Towards Robust and Scalable Large Language Models

By

Paras Jagdish Jain

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Co-Chair
Professor Joseph E. Gonzalez, Co-Chair
Professor Jacob Steinhardt
Professor Matei Zaharia

Spring 2023

Towards Robust and Scalable Large Language Models

By

Paras Jagdish Jain

Abstract

Towards Robust and Scalable Large Language Models

By

Paras Jagdish Jain

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Co-Chair

Professor Joseph E. Gonzalez, Co-Chair

This dissertation addresses two significant challenges of large language models (LLMs): robustness and scalability. Firstly, we focus on improving large language model robustness through the lens of learning code representations. I highlight our work on ContraCode which learns representations of code that are robust to label-preserving edits. Secondly, we tackle scalability challenges from a systems perspective. We present Checkmate, a system to support training models beyond GPU memory capacity limits through optimal rematerialization. Furthermore, Skyplane, a system that optimizes bulk data transfers between cloud object stores, enables training models on larger pre-training datasets in the cloud. Together, these contributions present a roadmap for enhancing the robustness and scalability of large language models.

To my family and my partner.

Contents

Acknowledgements

Firstly, I would like to extend my deepest gratitude to my advisors, Ion Stoica and Joseph E. Gonzalez. The value of focus and drive that I have learned from Ion has been a guiding force in accomplishing ambitious research visions. From Joey, I have learned the true essence of optimism in the face of critical reviewers and the numerous challenges that a PhD journey brings.

I also extend my heartfelt thanks to my committee members, Matei Zaharia and Jacob Steinhardt, for their valuable feedback and guidance throughout my research journey. Special mention goes to Barna Saha and Laurent El Ghaoui for aiding me in applying convex optimization to my first major paper, a tool that has since been pivotal in my research. I also want to express my gratitude to Pieter Abbeel for the enriching collaborations in generative modeling.

Special thanks go out to John Kubiatowicz, under whose guidance I had the pleasure of teaching CS 162 operating systems. I thank Azalia Mirhoseini, Safeen Huda, and Martin Maas for their mentorship during my tenure at Google Brain. I particularly would like to thank Polo Chau and Shang-Tse Chen who took a chance mentoring me as a freshman at Georgia Tech and teaching me the fundamentals of academic research.

I extend my appreciation to the Skyplane team, with whom I had the privilege of working closely during the final years of my PhD. My gratitude goes out to Sam Kumar for the rigorous debates that undoubtedly made our paper more robust. I also thank Sarah Wooders, Shishir Patil, Shu Liu, Simon Mo, and Asim Biswal, my fellow PhD students on the Skyplane project. I also acknowledge the undergraduates I had the honor of mentoring, including Anton Zabreyko, Jason Ding, Xuting Liu, and Hailey Jang. I am also grateful to Vincent Liu and Daniel Kang for their guidance on the Skyplane project.

It was a pleasure to collaborate with many wonderful people during my PhD: Ajay Jain, Alexey Tumanov, Amir Gholami, Aniruddha Nrusimha, Azalia Mirhoseini, Bartolomeo Stellato, Conor Power, Dawn Song, Francesco Borrelli, Goran Banjac, Harikaran Subbaraj, Jeffrey Ichnowski, Ken Goldberg, Kurt Keutzer, Martin Maas, Matei Zaharia, Matthew Wright, Michael Luo, Peter Kraft, Pieter Abbeel, Prabal Dutta, Rehan Durrani, Safeen Huda, Sam Kumar, Sarah Wooders, Shishir G Patil, Tathagata Das, Tianjun Zhang, Wendi Zhang, Simon Mo, Yu Gai and Zhanghao Wu.

Lastly, but most importantly, I dedicate this thesis to my family. My parents, Pradeep and Chanchal Jain, my sister, Aditi Jain, and my brother, Ajay Jain, have been a source of constant support throughout my PhD. Collaborating with Ajay has been a treasure, and I'm grateful for our shared journey. Lastly, my partner, Anjali Shankar, deserves my deepest gratitude for her unwavering love and support throughout this journey.

Chapter 1

Introduction

In 1945, Vannevar Bush envisioned the memex, a hypothetical device that could store and index all of humanity's knowledge, enabling users to query and navigate the knowledge with "wholly new forms of encyclopedias". Although Bush imagined the memex as a mechanical microfilm-based device, his vision went far beyond the physical form of the device. He foresaw the development of systems capable of deep language understanding, knowledge storage, and reasoning.

Large language models (LLMs) have made significant progress towards this vision by learning representations of language that can be queried and reasoned over. Unlike previous language models, these neural networks are trained on vast amounts of data to predict words and understand language. They have achieved human-level performance on certain benchmarks, but face significant challenges that limit their widespread deployment. Specifically, large language models confront crucial hurdles in two dimensions: robustness and scalability.

Robustness in large language models is a multifaceted challenge. While large language models have shown remarkable progress in understanding and generating text, they still struggle with hallucinations, sensitivity to input perturbations and compositional generalization. Scalability, on the other hand, is a challenge of size and computational resources. For large language models, the cross-entropy loss scales as a power-law with model size, dataset size, and the amount of compute used for training. In this dissertation, I contribute to the ongoing efforts to improve the robustness and scalability of large language models.

In Chapter 2, we investigate strategies to enhance the robustness of large language models. One question central to this discourse is whether language modeling objectives lead to learning robust semantic representations, or if they merely predict tokens based on local context. To answer this question, we turn to the context of source code, where the semantics of a program are defined by its execution. We explore the contrastive pre-training task, ContraCode, which learns code functionality instead of form. ContraCode pre-trains a neural network to distinguish functionally similar variants of a program from many non-equivalent distractors. This strategy has shown improvement in JavaScript summarization and TypeScript type inference accuracy. We also introduce a new zero-shot JavaScript code clone detection dataset, with results indicating that ContraCode is both more robust and semantically meaningful compared to other methods.

In Chapter 3, we start addressing the scalability challenges of large language models by examining the "memory wall" problem that arises during the training of large models. Here, we introduce Checkmate, a system that optimally trades off computation time and memory requirements for DNN training. Checkmate solves the tensor rematerialization optimization problem, a generalization of prior checkpointing strategies. It determines optimal rematerialization schedules using off-the-shelf MILP solvers and accelerates millions of training iterations. The system scales to complex, realistic architectures and is hardware-aware, using accelerator-specific, profile-based cost models. Checkmate enables the training of real-world networks with up to 5.1× larger inputs.

In Chapter 4, we explore the management of large pre-training datasets, another aspect of the scalability challenge. Specifically, we investigate how to collect and move these datasets between cloud destinations. We present Skyplane, a system for bulk data transfer between cloud object stores that uses cloud-aware network overlays. It optimally balances price and performance using mixed-integer linear programming to determine the optimal overlay path and resource allocation for data transfer. Skyplane outperforms public cloud transfer services by up to 4.6× for transfers within one cloud and by up to 5.0× across clouds.

Chapter 2

Improving the robustness of large language models

Large language models have achieved remarkable progress on natural language tasks, yet struggle when generalizing to new domains. An open question is whether language modeling objectives actually learn semantic representations or simply predict tokens based on local context.

It remains challenging to quantitatively evaluate the robustness of large language models. We consider the domain of source code where the meaning of a program is defined by its execution. Programming languages provide clear semantics and structure that allows quantitative evaluation.

In “Contrastive Code Representation Learning”, I investigate adversarial robustness of code representations to gain insights into their semantic meaning. I find that popular models like RoBERTa are highly sensitive to simple, semantics-preserving edits to input code, showing their representations are not robust to rephrasing or reformatting the same program logic. To address this issue, I propose ContraCode, a pre-training approach that learns robust code semantics by generating program variants that differ syntactically but are functionally equivalent. ContraCode pre-trains a model to identify these variants among many non-equivalent distractors, requiring the model to develop representations indifferent to superficial changes in program form and structure. By learning representations aligned with program semantics rather than syntax, ContraCode improves robustness to adversarial evaluation.

This work was the result of a collaboration with Ajay Jain, Tianjun Zhang and Pieter Abbeel.

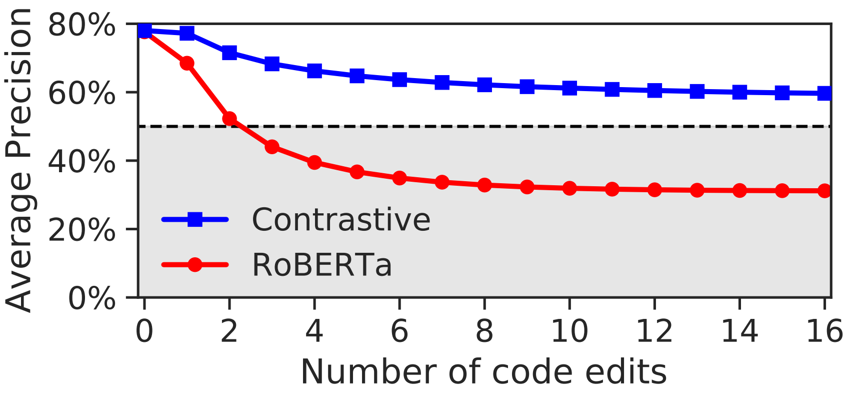
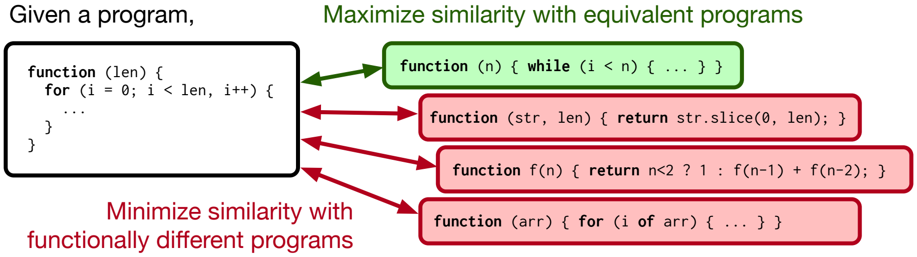


Figure 2.1: **Robust code clone detection:** On source code, *RoBERTa is not robust to simple label-preserving code edits* like renaming variables. Adversarially selecting between possible edits lowers performance below random guessing (dashed line). Contrastive pre-training with ContraCode learns a more robust representation of functionality, consistent across code edits.


Given a program,
Maximize similarity with equivalent programs
Minimize similarity with functionally different programs

```
function (len) {
  for (i = 0; i < len, i++) {
    ...
  }
}
```

```
function (n) { while (i < n) { ... } }
function (str, len) { return str.slice(0, len); }
function f(n) { return n<2 ? 1 : f(n-1) + f(n-2); }
function (arr) { for (i of arr) { ... } }
```

Figure 2.2: For many analyses, programs with the same functionality should have similar representations. ContraCode learns such representations by pre-training an encoder to retrieve equivalent, transformed programs among many distractors.

2.1 Introduction

Programmers increasingly rely on machine-aided programming tools that analyze or transform code automatically to aid software development [100]. Traditionally, code analysis uses handwritten rules, though the wide diversity of programs encountered in practice can limit their generality. Recent work leverages machine learning for richer language understanding, such as learning to detect bugs [142] and predict performance [122].

Still, neural models of source code are susceptible to adversarial attacks. Yefet, Alon, and Yahav [181] and Schuster, Song, Tromer, and Shmatikov [153] find accuracy degrades significantly under adversarial perturbations for both discriminative and generative code models. In our work, we investigate adversarial attacks on code clone detection. Successful adversarial attacks could circumvent malware detectors.

While self-supervision can improve adversarial robustness [77], we find that RoBERTa is sensitive to stylistic implementation choices of code inputs. Fig. 2.1 plots the performance of RoBERTa and ContraCode, our proposed method, on a code clone detection task as small label-preserving perturbations are applied to the input code syntax. With just three adversarial edits to code syntax, RoBERTa underperforms the random classifier (in gray). In Fig. 2.3, we show that RoBERTa's representations of code are sensitive to code edits as studied in prior work [145, 171, 172].

To address this issue, we develop ContraCode: a self-supervised representation learning algorithm that captures program semantics. We hypothesize that *programs with the same functionality should have similar underlying representations* for downstream code understanding tasks.

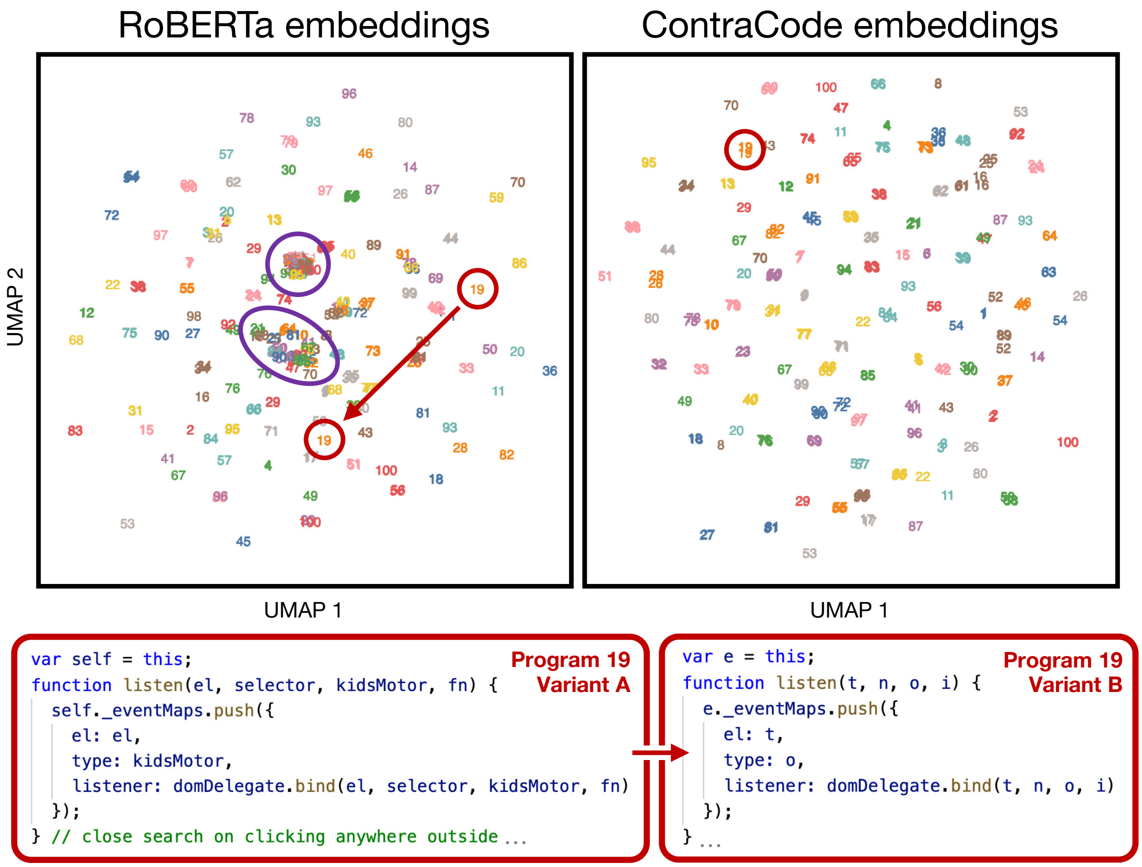
ContraCode generates syntactically diverse but functionally equivalent programs using source-to-source compiler transformation techniques (*e.g.*, dead code elimination, obfuscation and constant folding). It uses these programs in a challenging discriminative pretext task that requires the model to identify similar programs out of a large dataset of distractors (Fig. 2.2). To solve this task, the model must embed code semantics rather than syntax. ContraCode improves adversarial robustness in Fig. 2.1. Surprisingly, adversarial robustness transfers to better natural code understanding.

Our novel contributions include:

1. the novel use of compiler-based transformations as data augmentations for code,
2. the concept of program representation learning based on functional equivalence, and
3. a detailed analysis of architectures, code transforms and pre-training strategies, showing ContraCode improves type inference top-1 accuracy by 9%, learned inference by 2%–13%, summarization F1 score by up to 8% and clone detection AUROC by 2%–46%.

2.2 Related work

Self-supervised learning (SSL) is a learning strategy where some attributes of a datapoint are predicted from remaining parts. BERT [49] is a SSL method for NLP that reconstructs masked tokens as a pretext task. RoBERTa [114] further tunes BERT. Contrastive approaches minimize distance between learned representations of similar examples (positives) and maximize distance between dissimilar negatives [72]. CPC [76, 133] encodes segments of sequential data to predict future segments. SimCLR [38] and MoCo [40, 73] use many negatives for dense loss signal.

