good enough to translate into performance gains. In general, the prediction accuracy follows a declining curve without sharp drops all the way up to a depth of 100 where it drops to 50%, which indicates relative robustness of our predictor.

Next, consider Figure 13 where we show relative benefit of deeper predictions in terms of JIT compilation speedup. Across all benchmarks we observe speedup of JIT compilation as we progressive increase prediction depth from 1-gram LSTM to 100-gram LSTM. The response to deeper prediction

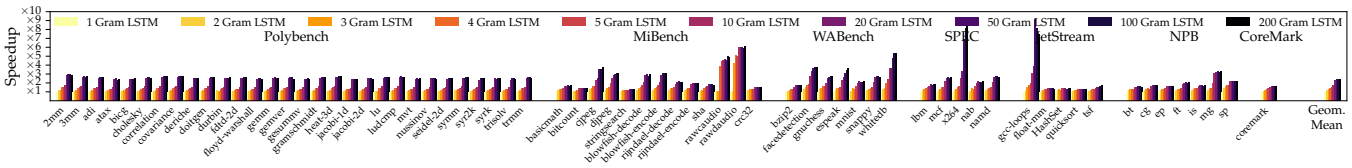


Figure 13. Achievable speedup of JIT compilation related waiting times in LVM clients in relation to the depth of prediction. Deeper predictions lead to reduced JIT latency, but we observe diminishing returns with a sweet spot at around a depth of 10.

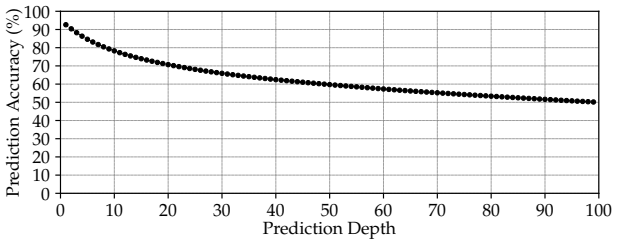


Figure 14. Prediction accuracy vs prediction depth for the SPEC CPU2017 *gcc* benchmark: While prediction accuracy drops the further ahead we predict (depths), the slope of decline is gentle and even 10 JIT compilation requests into the future can be predicted with around 80% accuracy.

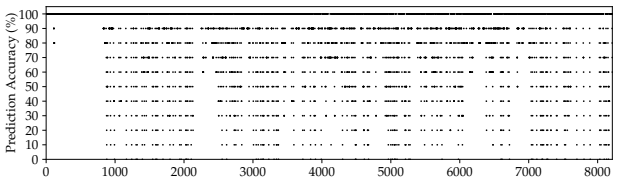
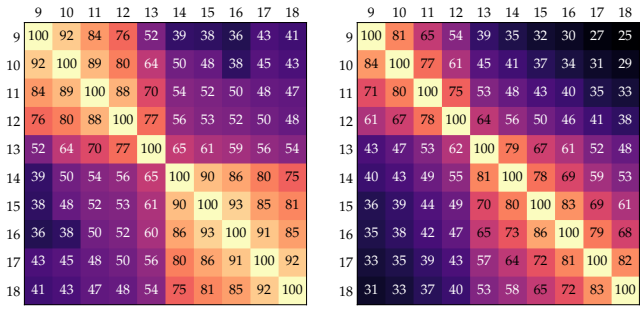


Figure 15. Prediction accuracy over time as we predict the 10 functions ahead at each point during the execution of *gcc*.



(a) Similarity **(b)** Prediction Accuracy

Figure 16. Heatmaps of pair-wise Ratcliff/Obershelp similarity scores between traces of LLVM 9.0 and later versions (left), and prediction accuracy when the LSTM is trained with LLVM 9.0 and evaluated against later versions (right).

depends on the specific benchmark, and we see variation with some applications (e.g. *stringsearch*) only marginally benefiting to others (e.g. *x264*) experiencing over 8x speedup of JIT latency. Typically, applications with a large number of functions benefit more from deeper prediction than small benchmarks with few functions. We also observe diminishing returns of ever-increasing prediction depth, with a sweet spot at depth 10.

In Figure 15 we show how prediction accuracy is not constant, but a function over time as we progress through program execution. For the *gcc* benchmark shown in the diagram, we initially observe 100% prediction accuracy, which is related to application startup code that does not exhibit any control flow variation. Throughout the execution we observe alternating phases with lower and higher prediction accuracy. This phase changing application behavior is well studied and exploited, e.g. in processor simulation methodologies like SimPoint [31].

3.9 Model Reuse

Can we reuse a predictive model without retraining for another version of the same application? To answer this question we set up an experiment involving all major versions of the LLVM compiler framework ranging from version 9.0 (released in 2019) to 18.0 (released in 2024). We build each compiler and use it to compile the *sqlite3* library using optimisation level *-O3*. For each compilation run we record the trace of invoked functions, in the same way as described for our motivating example in section 1. We then compute the similarity between traces (in Figure 16a) and the prediction accuracy of our LSTM when trained with one version of LLVM, and evaluated against all other versions in turn (in Figure 16b).