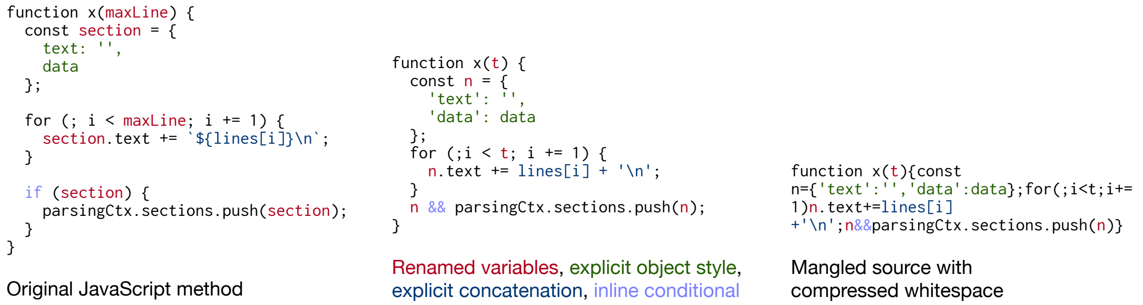


```
var self = this;
function listen(el, selector, kidsMotor, fn) {
  self._eventMaps.push({
    el: el,
    type: kidsMotor,
    listener: domDelegate.bind(el, selector, kidsMotor, fn)
  });
} // close search on clicking anywhere outside ...
```

```
var e = this;
function listen(t, n, o, i) {
  e._eventMaps.push({
    el: t,
    type: o,
    listener: domDelegate.bind(t, n, o, i)
  });
} ...
```

Figure 2.3: A UMAP visualization of JavaScript method representations learned by RoBERTa and ContraCode, in $\mathbb{R}^2$. Programs with the same functionality share color and number. RoBERTa's embeddings often do not cluster by functionality, suggesting that it is sensitive to implementation details. For example, many different programs overlap, and renaming the variables of Program 19 significantly changes the embedding. In contrast, variants of Program 19 cluster in ContraCode's embedding space.

Code representation learning We address clone detection [174], type inference [75], and summarization [9]. Others explored summarization [2, 6, 86, 128] and types [4, 5, 27, 135, 141, 173] for various languages. Inst2vec [24] embeds statements in LLVM IR by processing a flow graph with a context prediction objective [127]. Code2seq [9] embeds AST paths with an attentional encoder for seq2seq tasks. Kanade, Maniatis, Balakrishnan, and Shi [94] and Feng, Guo, Tang, Duan, Feng, Gong, Shou, Qin, Liu, Jiang, and Zhou [53] pre-train a Transformer on code using the masked language modeling (MLM) objective [49, 167].



```
function x(maxLine) {
  const section = {
    text: '',
    data
  };

  for (; i < maxLine; i += 1) {
    section.text += `${lines[i]}\n`;
  }

  if (section) {
    parsingCtx.sections.push(section);
  }
}
```

Original JavaScript method

```
function x(t) {
  const n = {
    'text': '',
    'data': data
  };
  for (;i < t; i += 1) {
    n.text += lines[i] + '\n';
  }
  n && parsingCtx.sections.push(n);
}
```

Renamed variables, explicit object style, explicit concatenation, inline conditional

```
function x(t){const
n={'text':'','data':data};for(;i<t;i+=
1)n.text+=lines[i]
+'\n';n&&parsingCtx.sections.push(n)}
```

Mangled source with compressed whitespace

Figure 2.4: A JavaScript method from our unlabeled training set with two automatically generated semantically-equivalent programs. The method is from the StackEdit Markdown editor.

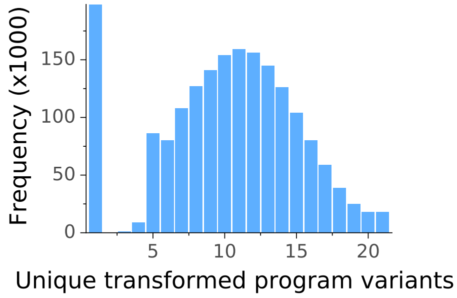


Figure 2.5: Histogram of the number of unique transformed variants per JavaScript method during pre-training.

Adversarial attacks on code models Yefet, Alon, and Yahav [180] find code models are highly sensitive to adversarial code edits in a discrimative setting. Schuster and Paliwal [152] discovers in-the-wild attacks on code autocompletion tools. Compared to language models, code models may be more vulnerable to adversarial attacks due to synthetic labels [25, 54, 142] and duplication [3] that degrade generalization.

2.3 Approach

Our core insight is to use compiler transforms as data augmentations, generating a dataset of equivalent functions (§2.3.1, 2.3.2). We then use a contrastive objective to learn a representation invariant to these transforms (§2.3.3).

	Code compression		**Identifier modification**
✓	Reformatting (R)	✓	Variable renaming (VR)
✓	Beautification (B)	✓	Identifier mangling (IM)
✓	Compression (C)		**Regularization**
✓	Dead-code elimination (DCE)	✓	Dead-code insertion (DCI)
✓	Type upconversion (T)	✓	Subword regularization (SW)
✓	Constant folding (CF)	✗	Line subsampling (LS)

✓ = semantics-preserving transformation ✗ = lossy transformation

Table 2.1: We augment programs with 11 automated source-to-source compiler transforms. 10 are correct-by-construction and preserve operational semantics.

2.3.1 Compilation as data augmentation

Modern programming languages afford great flexibility to software developers, allowing them to implement the same function in different ways. Yet, crowdsourcing equivalent programs from GitHub is difficult as verifying equivalence is undecidable [22, 93] and approximate verification is costly and runs untrusted code [118].

Instead of searching for equivalences, we propose correct-by-construction data augmentation. We apply compiler transforms to unlabeled code to generates many variants with equivalent functionality, *i.e.* operational semantics. For example, dead-code elimination (DCE) is an optimization that removes operations that do not change function output. While DCE preserves functionality, Wang and Christodorescu [171] find that up to 12.7% of the predictions of current supervised algorithm classification models change after DCE.

We parse a particular source code sequence, *e.g.* `W*x + b` into a tree-structured representation `(+ (* W x) b)` called an Abstract Syntax Tree (AST). We then transform the AST with automated traversal passes. A rich body of prior programming language work explores parsing and transforming ASTs to optimize a program. If source code is emitted by the compiler rather than machine code, this is called source-to-source transformation or transpilation. Transpilation is common for optimizing and obfuscating dynamic languages like JavaScript. Further, if each transform preserves code semantics, then any composition also preserves semantics.

We implement our transpiler with the Babel and Terser compiler infrastructures [121, 150] for the JavaScript programming language. In future work, a language-agnostic compiler [103] could be used to extend ContraCode to other languages. Each compiler transformation is a function $\tau : \mathcal{P} \to \mathcal{P}$, where the space of programs $\mathcal{P}$ is composed of the set of valid ASTs and the set of

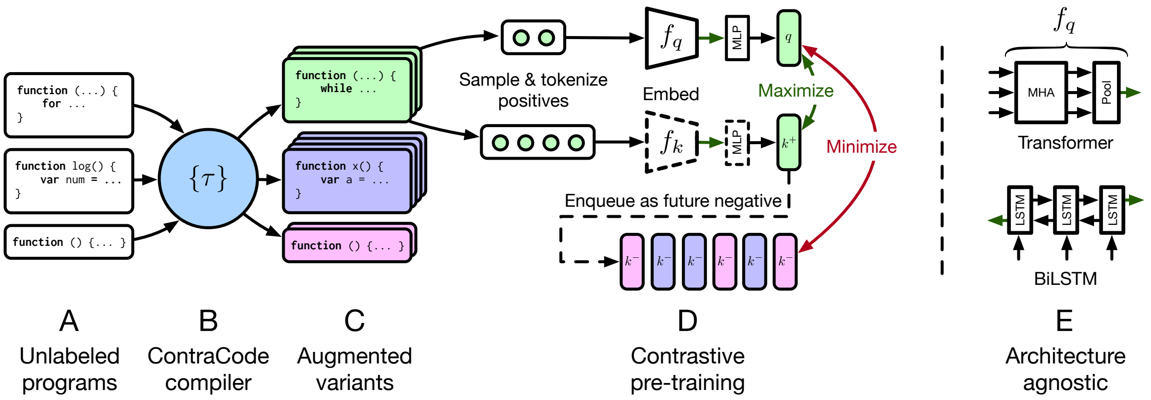


Figure 2.6: ContraCode pre-trains a neural program encoder f_q and transfers it to downstream tasks. **A-B.** Unlabeled programs are transformed **C.** into augmented variants. **D.** We pre-train f_q by maximizing similarity of projected embeddings of *positive* program pairs–variants of the same program–and minimizing similarity with a queue of cached negatives. **E.** ContraCode supports any architecture for f_q that produces a global program embedding such as Transformers and LSTMs. f_q is then fine-tuned on smaller labeled datasets.

programs in tokenized source form. Fig. 2.4 shows variants of an example program. Table 2.1 list program transformations in detail, but we broadly group them into three categories:

- **Code compression** changes the syntactic structure of code and performs correct-by-construction transforms such as pre-computing constant expressions.
- **Identifier modifications** substitute method and variable names with random tokens, masking some human-readable information in a program but preserving functionality.
- Finally, **Regularizing transforms** improve model generalization by reducing the number of trivial positive pairs with high text overlap. The line subsampling pass in this group potentially modifies program semantics.

2.3.2 Diversity through transform dropout

Stochastic augmentations in other modalities like random crops generate diverse outputs, but most of our compiler-based transformations are deterministic. To produce a diverse set of transformed programs, we randomly apply a subset of available compiler passes in a pre-specified order, applying transform τ_i with probability p_i. Intermediate programs are converted between AST and source form as needed for the compiler. Algorithm 1 details our transform dropout procedure.

Algorithm 1 Transform dropout for stochastic program augmentation.

```
Input: Program source $x$, transformation functions $\tau_1, \ldots \tau_k$, transform probabilities $p_1, \ldots p_k$, count $N$
Returns: $N$ variants of $x$
$\mathcal{V} \leftarrow \{x\}$, a set of augmented program variants
for Sample $i \leftarrow 1 \ldots N-1$ do
    $x' \leftarrow x$
    for transform $t \leftarrow 1 \ldots k$ do
        Sample $y_t \sim \text{Bernoulli}(p_t)$
        if $y_t = 1$ then
            if RequiresAST($\tau_t(\cdot)$) and ¬IsAST($x'$) then $x' \leftarrow$ ParseToAST($x'$)
            else if ¬RequiresAST($\tau_t(\cdot)$) and IsAST($x'$) then $x' \leftarrow$ LowerToSource($x'$)
            $x' \leftarrow \tau_t(x')$
        end if
    end for
    if IsAST($x'$) then $x' \leftarrow$ LowerToSource($x'$)
    $\mathcal{V} \leftarrow \mathcal{V} \cup \{x'\}$
end for
return $\mathcal{V}$
```

Figure 2.5 measures the resulting diversity in programs. We precompute up to 20 augmentations of 1.8M JavaScript methods from GitHub. Algorithm 1 deduplicates method variants before pre-training since some transforms will leave the program unchanged. 89% of the methods have more than one alternative after applying 20 random sequences of transformations. The remaining methods without syntactically distinct alternatives include one-line functions that are obfuscated. We apply subword regularization [107] as a final transformation to derive different tokenizations every batch, so pairs derived from the same original method will still differ. All transformations are fast; our compiler transforms 300 functions per second on a single CPU core.

2.3.3 Contrastive pre-training

We extend the Momentum Contrast (MoCo) methodology [73] that was designed for contrastive image representation learning. In our case, we learn a program encoder f_q that maps a sequence of program tokens to a single, fixed dimensional embedding. We organize programs into *functionally similar positive pairs* and *dissimilar negative pairs*. Generating two augmentations of the same GitHub program yields a positive pair (x^q, x^{k^+}), and an augmentation of a different program yields a negative x^{k^-}. The program x^q is called a "query" used to retrieve the corresponding "key" x^{k^+} during contrastive pre-training. We use these to shape representation space, drawing positives together and pushing away from negatives. Negatives are important to prevent the encoder f_q from mapping all programs to the same, trivial representation [151].

Pre-training objective Like He, Fan, Wu, Xie, and Girshick [73], we use the InfoNCE loss [133], a tractable objective that frames contrastive learning as a classification task: can the positives be identified among negatives? InfoNCE computes the probability of selecting the positive by taking the softmax of projected embedding similarities across a batch and a queue of negatives. Eq. (2.1) shows the InfoNCE loss, a function whose value is low when q is similar to the positive key embedding k^+ and dissimilar to negative key embeddings k^-. t is a temperature hyperparameter proposed by Wu, Xiong, Yu, and Lin [177].

$$-\log \frac{\exp(q \cdot k^+/t)}{\exp(q \cdot k^+/t) + \sum_{k^-} \exp(q \cdot k^-/t)} \tag{2.1}$$

The query representation $q = f_q(x^q)$ is computed by the encoder network f_q, and x^q is a query program. Likewise, $k = f_k(x^k)$ using a separate key encoder f_k. The summation $\sum_{k^-}$ in the normalizing denominator is taken over the queue of pre-computed negatives in the batch.

Following He, Fan, Wu, Xie, and Girshick [73], to reduce memory consumption during pre-training, we cache embedded programs from past batches in a queue containing negative samples, as shown in Fig. 2.6. The query encoder f_q is trained via gradient descent while the key encoder f_k is trained slowly via an exponential moving average (EMA) of the query encoder parameters. The EMA update stabilizes the pre-computed key embeddings across training iterations. Since keys are only embedded once per epoch, we use a very large set of negatives, over 100K, with minimal additional computational cost and no explicit hard negative mining.

ContraCode is agnostic to the architecture of the program encoder f_q. We evaluate contrastive pre-training of 6-layer Transformer [170] and 2-layer BiLSTM [82, 152] architectures (§2.4).

Transfer learning After pre-training converges, the encoder f_q is transferred to downstream tasks. For code clone detection, we use $f_q(x)$ without fine-tuning. For tasks where the output space differs from the encoder, we add a task-specific MLP or Transformer decoder after f_q, then fine-tune the resulting network end-to-end on labeled task data.

2.4 Evaluation

In order to evaluate whether ContraCode defend against adversarial code inputs, we benchmark adversarial code clone detection accuracy [21]. We evaluate results over natural and adversarial edits. We then evaluate how improvements to adversarial robustness translate to improvements on established in-the-wild code benchmarks. While improvements on adversarial benchmarks

	Natural code		**Adversarial** (N=4)		**Adversarial** (N=16)	
	AUROC	**AP**	**AUROC**	**AP**	**AUROC**	**AP**
Edit distance heuristic	69.55±0.81	73.75	31.63±0.82	42.85	12.11±0.54	32.46
Randomly initialized Transformer	72.31±0.79	75.82	22.72±0.20	37.73	3.09±0.28	30.95
+ RoBERTa MLM pre-train	74.04±0.77	77.65	25.83±0.21	39.46	4.51±0.33	31.17
+ ContraCode pre-train	75.73±0.75	78.02	**64.97**±0.24	**66.23**	**58.32**±0.88	**59.66**
+ ContraCode + RoBERTa MLM	**79.39**±0.70	**81.47**	37.81±0.24	51.42	10.09±0.50	32.52

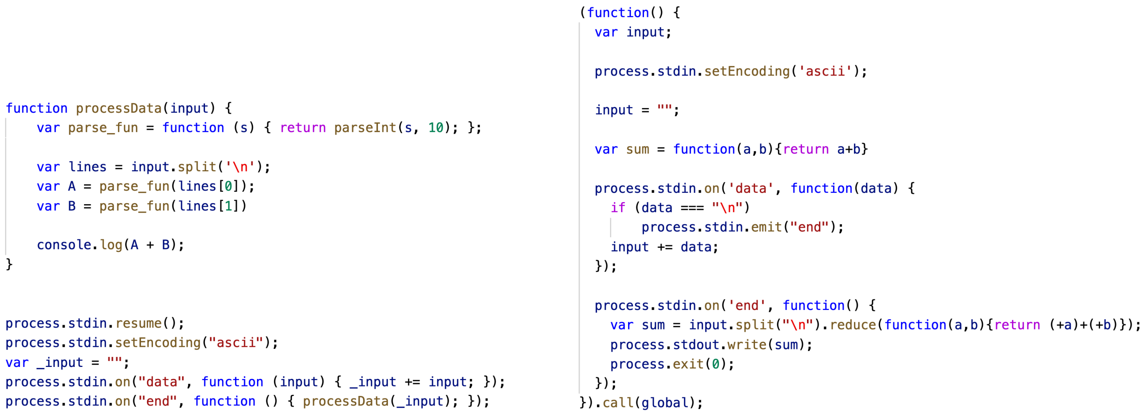
Table 2.2: **Zero-shot code clone detection** with cosine similarity probe. Contrastive and hybrid representations improve clone detection AUROC on unmodified (natural) HackerRank programs by +8% and +10% AUROC over a heuristic textual similarity probe, respectively, suggesting they are predictive of functionality. Contrastive representations are also the most robust to adversarial code transformations.

would not be expected to translate to real code, we find significant improvements in extreme code summarization [6] and type inference [75] tasks.

Clone detection experiments show that contrastive and hybrid representations with our compiler-based augmentations are predictive of program functionality in-the-wild, and that contrastive representations are the most robust to adversarial edits (§2.4.1). Contrastive pre-training outperforms baseline supervised and self-supervised methods on all three tasks (§2.4.1-2.4.3). Finally, ablations suggest it is better to augment unlabeled programs during pre-training rather than augmenting smaller supervised datasets (§2.4.4).

Experimental setup Models are pre-trained on CodeSearchNet, a large corpus of methods extracted from popular GitHub repositories [83]. CodeSearchNet contains 1,843,099 JavaScript programs. Only 81,487 methods have both a documentation string and a method name. The asymmetry between labeled and unlabeled programs stems from JavaScript coding practices where anonymous functions are widespread. The pre-training dataset described in Section 2.3.1 is the result of augmenting all 1.8M programs.

We evaluate two architectures: a 2-layer Bidirectional LSTM with 18M parameters, similar to the supervised model used by Hellendoorn, Bird, Barr, and Allamanis [75], and a 6-layer Transformer with 23M parameters. For a baseline self-supervised approach, we pre-train both architectures with the RoBERTa MLM objective, then transfer it to downstream tasks.



```javascript
function processData(input) {
    var parse_fun = function (s) { return parseInt(s, 10); };

    var lines = input.split('\n');
    var A = parse_fun(lines[0]);
    var B = parse_fun(lines[1])

    console.log(A + B);
}


process.stdin.resume();
process.stdin.setEncoding("ascii");
var _input = "";
process.stdin.on("data", function (input) { _input += input; });
process.stdin.on("end", function () { processData(_input); });
```

```javascript
(function() {
  var input;

  process.stdin.setEncoding('ascii');

  input = "";

  var sum = function(a,b){return a+b}

  process.stdin.on('data', function(data) {
    if (data === "\n")
        process.stdin.emit("end");
    input += data;
  });

  process.stdin.on('end', function() {
    var sum = input.split("\n").reduce(function(a,b){return (+a)+(+b)});
    process.stdout.write(sum);
    process.exit(0);
  });
}).call(global);
```

Figure 2.7: Code clone detection example. These programs solve the same HackerRank coding challenge (reading and summing two integers), but use different coding conventions. The neural code clone detector should classify this pair as a positive, *i.e.* a clone.

2.4.1 Robust Zero-shot Code Clone Detection

ContraCode learns to match variants of programs with similar functionality. While transformations produce highly diverse token sequences (quantified in the supplement), they are artificial and do not change the underlying algorithm. In contrast, human programmers can solve a problem with many data structures, algorithms and programming models. To determine whether pre-trained representations are consistent across programs written by different people, we benchmark *code clone detection*, a binary classification task to detect whether two programs solve the same problem or different ones (Fig. 2.7). This is useful for deduplicating, refactoring and retrieving code, as well as checking approximate code correctness.

Benchmarks exist like BigCloneBench [164], but to the best of our knowledge, there is no benchmark for the JavaScript. We collected 274 in-the-wild JavaScript programs that correctly solve 33 problems from the HackerRank interview preparation website. There are 2065 pairs solving the same problem and 70K pairs solving different problems, which we randomly subsample to 2065 to balance the classes.

Since we probe zero-shot performance based on pre-trained representations, there is no training set. Instead, we threshold cosine similarity of pooled representations of the programs u and v: $u^T v/\|u\|\|v\|$. Many code analysis methods for clone detection measure textual similarity [21]. As a baseline, we threshold the dissimilarity score, a scaled Levenshtein edit distance between normalized and tokenized programs.

Table 2.2 reports the area under the ROC curve (AUROC) and average precision (AP, area under Precision-Recall). All learned representations improve over the heuristic on natural code. Self-supervision through RoBERTa MLM pre-training improves over a randomly initialized network by +1.7% AUROC. Contrastive pre-training achieves +3.4% AUROC over the same baseline. A hybrid objective combining both the contrastive loss and MLM has the best performance with +7.0% AUROC (+5.4% over MLM alone). Although MLM is still useful over natural code, ContraCode learns overall stronger representations of functionality.

However, are these representations robust to code edits? We adversarially edit one program in each pair by applying the loss-maximizing code compression and identifier modification transformation among N samples from Algorithm 1. These transformations preserve program functionality, so ground-truth labels are unchanged. With only 4 edits, RoBERTa underperforms both the heuristic (-5.8% AUROC) and random guessing (50% AUROC), indicating it is highly sensitive to these kinds of implementation details. ContraCode retains much of its performance (+39% AUROC over RoBERTa) as it explicitly optimizes for invariance to code edits. Surprisingly, the hybrid model is less robust than ContraCode alone, perhaps indicating that MLM learns non-robust features [84].

2.4.2 Fine-tuning for Type Inference

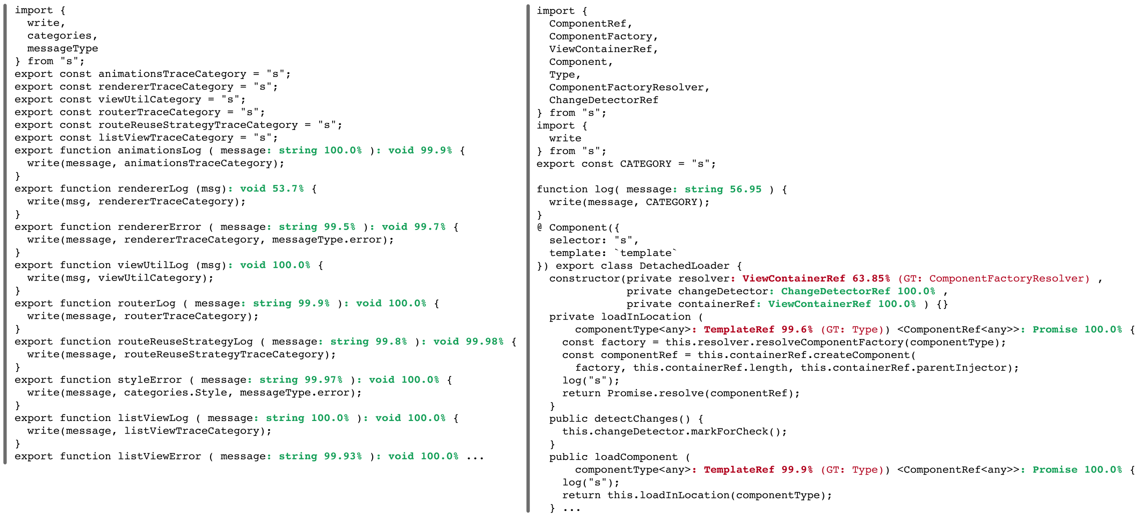
JavaScript is a dynamically typed language, where variable types are determined at runtime based on the values they represent. Manually annotating code with types helps tools flag bugs by detecting incompatible types. Annotations also document code, but are tedious to maintain. Type inference tools automatically predict types from context.

To *learn* to infer types, we use the annotated dataset of TypeScript programs from DeepTyper [75], excluding GitHub repositories that were made private or deleted since publication. The training set contains 15,570 TypeScript files from 187 repositories with 6,902,642 total tokens. Validation and test sets are from held-out repositories. For additional supervision, missing types are inferred by static analysis to augment user-defined types as targets. A 2-layer MLP head predicts types from token embeddings output by the DeepTyper LSTM. We early stop based on validation set top-1 accuracy.

For the rest of our experiments, baseline RoBERTa models are pre-trained on the same *augmented* data as ContraCode for fair comparison. Learning representations that transfer from unlabeled JavaScript programs is challenging because TypeScript supports a superset of JavaScript's grammar, with types annotations and other syntactic sugar that need to be learned during fine-tuning. Further, the pre-training data only has methods while DeepTyper's dataset uses entire files (modules). The model is only given source code for a single file, not dependencies.

Method	**Acc@1**	**Acc@5**
TypeScript CheckJS	45.11%	—
DeepTyper, variable name only	28.94%	70.07%
GPT-3 Codex (zero-shot, 175B)	26.62%	—
GPT-3 Codex (few-shot, 175B)	30.55%	—
Transformer	45.66%	80.08%
+ RoBERTa MLM pre-train	40.85%	75.76%
+ ContraCode pre-train	46.86%	**81.85%**
+ ContraCode + MLM (hybrid)	**47.16%**	81.44%
DeepTyper BiLSTM	51.73%	82.71%
+ RoBERTa MLM pre-train	50.24%	82.85%
+ ContraCode pre-train	**54.01%**	**85.55%**

Table 2.3: **Type inference accuracy on TypeScript programs**. As ContraCode does not modify model architecture, contrastive pre-training improves both BiLSTM and Transformer accuracy (1.5% to 2.28%). Compared with TypeScript's built-in type inference, we improve accuracy by 8.9%.



```
import {
  write,
  categories,
  messageType
} from "s";
export const animationsTraceCategory = "s";
export const rendererTraceCategory = "s";
export const viewUtilCategory = "s";
export const routerTraceCategory = "s";
export const routeReuseStrategyTraceCategory = "s";
export const listViewTraceCategory = "s";
export function animationsLog ( message: string 100.0% ): void 99.9% {
  write(message, animationsTraceCategory);
}
export function rendererLog (msg): void 53.7% {
  write(msg, rendererTraceCategory);
}
export function rendererError ( message: string 99.5% ): void 99.7% {
  write(message, rendererTraceCategory, messageType.error);
}
export function viewUtilLog (msg): void 100.0% {
  write(msg, viewUtilCategory);
}
export function routerLog ( message: string 99.9% ): void 100.0% {
  write(message, routerTraceCategory);
}
export function routeReuseStrategyLog ( message: string 99.8% ): void 99.98% {
  write(message, routeReuseStrategyTraceCategory);
}
export function styleError ( message: string 99.97% ): void 100.0% {
  write(message, categories.Style, messageType.error);
}
export function listViewLog ( message: string 100.0% ): void 100.0% {
  write(message, listViewTraceCategory);
}
export function listViewError ( message: string 99.93% ): void 100.0% ...
```

```
import {
  ComponentRef,
  ComponentFactory,
  ViewContainerRef,
  Component,
  Type,
  ComponentFactoryResolver,
  ChangeDetectorRef
} from "s";
import {
  write
} from "s";
export const CATEGORY = "s";

function log( message: string 56.95 ) {
  write(message, CATEGORY);
}
@ Component({
  selector: "s",
  template: `template`
}) export class DetachedLoader {
  constructor(private resolver: ViewContainerRef 63.85% (GT: ComponentFactoryResolver) ,
              private changeDetector: ChangeDetectorRef 100.0% ,
              private containerRef: ViewContainerRef 100.0% ) {}
  private loadInLocation (
      componentType<any>: TemplateRef 99.6% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    const factory = this.resolver.resolveComponentFactory(componentType);
    const componentRef = this.containerRef.createComponent(
      factory, this.containerRef.length, this.containerRef.parentInjector);
    log("s");
    return Promise.resolve(componentRef);
  }
  public detectChanges() {
    this.changeDetector.markForCheck();
  }
  public loadComponent (
      componentType<any>: TemplateRef 99.9% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    log("s");
    return this.loadInLocation(componentType);
  } ...
```

Figure 2.8: A variant of DeepTyper pre-trained with ContraCode generates type annotations for two held-out programs. The model consistently predicts correct function return types, and often correctly predicts project-specific variable types.

In Table 2.3, contrastive pre-training outperforms all baseline learned methods. ContraCode is applied in a drop-in fashion to each of the baselines. Pre-training with our contrastive objective

Method	**Precision**	**Recall**	**F1**
code2vec	10.78%	8.24%	9.34%
code2seq	12.17%	7.65%	9.39%
RoBERTa MLM	15.13%	11.47%	12.45%
Transformer	18.11%	15.78%	16.86%
+ ContraCode	20.34%	14.96%	**17.24%**

Table 2.4: Results for different settings of **code summarization**: supervised training with 81K functions, masked language model pre-training, training from scratch and contrastive pre-training with fine-tuning.

and data augmentations yields absolute accuracy improvements of +1.2%, +6.3%, +2.3% top-1 and +1.8%, +5.7%, +2.8% top-5 over the Transformer, RoBERTa, and DeepTyper, respectively.

The RoBERTa baseline may perform poorly as the MLM objective, sensitive to local syntactic structure, focuses on token reconstruction or due to available fine-tuning data, termed as weight "ossification" by Hernandez, Kaplan, Henighan, and McCandlish [78]. To combine approaches, we minimized our loss alongside MLM for a hybrid local-global objective, improving accuracy by +6.31% over the RoBERTa Transformer.

We also evaluate the recent GPT-3 Codex model by OpenAI [14] using their API. We benchmark the 175B parameter DaVinci model in both a zero-shot as well as a few-shot prompting setup. Although the Codex model was trained over TypeScript programs, it performs poorly as it achieves an accuracy of 26.6% in the zero-shot setup and 30.6% in the few-shot setup. We only evaluate Top-1 accuracy for GPT-3 models as GPT-3 does not reliably output confidence scores.

Learning outperforms static analysis by a large margin. Overall, our best model has +8.9% higher top-1 accuracy than the built-in TypeScript CheckJS type inference system, showing the promise of learned code analysis. Surfacing multiple candidate types can also be useful to users, while CheckJS only has a single prediction.

Fig. 2.8 shows two files from held-out repositories. For the first, our model consistently predicts the correct return and parameter types. The model correctly predicts that the variable `message` is a string, even though its type is ambiguous without access to the imported `write` method signature. For the second, ContraCode predicts 4 of 8 types correctly including `ViewContainerRef` and `ChangeDetectorRef` from the AngularJS library.

2.4.3 Extreme Code Summarization

The extreme code summarization task asks a model to predict the name of a method given its body [6]. These names often summarize the method, such as `reverseString(...)`. Summarization

```
function x(url, callback, error) {
  var img = new Image();
  img.src = url;
  if(img.complete){
    return callback(img);
  }
  img.onload = function(){
    img.onload = null;
    callback(img);
  };
  img.onerror = function(e){
    img.onerror = null;
    error(e);
  };
}
```

Ground truth: `loadImage`
Prediction: `loadImage`

Top predictions:

1. `getImageItem`
2. `createImage`
3. `loadImageForBreakpoint`
4. `getImageSrcCSS`

Figure 2.9: A held-out JavaScript program from CodeSearchNet and method names generated by a Transformer pre-trained with ContraCode. The correct method name is predicted as the most likely decoding.

models could help programmers interpret poorly documented code. We create a JavaScript summarization dataset using the 81,487 labeled methods in the CodeSearchNet dataset. The name is masked in the method declaration. A sequence-to-sequence model with an autoregressive decoder is trained to maximize log likelihood of the ground-truth name, a form of abstractive summarization. All models overfit, so we stop early according to validation loss. As proposed by Allamanis, Peng, and Sutton [6], we evaluate model predictions by precision, recall and F1 scores over the set of method name tokens.

Table 2.4 shows results in four settings: (1) supervised training using baseline tree-structured architectures that analyze the AST (code2vec, code2seq), (2) pre-training on all 1.8M programs using MLM followed by fine-tuning on the labeled programs (RoBERTa), (3) training a Transformer from scratch and (4) contrastive pre-training followed by fine-tuning with augmentations.

Contrastive pre-training outperforms code2seq by +8.2% test precision, +7.3% recall, and +7.9% F1 score. ContraCode outperforms self-supervised pre-training with RoBERTa by +4.8% F1. ContraCode also achieves slightly higher performance than the Transformer learned from scratch. While this improvement is smaller, code summarization challenging as identifier names are not consistent between programmers.

Figure 2.9 shows a qualitative example of predictions for the code summarization task. The JavaScript method is not seen during training. A Transformer pre-trained with ContraCode predicts the correct method name through beam search. The next four predictions are reasonable, capturing that the method processes an image. The 2nd and 3rd most likely decodings, `getImageItem` and `createImage`, use `get` and `create` as synonyms for `load`, though the final two unlikely decodings include terms not in the method body.

Code summarization model	**F1**
Transformer (Table 2.4)	**16.86**
+ augmentations	15.65

Type inference model	**Acc@1**
Transformer (Table 2.3)	**45.66**
+ augmentations	44.14
DeepTyper (Table 2.3)	**51.73**
+ augmentations	50.33

Table 2.5: Compiler data augmentations degrade performance when training supervised models *from scratch*.

2.4.4 Understanding augmentation importance

We analyze the effect of augmentations on supervised learning and on pre-training.