Despite significant code changes between major LLVM versions, there is a high degree of similarity between JIT compilation traces. Those from LLVM 9.0 and 10.0 exhibit more than 90% of similarity, and between versions 9.0 and 11.0 we still observe 84% similarity. Even between extremes, i.e. versions 9.0 and 18.0 released 5 years apart, more than 40% of similarity can found. This similarity directly translates into prediction accuracy. An LSTM trained with version 9.0 still provides over 80% prediction accuracy when evaluated against version 10.0. While prediction accuracy drops faster than similarity, this experiment demonstrates that our prediction model is robust enough to be useful even when the

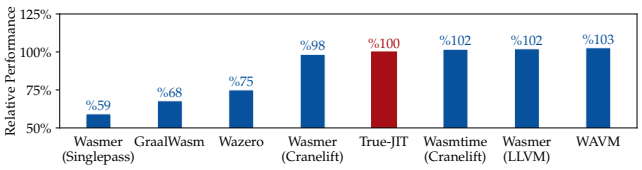


Figure 17. Comparing True-JIT with other WebAssembly environments using a weighted geometric mean of end-to-end execution times across benchmarks. True-JIT delivers similar code quality as its top-performing competitors.

underlying application has gone through major revisions, confirming that reuse of a model without retraining is viable.

3.10 Comparison to Other WebAssembly LVMs

How does True-JIT's performance compare to that of other WebAssembly runtimes? We have compared True-JIT to state-of-the-art JIT-compiled WebAssembly runtimes, including Wasmer [3] (LLVM, Cranelift, Singlepass), WAVM [5], Wasmtime [4], Wazero [6], GraalWasm [2]. Performance results as weighted geometric mean over the set of benchmarks used in this paper are plotted in Figure 17, normalised to True-JIT's baseline performance. Measuring end-to-end performance, i.e. combined execution time and JIT compilation time, True-JIT is on par with Wasmer (LLVM and Cranelift), WAVM and Wasmtime (Cranelift) and significantly exceeds the performance of Wazero, GraalWasm, and Wasmer (Singlepass). These results confirm that True-JIT is able to compete with the top-performing WebAssembly runtimes featuring integrated JIT compilers.

3.11 Summary of Results

Our evaluation shows that:

- Centralized JIT compilation, even when utilizing a global code cache like *JITServer*, introduces additional compilation latency due to network transfers of compilation requests and code.
- JIT compilation sequences for an application exhibit a high degree of similarity, even when run with different inputs.
- LSTMs can effectively learn JIT compilation sequences and be used to predict the next n functions to compile.
- It is feasible to speculatively compile and deliver code to LVMs, largely hiding JIT compilation latency in a distributed environment.
- True-JIT's predictive JIT compilation and code delivery approach is particularly suitable to latency-sensitive environments such as FaaS where workloads need to be upscaled quickly.
- Training of the predictor for the True-JIT scheme can be incorporated in an CI/CD workflow, and trained models can be successfully reused across versions of the same application.

4 Related Work

JITServer [25] has been an inspiration for the work presented in this paper. *JITServer* is a disaggregated caching JIT compiler implemented in the *Eclipse OpenJ9* JVM. Aiming for improvements of system-wide resource utilisation it pioneers centralised caching of compiled native code and its reuse in JVMs running on different client machines. In our work we go beyond centralised caching and explore speculative code compilation and code delivery in a WebAssembly environment, thus hiding the latency introduced by moving the JIT compiler out of the LVM and placing it on a different machine. Persistent code caches and code sharing across processes has been studied in e.g. [13, 20] as an effective means to reuse JIT compiled code. *ShareJIT* [37] introduces a global code cache and code sharing across processes in the Android ART runtime. Conceptually, *JITServer* and *ShareJIT* are similar in their use of a global code cache, but *ShareJIT* is limited to a single host whereas *JITServer* can serve multiple networked clients. Code sharing in *ShareJIT* results in reduced JIT compilation time like in our True-JIT scheme, whereas *JITServer* improves overall system utilization at the cost of increased JIT compilation time. While earlier work exists, this is typically restricted to sharing of meta-information, but not compiled code [19]. Prediction of control flow outside the realm of processor microarchitecture typically focuses on branch behaviour [28], and not sequences of functions.

5 Summary & Conclusions

In this paper we have introduced our predictive True-JIT framework, which utilises machine learning for the prediction of future JIT compilation requests and code required by actively running WebAssembly LVM clients. We show that JIT compilation requests can be successfully predicted, and we exploit this to proactively compile code in a centralised JIT compiler and deliver native code to LVM clients before these issue JIT compilation requests. We have demonstrated True-JIT's ability to hide JIT compilation latency across a wide range of benchmarks running in a cloud environment. We observe substantial reductions in waiting time for LVM clients while incurring only minimal network and memory overheads. True-JIT is particularly suited to support the rapid auto-scaling of workloads with ultra-low startup latency as required by FaaS.

5.1 Future Work

We will investigate methods for eliminating the need for training of new applications, e.g. by transfer learning where a pre-trained model is being re-used for a different application. Furthermore, we plan to integrate dynamic optimisation and code versioning into True-JIT.