Supervised learning with augmentations As a baseline, we re-train models from scratch with compiler transforms during *supervised learning* rather than pre-training. Data augmentation artificially expands labeled training sets. For sequence-to-sequence summarization, we apply a variety of augmentations (LS, SW, VR, DCI). These all preserve the method name. For type inference, labels are aligned to input tokens, so they must be realigned after transformation. We only apply token-level transforms (LS, SW) as we can track labels.

Table 2.5 shows results. Compiler-based data augmentations degrade supervised models, perhaps by creating a training distribution not reflective of evaluation programs. However, as shown in §2.4.1–2.4.3, augmenting during ContraCode pre-training yields a more accurate model. Our contrastive learning framework also allows learning over large numbers of unlabeled programs that supervised learning alone cannot leverage. The ablation indicates that augmentations do not suffice, and contrastive learning is important.

Ablating pre-training augmentations Some data augmentations could be more valuable than others. Empirically, pre-training converges faster with a smaller set of augmentations at the same batch size since the positives are syntactically more similar, but this hurts downstream performance. Table 2.6 shows that type inference accuracy degrades when different groups of augmentations are removed. Semantics-preserving code compression passes that require code analysis are the most important, improving top-1 accuracy by 1.95% when included. Line subsampling serves as a

Pre-training augmentations	**Acc@1**	**Acc@5**
All augmentations (Table 2.3)	**52.65%**	**84.60%**
w/o identifier modification (-VR, -IM)	51.94%	84.43%
w/o line subsampling (-LS)	51.05%	81.63%
w/o code compression (-T,C,DCE,CF)	50.69%	81.95%

Table 2.6: Ablating compiler transformations used during contrastive pre-training. The DeepTyper BiLSTM is pre-trained with constrastive learning for 20K steps, then fine-tuned for type inference. Augmentations are only used during pre-training. Each transformation contributes to accuracy.

regularizer, but changes program semantics. LS is relatively less important, but does help accuracy. Identifier modifications preserve semantics, but change useful naming information.

2.5 Conclusion

Large-scale code repositories like GitHub are a powerful resource for learning machine-aided programming tools. However, most current code representation learning approaches need labels, and popular label-free self-supervised methods like RoBERTa are not robust to adversarial inputs. Instead of reconstructing tokens like BERT, learning *what code says*, we learn *what code does*. We propose ContraCode, a contrastive self-supervised algorithm that learns representations invariant to transformations via compiler-based data augmentations. In experiments, ContraCode learns effective representations of functionality, and is robust to adversarial code edits. We find that ContraCode significantly improves performance on three downstream JavaScript code understanding tasks.

Acknowledgments

We thank Lisa Dunlap, Jonathan Ho, Koushik Sen, Rishabh Singh, Aravind Srinivas, Daniel Rothchild, and Justin Wong for helpful feedback. In addition to NSF CISE Expeditions Award CCF-1730628, the NSF GRFP under Grant No. DGE-1752814, and ONR PECASE N000141612723, this research is supported by gifts from Amazon Web Services, Ant Financial, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware.

Chapter 3

Training models beyond memory capacity limits

Rapid scaling of large language models presents a significant challenge. As these models grow in complexity and size, so too does the computational resource demand for their training. One pressing issue is an emerging memory wall where GPU memory capacity is growing more slowly than the size of modern large models.

In "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization", I examine the trade-off between computation and memory in deep neural network training. Popular models often face limitations in memory capacity when dealing with large inputs or complex architectures. To address this issue, I introduce Checkmate, a system designed to optimally manage computation and memory during DNN training.

Since publication, Checkmate has become an important paper in the rematerialization area. Since we published the paper, follow-up work like Dynamic Tensor Rematerialization [101] have generalized the approach in Checkmate to dynamic neural network graphs. I also worked on follow-up work [138] to apply optimal rematerialization and also paging to flash storage to enable training larger models on edge devices.

This work is the result of a collaboration with Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel and Kurt Keutzer.

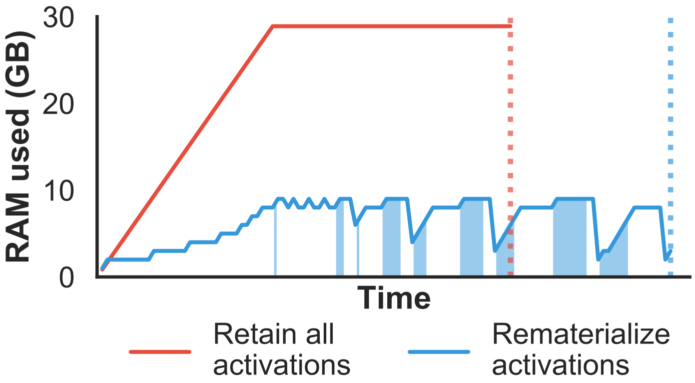


Figure 3.1: This 32-layer deep neural network requires 30GB of memory during training in order to cache forward pass activations for the backward pass. Freeing certain activations early and rematerializing them later reduces memory requirements by 21GB at the cost of a modest runtime increase. Rematerialized layers are denoted as shaded blue regions. We present Checkmate, a system to rematerialize large neural networks *optimally*. Checkmate is hardware-aware, memory-aware and supports arbitrary DAGs.

3.1 Introduction

Deep learning training workloads demand large amounts of high bandwidth memory. Researchers are pushing the memory capacity limits of hardware accelerators such as GPUs by training neural networks on high-resolution images [50, 99, 166], 3D point-clouds [39, 178], and long natural language sequences [41, 48, 169]. In these applications, training memory usage is dominated by the intermediate activation tensors needed for backpropagation (Figure 3.3).

The limited availability of high bandwidth on-device memory creates a *memory wall* that stifles exploration of novel architectures. Across applications, authors of state-of-the-art models cite memory as a limiting factor in deep neural network (DNN) design [36, 41, 47, 62, 74, 106, 115, 140].

Given insufficient RAM to cache all activation tensors for backpropagation, select tensors can be discarded during forward evaluation. When needed for gradient calculation, a discarded tensor can be *rematerialized*. As Figure 3.1 illustrates, rematerializing values allows a large DNN to fit within RAM, albeit with additional computation.

However, their approaches cannot be applied generally to nonlinear DNN structures such as residual connections, and rely on the strong assumption that all nodes in the graph have the same cost. Prior work also assumes that gradients may never be rematerialized. These assumptions limit the efficiency and generality of prior approaches.

Our work formalizes tensor rematerialization as a constrained optimization problem. Utilizing off-the-shelf numerical solvers, we discover optimal rematerialization strategies for arbitrary deep neural networks in TensorFlow with non-uniform computation and memory costs. We demonstrate that optimal rematerialization permits larger batch sizes and substantially reduced memory usage with minimal computational overhead across various image classification and semantic segmentation architectures. Consequently, our approach enables researchers to explore larger models and batch sizes on complex signals with minimal computation overhead.

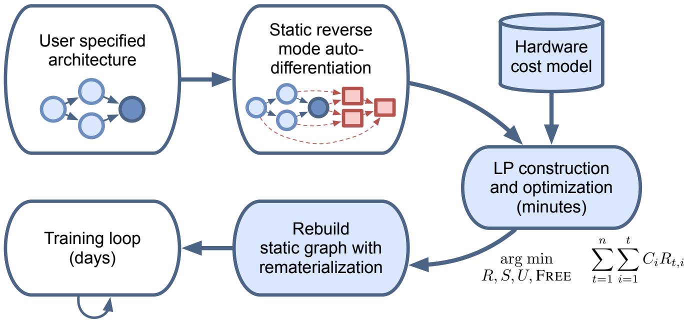

User specified architecture
Static reverse mode auto-differentiation
Hardware cost model
LP construction and optimization (minutes)
Training loop (days)
Rebuild static graph with rematerialization
$\arg\min_{R,S,U,\text{FREE}} \sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i}$

Figure 3.2: Overview of the Checkmate system.

In particular, the contributions of this work include:

- a formalization of the rematerialization problem as a mixed integer linear program with a substantially more flexible search space than prior work, in Section 3.4.7.
- a fast approximation algorithm based on two-phase deterministic LP rounding, in Section 3.5.
- Checkmate, a system implemented in TensorFlow that enables training models with up to 5.1× larger input sizes than prior art at minimal overhead.

3.2 Motivation

While inference optimizations are well studied [88], training workloads have received less attention. Memory consumption during training consists of (a) intermediate features, or activations, whose size depends on input dimensions and (b) parameters and their gradients whose size depends on weight dimensions. Given that inputs are often several order of magnitude larger than kernels, most memory is used by features, demonstrated in Figure 3.3.

Frameworks such as TensorFlow [1] and PyTorch [136, 137] store all activations during the forward pass. Gradients are backpropagated from the loss node, and each activation is freed after its gradient has been calculated. In Figure 3.1, we compare this memory intensive policy and a rematerialization strategy for a real neural network. Memory usage is significantly reduced by deallocating some activations in the forward pass and recomputing them in the backward pass. Our goal is fit an arbitrary network within our memory budget while incurring the minimal additional runtime penalty from recomputation.

Most prior work assumes networks have linear graphs. For example, Chen, Xu, Zhang, and Guestrin [37] divides the computation into $\sqrt{n}$ segments, each with $\sqrt{n}$ nodes. Each segment endpoint is stored during the forward pass. During the backward pass, segments are recomputed in reverse order at $O(n)$ cost.

Linear graph assumptions limit applicability of prior work. For example, while the popular ResNet50 [74] can be linearized by treating each residual block as a single node, this leads to inefficient solutions. For networks with longer skip connections, *e.g.*, U-Net [147], grouping nodes oversimplifies the graph.

Prior work also assumes all layers are equally expensive to recompute. In the VGG19 architecture [156], the largest layer is *six orders of magnitude* more expensive than the smallest layer.

Our work makes few assumptions on neural network graphs. We explore a solution space that allows for (a) arbitrary graphs with several inputs and outputs for each node, (b) variable memory costs across layers and (c) variable computation costs for each layer (such as FLOPs or profiled runtimes). We constrain solutions to simply be *correct* (a node's dependencies must be materialized before it can be evaluated) and *within the RAM budget* (at any point during execution, resident tensors must fit into RAM).

Subject to these constraints, we find solutions that minimize the amount of time it takes to perform a single training iteration. We project schedules into space and time, allowing us to cast the objective as a linear expression. This problem can then be solved using off-the-shelf mixed integer linear program solvers such as *GNU Project - Free Software Foundation (FSF)* [61] or COIN-OR Branch-and-Cut [56]. An optimal solution to the MILP will minimize the amount of additional compute cost within the memory budget.

3.3 Related Work

We categorize related work as checkpointing, reversible networks, distributed computation, and activation compression.

Checkpointing and rematerialization Chen, Xu, Zhang, and Guestrin [37] propose a heuristic for checkpointing unit-cost linear n-layer graphs with $O(\sqrt{n})$ memory usage. Griewank and Walther [67] checkpoint similar graphs with $O(\log n)$ memory usage and prove optimality

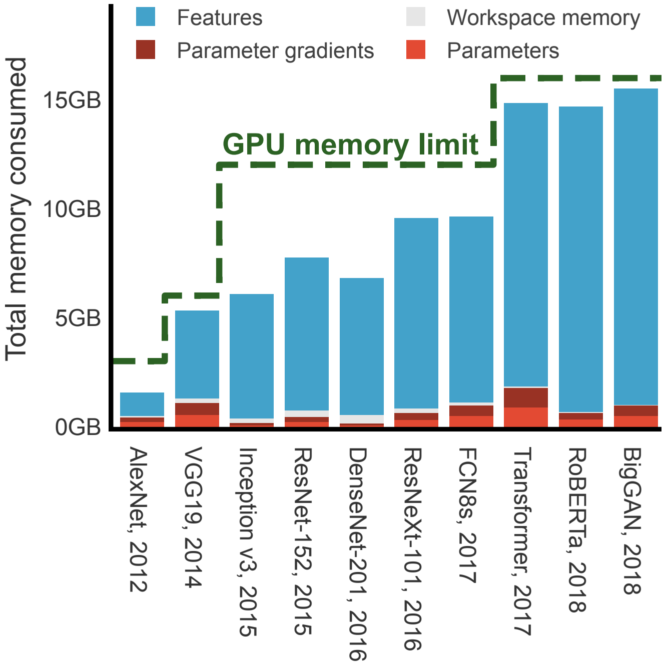


Figure 3.3: Memory consumed by activations far outweigh parameters for popular model architectures. Moreover, advances in GPU DRAM capacity are quickly utilized by researchers; the dashed line notes the memory limit of the GPU used to train each model.

for linear graphs with unit per-node cost and memory. However, real-world DNN layers vary significantly in memory usage and computational cost [165], so these heuristics are not always optimal. Chen, Xu, Zhang, and Guestrin [37] develop a greedy algorithm checkpointing network layers in roughly equal memory segments, defined by hyperparameter *b*. Yet, neither procedure is cost-aware nor deallocates checkpoints when possible. Gruslys, Munos, Danihelka, Lanctot, and Graves [68] introduce a dynamic programming algorithm for checkpoint selection in RNN training. Feng and Huang [52] provide a dynamic program for selecting checkpoints in branching networks, but overlook layer costs and memory usage. Siskind and Pearlmutter [159] propose a divide-and-conquer strategy in programs, while Beaumont, Herrmann, Pallez, and Shilova [23] employ dynamic programming for checkpoint selection for DNNs with joining sub-networks.

Intermediate value recomputation is also common in register allocation. Compiler backends lower an intermediate representation of code to an architecture-specific executable binary. During lowering, an abstract *static single assignment* (SSA) graph of values and operations [46, 148] is concretized by mapping values to a finite number of registers. If insufficient registers are available

Method	Description	General graphs	Cost aware	Memory aware
Checkpoint all (Ideal)	No rematerialization. Default in deep learning frameworks.	√	×	×
Griewank et al. log n	Griewank and Walther [67] REVOLVE procedure	×	×	×
Chen et al. $\sqrt{n}$	Chen, Xu, Zhang, and Guestrin [37] checkpointing heuristic	×	×	×
Chen et al. greedy	Chen, Xu, Zhang, and Guestrin [37], with search over parameter b	×	×	~
AP $\sqrt{n}$	Chen et al. $\sqrt{n}$ on articulation points + optimal R solve	~	×	×
AP greedy	Chen et al. greedy on articulation points + optimal R solve	~	×	~
Linearized $\sqrt{n}$	Chen et al. $\sqrt{n}$ on topological sort + optimal R solve	√	×	×
Linearized greedy	Chen et al. greedy on topological sort + optimal R solve	√	×	~
Checkmate ILP	Our ILP as formulated in Section 3.4	√	√	√
Checkmate approx.	Our LP rounding approximation algorithm (Section 3.5)	√	√	√

Table 3.1: Rematerialization baselines and our extensions to make them applicable to non-linear architectures

for an SSA form computation graph, values are *spilled* to main memory by storing and later loading the value. Register allocation has been formulated as graph coloring problem [34], integer program [63, 116], and network flow [102].

Register allocators may recompute constants and values with register-resident dependencies if the cost of doing so is less than the cost of a spill [29, 34, 144]. While similar to our setup, register rematerialization is limited to exceptional values that can be recomputed in a single instruction with dependencies already in registers. For example, memory offset computations can be cheaply recomputed, and loads of constants can be statically resolved. In contrast, Checkmate can recompute entire subgraphs of the program's data-flow.

During the evaluation of a single kernel, GPUs spill per-thread registers to a thread-local region of global memory (*i.e.* local memory) [124, 130]. NN training executes DAGs of kernels and stores intermediate values in shared global memory. This produces a high range of value sizes, from 4 byte floats to gigabyte tensors, whereas CPU and GPU registers range from 1 to 64 bytes. Our problem of interkernel memory scheduling thus differs in scale from the classical problem of register allocation within a kernel or program. Rematerialization is more appropriate than copying values out of core as the cost of spilling values from global GPU memory to main memory (RAM) is substantial [87, 124], though possible [123].

Reversible Networks Gomez, Ren, Urtasun, and Grosse [62] propose a reversible (approximately invertible) residual DNN architecture, where intermediate temporary values can be recomputed from values derived *later* in the standard forward computation. Reversibility enables recomputation during the backward pass. Bulo, Porzi, and Kontschieder [30] replace only ReLU and batch normalization layers with invertible variants and reduce memory usage up to 50%. We find rematerialization enables greater savings and a wider range of budgets, but reversibility is a promising complementary approach.

Distributed computation Orthogonal approaches to address the limited memory problem are distributed-memory computations and gradient accumulation. However, model parallelism requires access to additional expensive compute accelerators, fast networks, and non-trivial partitioning of model state to balance communication and computation [60, 92, 120]. Gradient accumulation enables larger batch sizes by computing the gradients with multiple sub-batches across a mini-batch. However, gradient accumulation can degrade performance as batch normalization performs poorly on small batch sizes [85, 176].

Activation compression In some DNN applications, it is possible to process compressed representations with minimal accuracy loss. Gueguen, Sergeev, Kadlec, Liu, and Yosinski [69] classify discrete cosine transforms of JPEG images rather than raw images. Jain, Phanishayee, Mars, Tang, and Pekhimenko [87] quantize activations, cutting memory usage in half. Compression reduces memory usage by a constant factor, but reduces accuracy. Our approach is mathematically equivalent to rematerialization-free training and incurs no accuracy penalty.

3.4 Optimal Rematerialization

In this section, we develop an optimal solver that schedules computation and garbage collection during the evaluation of general data-flow graphs including those used in DNN training. Our proposed scheduler minimizes computation or execution time while guaranteeing that the schedule will not exceed device memory limitations. The rematerialization problem is formulated as a mixed integer linear program (MILP) that can be solved with commercial or open-source solvers.

3.4.1 Problem definition

A computation or data-flow graph $G = (V, E)$ is a directed acyclic graph with n nodes $V = \{v_1, \ldots, v_n\}$ that represent operations yielding values (*e.g.* tensors). Edges represent dependencies between operators, such as layer inputs in a neural network. Nodes are numbered according to a topological order, such that operation v_j may only depend on the results of operations $v_{i<j}$.

Each operator's output takes M_v memory to store and costs C_v to compute from its inputs. We wish to find the terminal node v_n with peak memory consumption under a memory budget, M_{budget}, and minimum total cost of computation.

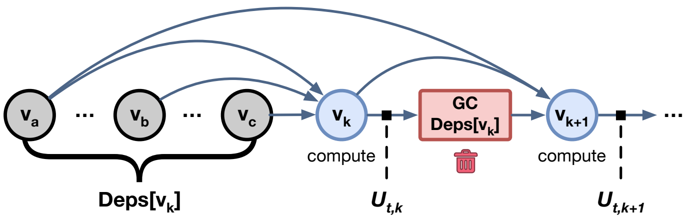


Figure 3.4: Dependencies of v_k can only be garbage collected after it is evaluated. $U_{t,k}$ measures the memory used after evaluating v_k and before deallocating its dependencies. v_b and v_c may be deallocated during garbage collection, but v_a may not due to a forward edge.

3.4.2 Representing a schedule

We define a schedule as a series of nodes being saved or (re)computed. The network execution unrolls into T stages, allowing a node to be computed once per stage. $S_{t,i} \in 0, 1$ signifies the retention of operation i result in memory at stage $t-1$ until stage t, while $R_{t,i} \in 0, 1$ determines if operation i is recomputed at time step t.

Our representation generalizes checkpointing [37, 52, 67, 68, 158], with the ability to retain and deallocate values multiple times, at a cost of $O(Tn)$ decision variables. To balance decision variables number and schedule flexibility, we set $T = n$, which permits $O(n^2)$ operations and constant memory in linear graphs.

3.4.3 Scheduling with ample memory

In neural network evaluation with ample memory, our solver ensures checkpointed and computed operations have resident dependencies. Minimizing total computation cost across stages with these constraints yields objective (3.1a):

$$\underset{R,\,S}{\arg\min} \quad \sum_{t=1}^{n} \sum_{i=1}^{t} C_i R_{t,i} \tag{3.1a}$$

subject to

$$R_{t,j} \leq R_{t,i} + S_{t,i} \qquad \forall t\ \forall (v_i, v_j) \in E, \tag{3.1b}$$

$$S_{t,i} \leq R_{t-1,i} + S_{t-1,i} \qquad \forall t \geq 2\ \forall i, \tag{3.1c}$$

$$\textstyle\sum_i S_{1,i} = 0, \tag{3.1d}$$

$$\textstyle\sum_t R_{t,n} \geq 1, \tag{3.1e}$$

$$R_{t,i}, S_{t,i} \in \{0, 1\} \qquad \forall t\ \forall i \tag{3.1f}$$

Constraints ensure feasibility and completion. Constraint (3.1b) and (3.1c) ensure that an operation is computed in stage t only if all dependencies are available. To cover the edge case of the first stage, constraint (3.1d) specifies that no values are initially in memory. Finally, covering constraint (3.1e) ensures that the last node in the topological order is computed at some point in the schedule so that training progresses.

3.4.4 Constraining memory utilization

To constrain memory usage, we introduce memory accounting variables $U_{t,k} \in \mathbb{R}_+$ into the ILP. Let $U_{t,k}$ denote the memory used just after computing node v_k in stage t. $U_{t,k}$ is defined recursively in terms of auxiliary binary variables $\textsc{Free}_{t,i,k}$ for $(v_i, v_k) \in E$, which specifies whether node v_i may be deallocated in stage t after evaluating node v_k.

We assume that (1) network inputs and parameters are always resident in memory and (2) enough space is allocated for gradients of the loss with respect to parameters.[*] Parameter gradients are typically small, the same size as the parameters themselves. Additionally, at the beginning of a stage, all checkpointed values are resident in memory. Hence, we initialize the recurrence,

$$U_{t,0} = \underbrace{M_{\text{input}} + 2M_{\text{param}}}_{\text{Constant overhead}} + \sum_{i=1}^{n} \underbrace{M_i S_{t,i}}_{\text{Checkpoints}} \tag{3.2}$$

Suppose $U_{t,k}$ bytes of memory are in use after evaluating v_k. Before evaluating v_{k+1}, v_k and dependencies (parents) of v_k may be deallocated if there are no future uses. Then, an output tensor for the result of v_{k+1} is allocated, consuming memory M_{k+1}. The timeline is depicted in Figure 3.4, yielding recurrence (3.3):

$$U_{t,k+1} = U_{t,k} - \text{mem\_freed}_t(v_k) + R_{t,k+1} M_{k+1}, \tag{3.3}$$

where $\text{mem\_freed}_t(v_k)$ is the memory freed by deallocating v_k and its parents at stage t. Let

$$\begin{aligned} \textsc{Deps}[k] &= \{i : (v_i, v_k) \in E\}, \text{ and} \\ \textsc{Users}[i] &= \{j : (v_i, v_j) \in E\} \end{aligned}$$

[*] While gradients can be deleted after updating parameters, we reserve constant space since many parameter optimizers such as SGD with momentum maintain gradient statistics.

denote parents and children of a node, respectively. Then, in terms of auxiliary variable $\textsc{Free}_{t,i,k}$, for $(v_i, v_k) \in E$,

$$\text{mem\_freed}_t(v_k) = \sum_{\substack{i \in \textsc{Deps}[k] \\ \cup\{k\}}} M_i * \textsc{Free}_{t,i,k}, \text{ and} \tag{3.4}$$

$$\textsc{Free}_{t,i,k} = R_{t,k} * \underbrace{(1 - S_{t+1,i})}_{\text{Not checkpoint}} \prod_{\substack{j \in \textsc{Users}[i] \\ j > k}} \underbrace{(1 - R_{t,j})}_{\text{Not dep.}} \tag{3.5}$$

The second factor in (3.5) ensures that M_i bytes are freed only if v_i is not checkpointed for the next stage. The final factors ensure that $\textsc{Free}_{t,i,k} = 0$ if any child of v_i is computed in the stage, since then v_i needs to be retained for later use. Multiplying by $R_{t,k}$ in (3.5) ensures that values are only freed at most once per stage according to Theorem 3.4.1,

Theorem 3.4.1 (No double deallocation). If (3.5) holds for all $(v_i, v_k) \in E$, then $\sum_{k \in \textsc{Users}[i]} \textsc{Free}_{t,i,k} \leq 1 \; \forall t, i$.

Proof. Assume for the sake of contradiction that $\exists k_1, k_2 \in \textsc{Users}[i]$ such that $\textsc{Free}_{t,i,k_1} = \textsc{Free}_{t,i,k_2} = 1$. By the first factor in (3.5), we must have $R_{t,k_1} = R_{t,k_2} = 1$. Assume without loss of generality that $k_2 > k_1$. By the final factor in (3.5), we have $\textsc{Free}_{t,i,k_1} \leq 1 - R_{t,k_2} = 0$, which is a contradiction. □

3.4.5 Linear reformulation of memory constraint

While the recurrence (3.2-3.3) defining U is linear, the right hand size of (3.5) is a polynomial. To express $\textsc{Free}$ in our ILP, it must be defined via linear constraints. We rely on Lemma 3.4.1 and 3.4.2 to reformulate (3.5) into a tractable form.

Lemma 3.4.1 (Linear Reformulation of Binary Polynomial). *If* $x_1, \ldots, x_n \in \{0, 1\}$*, then*

$$\prod_{i=1}^{n} x_i = \begin{cases} 1 & \sum_{i=1}^{n}(1 - x_i) = 0 \\ 0 & \textit{otherwise} \end{cases}$$

Proof. If all $x_1, \ldots, x_n = 1$, then $\sum_{i=1}^{n}(1 - x_i) = 0$ and we have $\Pi_{i=1}^{n} x_i = 1$. If otherwise any $x_j = 0$, then we have $\Pi_{i=1}^{n} x_i = 0$, as desired. This can also be seen as an application of De Morgan's laws for boolean arithmetic. □

Lemma 3.4.2 (Linear Reformulation of Indicator Constraints). *Given* $0 \le y \le \kappa$ *where* y *is integral and* κ *is a constant upper bound on* y*, then*

$$x = \begin{cases} 1 & y = 0 \\ 0 & \textit{otherwise} \end{cases}$$

if and only if $x \in \{0, 1\}$ *and* $(1 - x) \le y \le \kappa(1 - x)$.

Proof. For the forward direction, first note that by construction, $x \in \{0, 1\}$. If $y = 0$ and $x = 1$, then $(1 - x) = 0 \le y \le 0 = \kappa(1 - x)$. Similarly, if $y \ge 1$ and $x = 0$, then $1 \le y \le \kappa$, which is true since $0 \le y \le \kappa$ and y is integral. The converse holds similarly. □

To reformulate Constraint 3.5, let num_hazards(t, i, k) be the number of zero factors on the RHS of the constraint. This is a linear function of the decision variables,

$$\text{num\_hazards}(t, i, k) = (1 - R_{t,k}) + S_{t+1,i} + \sum_{\substack{j \in \textsc{Users}[i] \\ j > k}} R_{t,j}$$

Applying Lemma 3.4.1 to the polynomial constraint, we have,

$$\textsc{Free}_{t,i,k} = \begin{cases} 1 & \text{num\_hazards}(t, i, k) = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

By Lemma 3.4.2, if κ is the maximum value that num_hazards(t, i, k) can assume, the following constraints are equivalent to (3.6),

$$\textsc{Free}_{t,i,k} \in \{0, 1\} \tag{3.7a}$$

$$1 - \textsc{Free}_{t,i,k} \le \text{num\_hazards}(t, i, k) \tag{3.7b}$$

$$\kappa(1 - \textsc{Free}_{t,i,k}) \ge \text{num\_hazards}(t, i, k) \tag{3.7c}$$

3.4.6 Tractability via frontier-advancing stages

Fixing the execution order of nodes in the graph can improve the running time of the algorithm. In eager-execution frameworks such as PyTorch, the order is given by user code and operations are executed serially. Separating ordering and allocation is common in compiler design, and both LLVM [113] and GCC [132] have separate instruction scheduling and register allocation passes.

Any topological order of the nodes is a possible execution order. Given a topological order, such as the one introduced in Section 3.4.1, we partition the schedule into frontier-advancing stages such that node v_i is evaluated for the first time in stage i. We replace constraints (3.1d, 3.1e) that ensure the last node is computed with stricter constraints (3.8a-3.8c),

$$R_{i,i} = 1 \;\; \forall i \quad \text{(frontier-advancing partitions)} \tag{3.8a}$$

$$\textstyle\sum_{i \geq t} S_{t,i} = 0 \quad \text{(lower tri., no initial checkpoints)} \tag{3.8b}$$

$$\textstyle\sum_{i > t} R_{t,i} = 0 \quad \text{(lower triangular)} \tag{3.8c}$$

This reduces the feasible set, constraining the search space and improving running time. For an 8 layer ($n = 17$) linear graph neural network with unit C_i, M_i at a memory budget of 4, Gurobi optimizes the unpartitioned MILP in 9.4 hours and the partitioned MILP in 0.23 seconds to the same objective.

3.4.7 Complete Integer Linear Program formulation

The complete memory constrained MILP follows in (3.9), with $O(|V||E|)$ variables and constraints.

$$\begin{aligned} \underset{R, S, U, \textsc{Free}}{\text{arg min}} \quad & \sum_{t=1}^{n} \sum_{i=1}^{t} C_i R_{t,i} \\ \text{subject to} \quad & (3.1b), (3.1c), (3.1f), (3.2), (3.3), \\ & (3.7a), (3.7b), (3.7c), (3.8a), (3.8b), (3.8c), \\ & U_{t,k} \leq M_{\text{budget}} \end{aligned} \tag{3.9}$$

3.4.8 Constraints implied by optimality

Problem 3.9 can be simplified by removing constraints implied by optimality of a solution. $\textsc{Free}_{t,k,k} = 1$ only if operation k is spuriously evaluated with no uses of the result. Hence, the solver can set $R_{t,k} = 0$ to reduce cost. We eliminate $|V|^2$ variables $\textsc{Free}_{t,k,k}$, assumed to be 0, by modifying (3.4) to only sum over $i \in \textsc{Deps}[k]$. These variables can be computed inexpensively after solving.

3.4.9 Generating an execution plan

Given a feasible solution to (3.9), $(R, S, U, \textsc{Free})$, Algorithm 2 generates an execution plan via a row major scan of R and S with deallocations determined by $\textsc{Free}$. An execution plan is a program

Algorithm 2 Generate execution plan

```
Input: graph $G = (V, E)$, feasible $(R, S, \textsc{Free})$
Output: execution plan $P = (s_1, \ldots, s_k)$
Initialize $\textsc{Regs}[1 \ldots |V|] = -1$, $r = 0$, $P = ()$.
for $t = 1$ to $|V|$ do
  for $k = 1$ to $|V|$ do
    if $R_{t,k}$ then
      // Materialize $v_k$
      add %r = compute v_k to $P$
      $\textsc{Regs}[k] = r$
      $r = r + 1$
    end if
    // Free $v_k$ and dependencies
    for $i \in \textsc{Deps}[k] \cup \{k\}$ do
      if $\textsc{Free}_{t,i,k}$ then
        add deallocate %Regs[i] to $P$
      end if
    end for
  end for
end for
return $P$
```

$P = (s_1, \ldots, s_k)$ with k statements. When statement `%r = compute v` is interpreted, operation v is evaluated. The symbol `%r` denotes a virtual register used to track the resulting value. Statement `deallocate %r` marks the value tracked by virtual register `%r` for garbage collection.

The execution plan generated by Algorithm 2 is further optimized by moving deallocations earlier in the plan when possible. Spurious checkpoints that are unused in a stage can be deallocated at the start of the stage rather than during the stage. Still, this code motion is unnecessary for feasibility as the solver guarantees that the unoptimized schedule will not exceed the desired memory budget.

The execution plan can either be interpreted during training, or encoded as a static computation graph. In this work, we generate a static graph $G' = (V', E')$ from the plan, which is executed by a numerical machine learning framework. See Section 3.6.2 for implementation details.

3.4.10 Cost model

To estimate the runtime of a training iteration under a rematerialization plan, we apply an additive cost model (3.1a), incurring cost C_i when node v_i is evaluated. Costs are determined prior to MILP construction by profiling network layers on target hardware with random inputs across a range of batch sizes and input shapes, and exclude static graph construction and input generation time. As neural network operations consist of dense numerical kernels such as matrix multiplication, these runtimes are low variance and largely independent of the specific input data [91, 162]. However, forward pass time per batch item decreases with increasing batch size due to improved data parallelism [32], so it is important to compute costs with appropriate input dimensions.

The memory consumption of each value in the data-flow graph is computed statically as input and output sizes are known. Values are dense, multi-dimensional tensors stored at 4 byte floating point precision. The computed consumption M_i is used to construct memory constraints (3.2-3.3).

3.5 Approximation

Many of our benchmark problem instances are tractable to solve using off-the-shelf integer linear program solvers, with practical solve times ranging from seconds to an hour. ILP results in this paper are obtained with a 1 hour time limit on a computer with at least 24 cores. Relative to training time, *e.g.* 21 days for the BERT model [48], solving the ILP adds less than a percent of runtime overhead.

While COTS solvers such as COIN-OR [56] leverage methods like branch-and-bound to aggressively prune the decision space, they can take superpolynomial time in the worst-case and solving ILPs is NP-hard in general. In the worst-case, for neural network architectures with hundreds of layers, it is not feasible to solve the rematerialization problem via our ILP. An instance of the VGG16 architecture [156] takes seconds to solve. For DenseNet161 [81], no feasible solution was found within one day.