References

[1] A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, 29(5):18–29, 2009.

[2] Graalwasm, 2024.

[3] Wasmer, 2024.

[4] Wasmtime, 2024.

[5] Wavm, 2024.

[6] Wazero, 2024.

[7] José Nelson Amaral, Edson Borin, Dylan R. Ashley, Caian Benedicto, Elliot Colp, Joao Henrique Stange Hoffmam, Marcus Karpoff, Erick Ochoa, Morgan Redshaw, and Raphael Ernani Rodrigues. The alberta workloads for the spec cpu 2017 benchmark suite. In *ISPASS*, pages 159–168. IEEE Computer Society, 2018.

[8] Tim Anderson. Azul lays claim to massive efficiency gains with remote compilation for Java, 2021.

[9] Rafael Auler, Edson Borin, Peli de Halleux, Michał Moskal, and Nikolai Tillmann. Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler. In *Compiler Construction: 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 23*, pages 218–237. Springer, 2014.

[10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46, 2010.

[11] P. Black. Ratcliff/obershelp pattern recognition.

[12] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. *SIGPLAN Not.*, 46(6):74–85, jun 2011.

[13] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 61–70, New York, NY, USA, 2008. Association for Computing Machinery.

[14] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE transactions on services Computing*, 5(2):164–177, 2011.

[16] Eric Coffin, Scott Young, Kenneth B Kent, and Marius Pirvu. A roadmap for extending MicroJIT: a lightweight just-in-time compiler for decreasing startup time. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 293–298, 2019.

[17] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[18] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Software Testing, Verification and Reliability*, 25(4):371–396, 2015.

[19] D. Dillenberger, R. Bordawekar, C. W. Clark III, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, and R. W. St. John. Building a java virtual machine for server applications: The jvm on os/390. *IBM Systems Journal*, 39(1):194–210, 2000.

[20] Lauren Guckert, Mike O'Connor, S Kumar Ravindranath, Zhuoran Zhao, and V Janapa Reddi. A case for persistent caching of compiled javascript code in mobile web browsers. In *Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT)*, 2013.

[21] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, page 3–14, USA, 2001. IEEE Computer Society.

[22] David Y. Hancock, Jeremy Fischer, John Michael Lowe, Winona Snapp-Childs, Marlon Pierce, Suresh Marru, J. Eric Coulter, Matthew Vaughn, Brian Beck, Nirav Merchant, Edwin Skidmore, and Gwen Jacobs. Jetstream2: Accelerating cloud computing via jetstream. In *Practice and Experience in Advanced Research Computing*, PEARC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Tim Hartley, Foivos S. Zakkak, Andy Nisbet, Christos Kotselidis, and Mikel Luján. Just-in-time compilation on arm—a closer look at call-site code consistency. *ACM Trans. Archit. Code Optim.*, 19(4), sep 2022.

[24] Serhii Ivanenko, Rodrigo Bruno, Jovan Stevanovic, Luís Veiga, and Vojin Jovanovic. Cloudjit: A just-in-time faas optimizer (work in progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 12–19, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Alexey Khrabrov, Marius Pirvu, Vijay Sundaresan, and Eyal de Lara. Jitserver: Disaggregated caching JIT compiler for the JVM in the cloud. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 869–884. USENIX Association, 2022.

[26] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

[27] Chandra J Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8):717–738, 2001.

[28] Tao Li, Lizy Kurian John, Anand Sivasubramaniam, N. Vijaykrishnan, and Juan Rubio. Understanding and improving operating system effects in control flow prediction. *SIGPLAN Not.*, 37(10):68–80, oct 2002.

[29] Geetha Manjunath and Venkatesh Krishnan. A small hybrid JIT for embedded systems. *ACM SIGPLAN Notices*, 35(4):44–50, 2000.

[30] Hyeong-Seok Oh, Beom-Jun Kim, Hyung-Kyu Choi, and Soo-Mook Moon. Evaluation of Android Dalvik virtual machine. In *Proceedings of the 10th international workshop on java technologies for real-time and embedded systems*, pages 115–124, 2012.

[31] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, page 318–319, New York, NY, USA, 2003. Association for Computing Machinery.

[32] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437:1–1, 2012.

[33] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices*, 36(11):180–195, 2001.

[34] Guillermo L Taboada, Sabela Ramos, Roberto R Expósito, Juan Tourino, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. *Science of Computer Programming*, 78(5):425–444, 2013.

[35] Wenwen Wang. How far we've come – a characterization study of standalone webassembly runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–241, 2022.

[36] Elliott Wen and Jens Dietrich. Wasmslim: Optimizing webassembly binary distribution via automatic module splitting. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 673–677, 2023.

[37] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. Sharejit: Jit code cache sharing across processes and its practical implementation. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[38] P. Yu, A. Leff, and J. Wolf. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel & Distributed Systems*, 4(11):1185–1204, nov 1993.

[39] Xianrong Zheng, Patrick Martin, Kathryn Brohman, and Li Da Xu. CLOUDQUAL: a quality model for cloud services. *IEEE transactions on industrial informatics*, 10(2):1527–1536, 2014.