For many classical NP-hard problems, approximation algorithms give solutions close to optimal with polynomial runtime. We review a linear program that produces fractional solutions in polynomial time in Section 3.5.1. Using the fractional solutions, we present a two-phase rounding algorithm in Section 3.5.2 that rounds a subset of the decision variables, then finds a minimum cost, feasible setting of the remaining variables to find near-optimal integral solutions.

3.5.1 Relaxing integrality constraints

By relaxing integrality constraints (3.1f), the problem becomes trivial to solve as it is a linear program over continuous variables. It is well known that an LP is solvable in polynomial time via Karmarkar's algorithm [97] or barrier methods [129]. With relaxation $R, S, \textsc{Free} \in [0, 1]$, the objective (3.1a) defines a lower-bound for the cost of the optimal integral solution.

Algorithm 3 Two-phase rounding

```
Input: Fractional checkpoint matrix $S^*$ from LP
Output: Binary $S^{\text{int}}$, $R^{\text{int}}$, Free
Round $S^*$ deterministically: $S^{\text{int}}_{t,i} \leftarrow \mathbb{1}[S^*_{t,i} > 0.5]$
$R^{\text{int}} \leftarrow \mathbf{I}_n$ thereby satisfying (3.8a)
while $\exists t \geq 2, i \in [n]$ such that $S^{\text{int}}_{t,i} > R^{\text{int}}_{t-1,i} + S^{\text{int}}_{t-1,i}$ i.e. (3.1c) violated do
    Compute $v_i$ to materialize checkpoint: $R^{\text{int}}_{t-1,i} \leftarrow 1$
end while
while $\exists t \geq 1, (i, j) \in E$ such that $R^{\text{int}}_{t,j} > R^{\text{int}}_{t,i} + S^{\text{int}}_{t,i}$ i.e. (3.1b) violated do
    Compute $v_i$ as temporary for dependency: $R^{\text{int}}_{t,i} \leftarrow 1$
end while
Evaluate Free by simulating execution
return $S^{\text{int}}$, $R^{\text{int}}$, Free
```

Rounding is a common approach to find approximate integral solutions given the result of an LP relaxation. For example, one can achieve a $\frac{3}{4}$-approximation for MAX SAT [179] via a simple combination of randomized rounding ($\mathbf{Pr}\left[x^{\text{int}}_i = 1\right] = x^*_i$) and deterministic rounding ($x^{\text{int}}_i = 1$ if $x^*_i \geq p$, where commonly $p = 0.5$).

We attempt to round the fractional solution R^*, S^* using these two strategies, and then apply Algorithm 2 to $R^{\text{int}}, S^{\text{int}}$. However, direct application of deterministic rounding returns infeasible results: the rounded solution violates constraints. Randomized rounding may show more promise as a single relaxed solution can be used to sample many integral solutions, some of which are hopefully feasible. Unfortunately, using randomized rounding with the LP relaxation for VGG16 at a 4× smaller budget than default, we could not find a single feasible solution out of 50,000 samples.

3.5.2 A two-phase rounding strategy

To find feasible solutions, we introduce *two-phase rounding*, detailed in Algorithm 3. Two-phase rounding is applicable when a subset of variables can be solved in polynomial time given the remaining variables. Our approximation algorithm only rounds the checkpoint matrix S^*. Given S^*, we solve for the conditionally optimal binary computation matrix R^{int} by setting as few values to 1 as possible. Algorithm 3 begins with an all-zero matrix $R^{\text{int}} = 0$, then iteratively corrects violated correctness constraints.

Note that during any of the above steps, once we set some $R^{\text{int}}_{i,j} = 1$, the variable is never changed. Algorithm 3 corrects constraints in a particular order so that constraints that are satisfied will continue to be satisfied as other violated constraints are corrected. The matrix R^{int} generated by

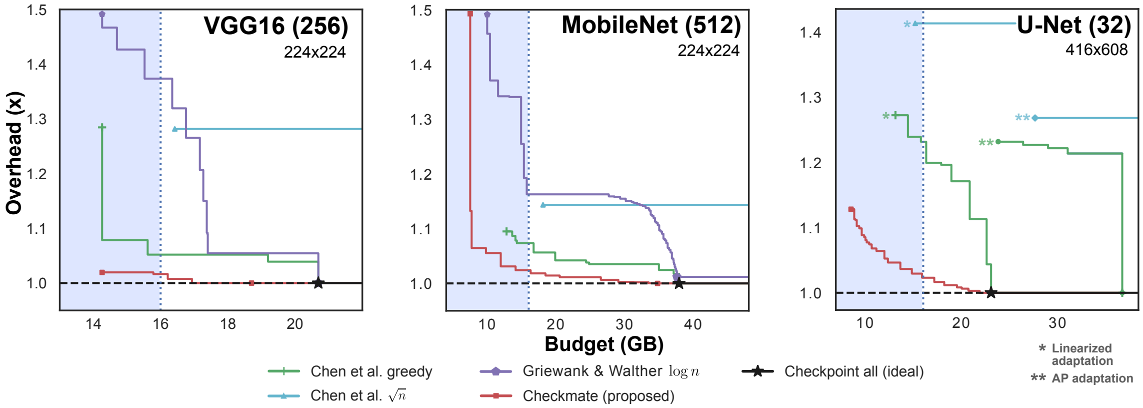


Figure 3.5: Computational overhead versus memory budget for (a) VGG16 image classification NN [156], (b) MobileNet image classification NN, and (c) the U-Net semantic segmentation NN [147]. Overhead is with respect to the best possible strategy without a memory restriction based on a profile-based cost model of a single NVIDIA V100 GPU. For U-Net (c), at the 16 GB V100 memory budget, we achieve a 1.20× speedup over the best baseline—linearized greedy—and a 1.38× speedup over the next best—linearized $\sqrt{n}$. **Takeaway:** our model- and hardware-aware solver produces in-budget solutions with the lowest overhead on linear networks (a-b), and dramatically lowers memory consumption *and* overhead on complex architectures (c).

this rounding scheme will be optimal up to the choice of S^{int} as every entry in R^{int} is set to 1 if and only if it is necessary to satisfy a constraint. In implementation, we detect and correct violations of (3.1b) in reverse topological order for each stage, scanning $R^{\text{int}}, S^{\text{int}}$ matrices from right to left.

3.5.3 Memory budget feasibility

Since we approximate S by rounding the fractional solution, $S^{\text{int}}, R^{\text{int}}$ can be infeasible by the budget constraint $U_{t,k} \leq M_{\text{budget}}$. While the fractional solution may come under the budget and two-phase rounding preserves correctness constraints, the rounding procedure makes no attempt to maintain budget feasibility. Therefore, we leave an allowance on the total memory budget constraint ($U_{t,k} \leq (1-\epsilon)M_{\text{budget}}$). We empirically find $\epsilon = 0.1$ to work well.

3.6 Evaluation

In this section, we investigate the impact of tensor rematerialization on the cost and memory usage of DNN training. We study the following experimental questions: (1) *What is the trade-off between memory usage and computational overhead when using rematerialization?* (2) *Are large inputs* *practical with rematerialization?* and (3) *How well can we approximate the optimal rematerialization policy?*

We compare our proposed solver against baseline heuristics on representative image classification and high resolution semantic segmentation models including VGG16, VGG19, ResNet50, MobileNet, U-Net and FCN with VGG layers, and SegNet. As prior work is largely limited to linear graphs, we propose novel extensions where necessary for comparison. Results show that optimal rematerialization allows significantly lower computational overhead than baselines at all memory budgets, and lower memory usage than previously possible. As a consequence, optimal rematerialization allows training with larger input sizes than previously possible, up to 5.1× higher batch sizes on the same accelerator. Finally, we find that our two-phase rounding approximation algorithm finds near-optimal solutions in polynomial time.

3.6.1 Baselines and generalizations

Table 3.1 summarizes baseline rematerialization strategies. The nominal evaluation strategy stores all features generated during the forward pass for use during the backward pass—this is the default in frameworks such as TensorFlow. Hence, every layer is computed once. We refer to this baseline as *Checkpoint all*, an ideal approach given ample memory.

On the linear graph architectures, such as VGG16 and MobileNet (v1), we directly apply prior work from Griewank and Walther [67] and Chen, Xu, Zhang, and Guestrin [37], baselines referred to as *Griewank and Walther* $\log n$, *Chen et al.* $\sqrt{n}$ and *Chen et al. greedy*. To build a tradeoff curve for computation versus memory budget, we search over the segment size hyperparameter b in the greedy strategy. However, these baselines cannot be used for modern architectures with residual connections. For a fair comparison, we extend the $\sqrt{n}$ and greedy algorithms to apply to general computation graphs (*e.g.* ResNet50 and U-Net).

Chen, Xu, Zhang, and Guestrin [37] suggests manually annotating good checkpointing candidates in a computation graph. For the first extensions, denoted by *AP* $\sqrt{n}$ and *AP greedy*, we automatically identify *articulation points*, or *cut vertices*, vertices that disconnect the forward pass DAG, and use these as candidates. The heuristics then select a subset of these candidates, and we work backwards from the checkpoints to identify which nodes require recomputation.

Still, some networks have few articulation points, including U-Net. We also extend heuristics by treating the original graph as a linear network, with nodes connected in topological order, again backing out the minimal recomputations from the selected checkpoints. These extensions are referred to as *Linearized* $\sqrt{n}$ and *Linearized greedy*.

3.6.2 Evaluation setup

Checkmate is implemented in Tensorflow 2.0 [1], accepting user-defined models expressed via the high-level Keras interface. We extract the forward and backward computation graph, then construct and solve optimization problem (3.9) with the Gurobi mathematical programming library as an integer linear program. Finally, Checkmate translates solutions into execution plans and constructs a new static training graph. Together, these components form the Checkmate system, illustrated in Figure 3.2.

To accelerate problem construction, decision variables R and S are expressed as lower triangular matrices, as are accounting variables U. Free is represented as a $|V| \times |E|$ matrix. Except for our maximum batch size experiments, solutions are generated with a user-configurable time limit of 3600 seconds, though the majority of problems solve within minutes. Problems with exceptionally large batch sizes or heavily constrained memory budgets may reach this time limit while the solver attempts to prove that the problem is infeasible. The cost of a solution is measured with a profile-based cost model (Section 3.4.10) and compared to the ideal, unachievable cost with no recomputation.

The feasible set of our optimal ILP formulation is a superset of baseline heuristics. We implement baselines as a static policy for the decision variable S and then solve for the lowest-cost recomputation schedule using a similar procedure to that described in Algorithm 3.

3.6.3 What is the trade-off between memory usage and computational overhead?

Figure 3.5 compares remateralization strategies on VGG-16, MobileNet, and U-Net. The y-axis shows the computational overhead of checkpointing in terms of time as compared to baseline. The time is computed by profiling each individual layer of the network. The x-axis shows the total memory budget required to run each model with the specified batch size, computed for single precision training. Except for the $\sqrt{n}$ heuristics, each rematerialization algorithm has a knob to trade-off the amount of recomputation and memory usage, where a smaller memory budget leads to higher overhead.

Takeaways: For all three DNNs, Checkmate produces clearly faster execution plans as compared to algorithms proposed by Chen, Xu, Zhang, and Guestrin [37] and Griewank and Walther [67] – over 1.2× faster than the next best on U-Net at the NVIDIA V100 memory budget. Our framework allows training a U-Net at a batch size of 32 images per GPU with less than 10% higher overhead. This would require 23 GB of memory without rematerialization, or with the original baselines without our generalizations.

3.6.4 Are large inputs practical with rematerialization?

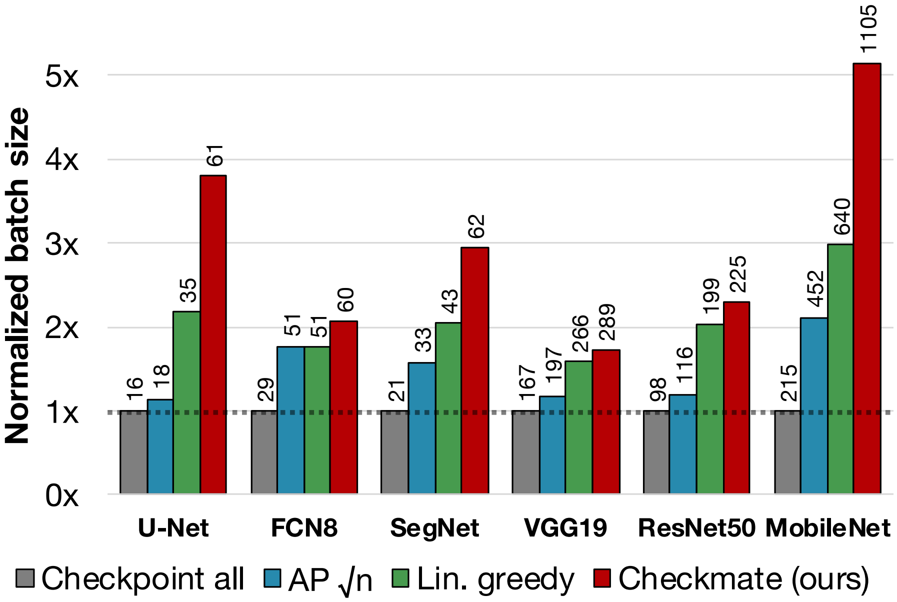


Figure 3.6: Maximum batch size possible on a single NVIDIA V100 GPU when using different generalized rematerialization strategies with at most a single extra forward pass. We enable increasing batch size by up to 5.1× over the current practice of caching all activations (on MobileNet), and up to 1.73× over the best baseline (on U-Net).

The maximum batch size enabled by different rematerialization strategies is shown in Figure 3.6. The y-axis shows the theoretical maximum batch size we could feasibly train with bounded compute cost. This is calculated by enforcing that the total cost must be less than the cost of performing just one additional forward pass. That is, in Figure 3.6 the cost is at most an additional forward pass higher, *if* the specified batch size would have fit in GPU memory. To find Checkmate's maximum batch size, we reformulate Problem (3.9) to maximize a batch size variable $B \in \mathbb{N}$ subject to modified memory constraints that use $B * M_i$ in place of M_i and subject to a cost constraint,

$$\sum_{t=1}^{n}\sum_{i=1}^{t} C_i R_{t,i} \le 2 \sum_{v_i \in G_{\text{fwd}}} C_i + \sum_{v_i \in G_{\text{bwd}}} C_i. \tag{3.10}$$

The modified integer program has quadratic constraints, and is difficult to solve. We set a time limit of one day for the experiment, but Gurobi may be unable to reach optimality within that limit. Figure 3.6 then provides a lower bound on the maximum batch size that Checkmate can achieve.

For fair comparison on the non-linear graphs used in U-Net, FCN, and ResNet, we use the AP $\sqrt{n}$ and linearized greedy baseline generalizations described in Section 3.6.1. For the baselines, we iterate over batch sizes, find candidate solutions (multiple candidates for linearized greedy), and filter out the solutions that cost more than an additional forward pass or that would exceed the 16GB memory budget. The iteration stops when no solutions are available.

Costs are measured in FLOPs, determined statically. U-Net, FCN8 and SegNet semantic segmentation networks use a resolution of 416×608, and classification networks ResNet50, VGG19 and MobileNet use resolution 224×224.

Takeaways: We can theoretically increase the batch size of U-Net to 61 at a high resolution, an unprecedented result. For many tasks such as semantic segmentation, where U-Net is commonly used, it is not possible to use batch sizes greater than 16, depending on resolution. This is suboptimal for batch normalization layers, and being able to increase the batch size by 3.8× (61 vs 16 at this resolution) is quite significant. Orthogonal approaches to achieve this include model parallelism and distributed memory batch normalization which can be significantly more difficult to implement and have high communication costs.

Furthermore, for MobileNet, Checkmate allows a batch size of 1105 which is 1.73× higher than the best baseline solution, a greedy heuristic, and 5.1× common practice, checkpointing all activations. The same schedules can also be used to increase image resolution rather than batch size.

3.6.5 How well can we approximate the optimal rematerialization policy?

To understand how well our LP rounding strategy (Section 3.5) approximates the ILP, we measure the ratio $\textsc{Cost}_{\text{approx}}/\textsc{Cost}_{\text{opt}}$, *i.e.* the speedup of the optimal schedule, in FLOPs. As in Section 3.6.3, we solve each strategy at a range of memory budgets, then compute the geometric mean of the ratio across budgets. The aggregated ratio is used because some budgets are feasible via the ILP but not via the approximations. Table 3.2 shows results. The two-phase deterministic rounding approach has approximation factors close to optimal, at most 1.06× for all tested architectures.

3.7 Conclusion

One of the main challenges when training large neural networks is the limited capacity of high-bandwidth memory on accelerators such as GPUs and TPUs. This has created a memory wall that limits the size of the models that can be trained. The bottleneck for state-of-the-art model

	AP $\sqrt{n}$	AP greedy	Griewank $\log n$	Two-phase LP rounding
MobileNet	1.14×	1.07×	7.07×	**1.06×**
VGG16	1.28×	1.06×	1.44×	**1.01×**
VGG19	1.54×	1.39×	1.75×	**1.00×**
U-Net	1.27×	1.23×	-	**1.03×**
ResNet50	1.20×	1.25×	-	**1.05×**

Table 3.2: Approximation ratios for baseline heuristics and our LP rounding strategy. Results are given as the geometric mean speedup of the optimal ILP across feasible budgets.

development is now memory rather than data and compute availability, and we expect this trend to worsen in the future.

To address this challenge, we proposed a novel rematerialization algorithm which allows large models to be trained with limited available memory. Our method does not make the strong assumptions required in prior work, supporting general non-linear computation graphs such as residual networks and capturing the impact of non-uniform memory usage and computation cost throughout the graph with a hardware-aware, profile-guided cost model. We presented an ILP formulation for the problem, implemented the Checkmate system for optimal rematerialization in TensorFlow, and tested the proposed system on a range of neural network models. In evaluation, we find that optimal rematerialization has minimal computational overhead at a wide range of memory budgets and showed that Checkmate enables practitioners to train high-resolution models with significantly larger batch sizes. Finally, a novel two-phase rounding strategy closely approximates the optimal solver.

Acknowledgements

We would like to thank Barna Saha and Laurent El Ghaoui for guidance on approximation, Mong H. Ng for help in evaluation, and the paper and artifact reviewers for helpful suggestions. In addition to NSF CISE Expeditions Award CCF-1730628 and ONR PECASE N000141612723, this work was supported by gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware. This work was also supported by the NSF GRFP under Grant No. DGE-1752814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Chapter 4

Scalable data transfer in the cloud

4.1 Introduction

Increasingly, cloud applications transfer data across datacenter boundaries, both across multiple regions within a cloud provider (multi-region) and across multiple cloud providers (multi-cloud). This is in part due to privacy regulations, the availability of specialized hardware, and the desire to prevent vendor lock-in. In a recent survey [57], more than 86% of 727 respondents had adopted a multi-cloud strategy across diverse workloads. Thus, support for fast, cross-cloud bulk transfers is increasingly important.

Applications transfer data between datacenters for batch processing (e.g. ETL [16], Geo-Distributed Analytics [143]), and production serving (e.g. search indices [79]). Extensive prior work optimizes the throughput of bulk data transfers between datacenters within application-defined minimum performance constraints [79, 89, 95, 182]. All major clouds offer services for bulk transfers such as AWS DataSync [11], Azure AzCopy [44], and GCP Storage Transfer Service [66].

From the perspective of a cloud customer, transfer throughput and cost (price) are the two important metrics of transfers in the cloud. Thus we ask *how can we optimize network cost and throughput for cloud bulk transfers?* We study this question in the context of designing and implementing Skyplane, an open-source cloud object transfer system.

A seemingly natural approach is to optimize the routing protocols in cloud providers internal networks to support higher-throughput data transfers. Unfortunately, this is not feasible for two reasons. First, rearchitecting the IP layer routing protocol to optimize for high-throughput bulk transfer could be negatively impact other applications that are sensitive to network latency. Second, cloud providers lack a strong incentive to optimize data transfer to other clouds. Indeed, AWS DataSync [11], AzCopy [44], GCP Storage Transfer [66], AWS Snowball [155], and Azure Data

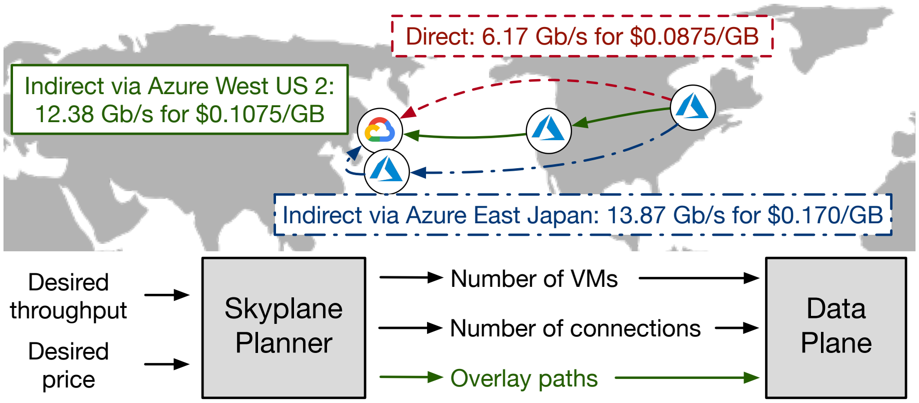

Direct: 6.17 Gb/s for $0.0875/GB
Indirect via Azure West US 2: 12.38 Gb/s for $0.1075/GB
Indirect via Azure East Japan: 13.87 Gb/s for $0.170/GB
Desired throughput
Desired price
Skyplane Planner
Number of VMs
Number of connections
Overlay paths
Data Plane

Figure 4.1: **Cloud-aware overlays:** Skyplane optimally transfers across cloud regions and providers subject to the user's cost and throughput requirements. Skyplane finds the visualized overlay path from Azure's `Central Canada` region to GCP's `asia-northeast1`, which is 2.0× faster but just 1.2× higher in price than the direct path.

Box Disk [19], all support data transfer *into*, but not out of, their respective clouds. Improvements to cross-cloud peering must be achieved with the cooperation of both the source and destination providers.

Skyplane's key observation is that we can instead identify *overlay paths*—paths that send data via intermediate regions—that are faster than the direct path. The throughput of the direct path from Azure's `Central Canada` region to GCP's `asia-northeast1` region is 6.2 Gbps. Instead, Skyplane can route the transfer via an intermediate stop at Azure's `US West 2` with a throughput of 12.4 Gbps for a 2.0× speedup (Fig. 4.1). Crucially, this can be implemented on top of the cloud providers' services without their explicit buy-in.

We are not the first to propose the use of overlay networks on the public Internet [15]. However, these techniques ignore two key considerations of public clouds: **price** and **elasticity**.

First, the highest-bandwidth overlay path may have an unacceptably high **price**. Cloud providers charge for data egress separately for each hop along the overlay path. To reduce the cost of the overlay, it is essential to transfer data along cheap paths to trade off price and performance. For example, in Fig. 4.1, one can achieve 13.9 Gbps by instead using Azure's `East Japan` region as the relay, but the cost would be 1.9× that of transferring data directly. In contrast, using Azure's `West US 2` region has only a 1.2× cost overhead with similar performance. Thus, Skyplane operates in a richer *problem space* than traditional application-level routing—one where cloud instance and cloud egress fees are significant.

Second, whereas the bandwidth between two nodes in a traditional network overlay [15] is considered "fixed," in Skyplane's setting it depends on **elasticity**—the ability to allocate more resources at each cloud region. For example, one can increase the capacity of any overlay path by simply allocating more VM instances in each cloud region. There are a limited number of physical machines at each cloud region, which cloud providers pass on to users in the form of instance limits. An overlay enables improved throughput beyond this limit. Thus, Skyplane operates in a richer *solution space* than traditional application-level routing—one where we must choose the number of VMs to use as relays due to cloud elasticity.

Skyplane addresses both **price** and **elasticity**, empowering users to navigate the trade-off between price and performance while leveraging cloud elasticity. Users can ask Skyplane to maximize bandwidth subject to a cost ceiling, or minimize cost subject to a bandwidth floor.

At the heart of Skyplane is a planner that computes a data transfer plan, subject to the user's constraints, that specifies the overlay path to use and amount of cloud resources to allocate along that path. Price and elasticity make it challenging to compute the plan. Our insight is that, with some care, planning can be formulated as *linear* constraints. Thus, Skyplane's planner can discover the optimal plan by solving a mixed-integer linear program (MILP) which can be closely approximated by a linear program (LP). Both can be accomplished using a fast, off-the-shelf solver.

Our Skyplane prototype[*]outperforms AWS DataSync by up to 4.6× and GCP Storage Transfer by up to 5.0×. Skyplane outperforms academic baselines like RON by 34% at 62% lower cost.

4.2 Background

Network overlays In the early 2000s, network overlays emerged as a technique for *application-level routing* without the *participation of underlying network providers.* These network overlays can be designed to improve performance or reliability. Notable network overlays include Chord [163], Resilient Overlay Networks (RON) [15], Bullet [104], Baidu BDS [183] and Akamai [131, 160].

Although ISPs may have broad visibility into their networks, the metrics that ISPs use to select routes may not align with user preferences. Wide-area networks today do not allow specification of alternative routing preferences while network overlays provide applications a mechanism to control routing decisions. For example, Akamai uses a network overlay to reduce the latency of CDN misses while RON routes around network outages via an unaffected intermediate host.

RON is implemented by periodically measuring network performance via probes embedded in a fixed set of routers. When path outages occur, RON selects an intermediate relay router to

[*] `https://github.com/skyplane-project/skyplane`

circumvent the outage. This intermediate router is selected to have low packet loss or latency to/from the client and server. Optionally, RON can use a model of TCP Reno's throughput [134] to select intermediate routers. RON will generally select only a single intermediate node.

Wide-area networking in the cloud From the perspective of cloud customers, the cloud is *elastic*—additional resources can be allocated on demand. For example, an overloaded cloud application can leverage the cloud's elasticity by allocating additional VM instances. However, the physical reality of the cloud is that there are only finite resources at each region. Therefore, cloud providers impose *service limits* on their customers for resources such as VMs.

Each VM's network bandwidth is throttled according to its instance type. For example, an AWS `m5.8xlarge` instance can use at most 10 Gbps of network bandwidth, and an Azure `Standard_D32_v5` instance can use at most 16 Gbps of network bandwidth. Furthermore, only some of the available bandwidth can be used for egress traffic to another cloud provider. The policies differ by cloud provider. AWS limits VM egress bandwidth to the larger of 5 Gbps or 50% of total bandwidth [10], GCP limits VM egress bandwidth to any public IP address to 7 Gbps [65], and Microsoft Azure has no egress limit beyond the total bandwidth limit for a VM. Of course, the actual achievable TCP network bandwidth is subject to congestion control which may be less than the limit.

Cloud egress pricing Cloud providers charge egress prices for network traffic leaving a cloud region. Importantly, egress prices are assessed based on the *volume* of data transferred, not the rate at which it is transferred. Transferring a file at 10 Mbps or at 10 Gbps will result in the same egress charge. Egress charges introduce asymmetry in billing—there is no corresponding ingress charge for transfers into a cloud.

For intra-cloud transfers (i.e., transfers between two regions or zones in the same cloud), transfers between geographically distant endpoints are priced more than transfers between nearby endpoints. In contrast, inter-cloud transfers (i.e., transfers between two cloud providers) are billed at the same rate regardless of the transfer's geographic distance. For example, the egress price from a single Azure region is billed at the same rate for *any* destination outside of Azure, including any region in AWS or GCP [12, 64, 125].

Egress prices typically dominate the cost of a bulk transfers. For example, if a VM sends data at a rate of 1 Gbps for an hour on AWS with an Internet egress price of $0.09/GB, the total egress charge will total $40.50, which far exceeds the VM price of $1.50 (for `m5.8xlarge`) [12].

Cloud object storage AWS, Azure, and GCP provide object storage APIs that allow customers to save data attached to a string key. Data is stored immutably and therefore any updates require writing a new version. Unlike POSIX file systems, object stores do not provide atomic metadata operations (e.g., rename). Consistency models vary across providers. Cloud object stores store copies of a blob on multiple machines to improve availability and durability. Large objects support concurrent writes via sharding. Read throughput of a single shard may be limited by the provider (e.g. 60 MB/s for Azure [20]).

4.3 Overview of Skyplane

Skyplane allows applications to efficiently transfer large objects from an object store in one region to an object store in another cloud region or provider. To use Skyplane, the user installs the Skyplane client locally and configures it with access to cloud provider-supplied credentials. Then, the user submits a job, together with a constraint on price or bandwidth. The job specifies which objects to transfer, the source cloud provider and region, and the destination cloud provider and region. The constraint can have one of two forms: it can ask Skyplane to optimize either *bandwidth subject to a price ceiling*, or *price subject to a bandwidth floor*.

Skyplane itself comprises a *planner* (Fig. 4.1, bottom) and a *data plane* (Fig. 4.2). Given the user's job and constraint, the planner produces an optimal data transfer plan to complete the job subject to the constraint. The planner relies on a profile of the network throughput between different cloud regions. The data plane is responsible for executing the data transfer plan: allocating cloud resources (e.g., VMs), transferring data between them, and interacting with object stores.

4.3.1 Overlay formulation in Skyplane's planner

Suppose the user needs to transfer an object from a source cloud region, A, to a destination cloud region, B. A naïve object transfer system might spawn VMs in regions A and B, and transfer data via a TCP connection between the two VMs. Skyplane improves performance compared to this baseline by applying principles from overlay networks [15]. For example, Skyplane may identify a third cloud region, C, and transfer data from A to B via C. This is accomplished at the application layer; Skyplane will spawn a VM in region C, set up TCP connections from A to C and from C to B. We refer to intermediate regions like C as *relay regions*.

The baseline approach ($A \rightarrow B$) routes data along the "direct path," since it uses the default path provided by the Internet. However, Skyplane ($A \rightarrow C \rightarrow B$) routes data along the an "indirect path," that may not be on the Internet-provided default path. An indirect path may use multiple relays although a single relay is usually sufficient.

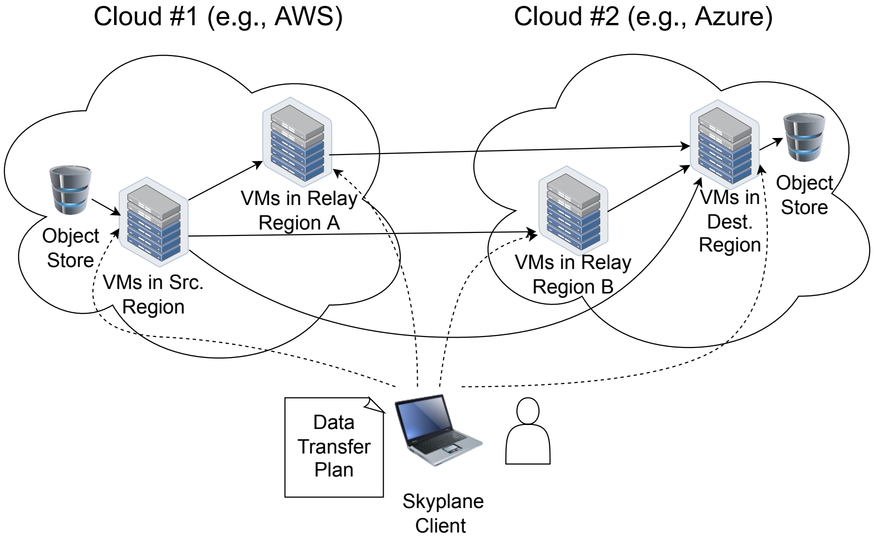


Figure 4.2: Skyplane splits an example data transfer over three paths: the direct path, and two indirect paths. Dashed lines indicate control orchestration (e.g., for spawning VMs) and solid lines depict the flow of object data.

A key difference between Skyplane and classical overlay networks is that Skyplane takes price into account when choosing the overlay path to use for a job. Concretely, Skyplane's planner uses a *price grid* and a *throughput grid* to determine which indirect path to use. The price grid specifies the price of transferring data between each pair of cloud regions, in each direction. We computed the price grid based on information on the cloud providers' websites and from querying the cloud APIs. The throughput grid is collected by measuring the network, as we explain in the next subsection.

Note that throughput grid measurements are made using TCP connections, subject to TCP congestion control. Thus, the throughput grid measures the bandwidth available to *a single user* for transferring data, accounting for cross-traffic from other users' flows. We assume a high degree of statistical multiplexing in wide-area network traffic—in other words, that the bandwidth consumed by a single user's bulk transfer is negligible compared to the total available inter-region bandwidth. This allows a Skyplane user to compute a data transfer plan without regard to other users' bulk transfers using Skyplane or other bulk transfer tools—all cross traffic from other users is assumed to be accounted for in the throughput grid. As we show in the next subsection, the bandwidth of inter-region TCP connections is relatively stable in the short term, validating our assumption of high statistical multiplexing.

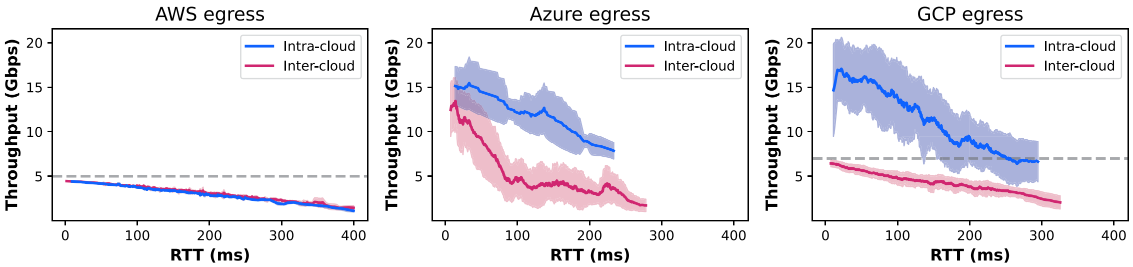


Figure 4.3: **Intra-cloud vs. inter-cloud links**: Inter-cloud links are consistently slower than intra-cloud links for network routes from Azure and GCP. Service limits are shown with a dashed line; GCP throttles inter-cloud egress to 7 Gbps while AWS throttles *all egress traffic* to 5 Gbps.

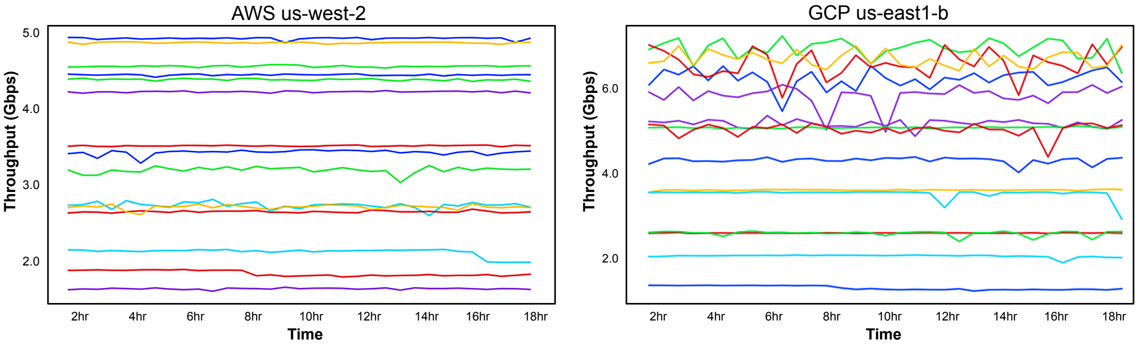


Figure 4.4: **Stability of egress flows over 18 hour period**: Continuous probes of cloud networks over one day reveal that routes from AWS have stable throughput over time. Paths between GCP regions are noisy but have a consistent mean.

4.3.2 Profiling cloud networks

The planner relies on a profile of the network throughput between pairs of cloud regions. We collected a throughput grid by measuring the TCP goodput between each region pair using `iperf3`. In total, computing this profile cost approximately $4000 in egress charges.

Fig. 4.3 displays the relationship between network latency and throughput for profiling routes originating from GCP and Azure for our measured throughput grid. For GCP, we leverage internal IPs which improve intra-cloud bandwidth. For both GCP and Azure, intra-cloud routes had lower tail RTTs than inter-cloud routes. We observe that in both GCP and Azure, inter-cloud links are slower than intra-cloud links. As Azure has no service limit for egress bandwidth, we see the fastest intra-cloud links achieve up to the NIC capacity of 16 Gbps. However, both GCP and AWS encounter egress throttling at 7 Gbps and 5 Gbps respectively.

A natural question is how frequently the throughput grid must be re-measured. Fig. 4.4 visualizes achieved throughput from AWS `us-west-2` and GCP `us-east1-b` taken every 30 minutes over an 18 hour timespan. Throughput is stable over time for both inter-cloud and intra-cloud routes from AWS `us-west-2`. Routes from GCP `us-east1-b` to AWS destinations is similarly stable but intra-cloud routes to GCP destinations are less stable. Regardless, the overall rank order of regions by throughput remains mostly consistent over a few hours. Thus, it should be sufficient to profile networks relatively infrequently (i.e. every few days). In practice, this information could be collected by third-party service, or measured via active probing along live transfers.

4.3.3 Skyplane's data plane

Skyplane's data plane executes data transfers using the plan computed by Skyplane's planner. Ephemeral VMs for a single transfer, called "gateways," are provisioned in the source region, destination region, and overlay regions for a transfer plan. Each source gateway reads a small shard of data from the object store and transfers data via intermediate gateways to the destination where the shard is written.

Skyplane reads data from an object store in the source cloud region and writes data to an object store in the destination cloud region. We focus on the object stores provided as a service by AWS S3, Azure Blob Storage, and Google Storage. Unlike a traditional overlay network, there is no central Skyplane service that allocates resources to each user from a pool of "Skyplane resources." Instead, Skyplane can be understood as a local service run by each user that is invoked when an application needs to transfer data. Skyplane directly allocates cloud resources on the user's behalf when processing a job, and manages those resources to transfer the user's data across cloud regions. This allows Skyplane to manage each user's resources according to their cost and performance objectives, independently from the cloud providers' existing data transfer services, while relying on clouds to offer a large pool of resources and manage isolation between users.

4.4 Principles of Skyplane's planner

Skyplane's planner[†] is responsible for developing a plan for transferring data across the wide area to complete an object transfer job submitted by a user or their application (Fig. 4.5). This plan describes the overlay path and the amount of cloud resources to allocate along that path to facilitate the transfer.

Skyplane's planner supports two modes:

[†] Explore Skyplane's planner at `https://optimizer.skyplane.org`

Cost minimizing: The planner will minimize cost subject to an application-specified throughput constraint.

Throughput maximizing: The planner will maximize throughput subject to an application-specified cost constraint.

As we will describe in §4.5, Skyplane finds the optimal plan by formulating it as an Mixed-Integer Linear Program (MILP) and using a fast but exponential-time solver. This section describes the degrees of freedom available to the optimizer to navigate the price-performance trade-off for the user's specified constraint. Our goal is to describe what aspects of the plan are at the planner's disposal, justify why it is reasonable to vary those aspects of the plan, and describe certain techniques available to the planner to manage the price-performance trade-off. Note that the planner is not directly programmed to use these techniques; they are merely patterns that it discovers in the course of finding the optimal MILP solution.

4.4.1 Achieving low instance and egress costs

That bandwidth costs dominate the cost of data transfer (§4.2) is both a challenge and an opportunity for Skyplane. It is an opportunity because it allows Skyplane to be competitive with the price of using data transfer tools provided directly by the cloud providers (e.g. AWS DataSync, AzCopy, GCP Cloud Transfer Service), as those tools incur bandwidth costs but not instance costs. It is a challenge for Skyplane because it implies that, used naïvely, indirect paths are much more expensive than direct paths. This is because egress bandwidth is charged for each hop along the path. For example, for a path $A \rightarrow C \rightarrow B$, the bandwidth cost must be paid for both $A \rightarrow C$ and $C \rightarrow B$, which could be *double* the cost of transferring over the direct path. As a result, it is crucial for Skyplane's optimizer carefully manage egress transfer costs.

Choosing the relay region

One way for Skyplane to manage the additional cost associated with indirect paths is to carefully choose the relay region C to minimize this cost. For example, suppose that a user needs to transfer an object from AWS `us-west-2` (region A) to Azure `UK South` (region B). The direct path $A \rightarrow B$ would require the user to pay $0.09 per GB, the cost of bandwidth leaving AWS' network. If the relay region C is chosen in `us-central-1` or `us-east-1`, then the overall bandwidth price will only increase slightly; while the $C \rightarrow B$ transfer still incurs $0.09 per GB, as data is leaving AWS' network, the $A \rightarrow C$ bandwidth only costs $0.02 per GB, as it is an intra-continental transfer within the cloud provider's network. Skyplane's planner can use the throughput and price grids

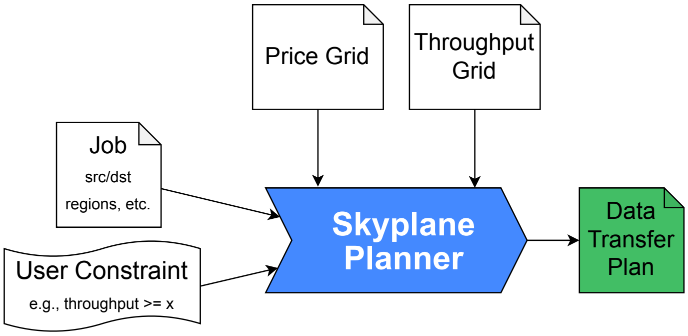


Figure 4.5: Skyplane's planner considers throughput and cost constraints from the user along with per-cloud price information and an inter-region throughput profile grid to determine the optimal data transfer plan.

to identify relay regions that improve the performance of the transfer while minimizing additional bandwidth costs.

Combining multiple paths

Another way to manage the cost of indirect paths is to split the data transfer over multiple paths, in order to make fine-grained trade-offs between price and performance. For example, suppose that Skyplane identifies a high-bandwidth indirect path, but that the path is more expensive than the user's price ceiling. Skyplane can still benefit partially from that indirect path by sending part of the data over that path, at higher cost, and the remaining data over the direct path $A \rightarrow B$, at lower cost. Thus, Skyplane may average the price and performance of multiple paths, when doing so allows Skyplane to more optimally satisfy the user's constraints.

4.4.2 Parallel TCP for high bandwidth

Skyplane uses parallel TCP connections—that is, bundles of TCP connections—to achieve high goodput over a chosen path. This is a well-known technique for achieving good performance, particularly for wide-area transfers [7, 161]. Our Skyplane implementation uses up to 64 outgoing connections for each VM instance, as we empirically measured that using additional connections typically resulted in diminishing benefits in aggregate goodput. When collecting measurements for the throughput grid, we make sure to use 64 parallel connections to measure the achievable TCP goodput for each ordered pair of regions.

It is known that using multiple TCP streams in parallel may cause an application to obtain more than its "fair share" of bandwidth [55, §A.1], particularly in contexts where networks are running at nearly 100% utilization [89]. Our view is that, despite this, it is acceptable to use multiple TCP connections in parallel in the context of Skyplane. There are three reasons for this. First, it is common for applications to use parallel TCP, including for workloads like bulk data transfer [7, 109]. It is important for Skyplane to appropriately compete with such applications for limited bandwidth. Second, the user pays the cloud provider for bandwidth, both in the form of the bandwidth price (total amount transferred) and the instance price (rate at which data can be transferred), and it is natural for users to be able to make use of the bandwidth that they pay for. Third, cloud providers control the datacenter network, and can shape traffic in the presence of congestion to ensure that each customer gets a fair share of bandwidth.

4.4.3 Multiple VMs for high bandwidth

For a given overlay path, Skyplane must allocate sufficient resources along the path to achieve high bandwidth. However, the achievable outgoing bandwidth from a VM instance is limited, as described in §4.2.

Therefore, Skyplane may allocate multiple VM instances at certain regions along the path, to increase *aggregate* data transfer rate of the VMs at each region. Although simply using larger VMs may seem like a viable alternative, it is less effective than using multiple instances due to per-instance bandwidth limits. Skyplane uses a fixed VM size, and its planner chooses how many instances to allocate in each region, under the assumption that TCP goodput scales linearly with the number of allocated VM sizes.

It may seem that Skyplane can achieve an arbitrarily high bandwidth by spawning many instances in each region. Unfortunately, this simple strategy does not work because cloud resources are not perfectly elastic. The finite capacity for VMs in a datacenter is passed down to cloud customers in the form of service limits, which limit the number of VM instances, and therefore the amount of network bandwidth, that users can allocate in each region. While users can request limit increases, these are ultimately subject to resource availability. To model this, Skyplane's planner takes into account a limit on the number of instances that a user can allocate per region.

4.5 Finding optimal transfer plans

Skyplane's planner searches for cost-efficient high-throughput transfer plans that jointly specify the overlay path, TCP connections between regions and VMs to provision per region.

At the core of Skyplane's planner is an optimizer that finds the optimal plan using off-the-shelf Linear Programming (LP) solvers. We formalize the constraints of our problem as Mixed Integer LP

Variables	
$\mathrm{F} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$	*Throughput grid*
$\mathrm{N} \in \mathbb{Z}_+^{\lvert V\rvert}$	*VMs per region*
$\mathrm{M} \in \mathbb{Z}_+^{\lvert V\rvert \times \lvert V\rvert}$	*TCP conn. per region*
Constraint: goal throughput	
$\textsc{tput goal} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$	*User's desired throughput*
Constants: provider limit	
$\textsc{Limit}^{link} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$	*Throughput grid limit*
$\textsc{Limit}^{conn} \in \mathbb{Z}_+^{\lvert V\rvert \times \lvert V\rvert}$	*TCP connection limit*
$\textsc{Limit}^{ingress} \in \mathbb{Z}_+^{\lvert V\rvert}$	*VM limit*
$\textsc{Limit}^{egress} \in \mathbb{Z}_+^{\lvert V\rvert}$	*Egress bandwidth limit*
Constants: provider cost	
$\textsc{Cost}^{egress} \in \mathbb{R}_+^{\lvert V\rvert}$	*Egress cost (\$/Gbit)*
$\textsc{Cost}^{VM} \in \mathbb{R}_+^{\lvert V\rvert}$	*VM cost (\$/s)*

Table 4.1: Symbol table for Skyplane's ILP formulation.

(MILP) which can quickly be solved in under 5 seconds with an open-source solver. The problem can be further relaxed into a continuous LP which is solvable in worst-case polynomial time via interior point methods [96].

Independently optimizing for each variable then combining partial solutions would not guarantee a globally optimal solution. It is therefore important that Skyplane's planner models all variables in an integrated search space to obtain provably optimal data transfer plans.

4.5.1 Cost minimizing overlay paths

Flow networks can naturally represent overlay networking topologies like those used by Akamai [160]. We start with a min-cost flow problem. The following primal LP finds the optimal flow matrix $\mathrm{F} \in \mathbb{R}_+^{\lvert V\rvert \times \lvert V\rvert}$ for a network topology graph $G = (V, E)$ where nodes represent regions and edges are links:

$$\begin{array}{ll} \underset{\mathrm{F}}{\arg\min} & \langle C, \mathrm{F} \rangle \\ \text{subject to} & \sum_{(c,v)\in E} \mathrm{F}_{c,v} \geq \textsc{tput goal}, \\ & \sum_{(u,v)\in E} \mathrm{F}_{u,v} = \sum_{v,w} \mathrm{F}_{v,w} \quad \forall v \in V - \{s,t\}, \\ & \quad 0 \leq \mathrm{F} \leq \textsc{Limit}^{link} \end{array} \tag{4.1}$$

where s and t are the source and destination regions, $\textsc{Limit}^{link} \in \mathbb{R}_+^{|V|\times|V|}$ is the maximum capacity for each link and $C \in \mathbb{R}_+^{|V|\times|V|}$ is the cost per unit of bandwidth between regions. We use the same notation for matrix and vector inner products: $\langle C, \mathrm{F}\rangle = \sum_{u,v} C_{u,v}\mathrm{F}_{u,v}$.

Objective: Minimize cost from egress and VMs

Min-cost flows do not accurately reflect the cost of transfers in the cloud. The total cost of a transfer in Skyplane includes *egress cost* and *VM cost*. Note that this objective is not linear; we present a linear reformulation in Sec. 4.5.1. We present the full objective is in the in Equation 4.4a.

Modeling egress cost Unlike physical networks, virtual networks in the cloud will charge the same amount if 1GB of data is sent at 1 Mbps or 10 Gbps. Transfers are priced according to *egress volume* ($ per GB, $\textsc{Cost}^{egress}$) rather than *bandwidth* ($ per Gbps). We can update the cost function to instead model the transfer cost by first computing how much the overlay path costs to run per unit time and then scale that by the runtime for a transfer. We denote the total volume of the transfer as VOLUME. Total egress cost is then:

$$\underbrace{\langle \mathrm{F}, \textsc{Cost}^{egress}\rangle}_{\text{Egress cost per s}} * \underbrace{\textsc{volume} \div \textstyle\sum_{v\in V} \mathrm{F}_{s,v}}_{\text{Transfer time}} \tag{4.2}$$

Modeling VM cost Multiple VMs can increase aggregate bandwidth as discussed in Sec. 4.4.3. To optimally trade-off parallel VMs with the overlay, we introduce a new decision variable $\mathrm{N} \in \mathbb{Z}_+^{|V|}$ that models the number of instances use to transfer data per region. VM count per region may vary due to asymmetric egress and ingress limits. To accurately consider transfer costs from VMs, we add the the following instance cost expression to Equation 4.2 where $\textsc{Cost}^{VM}$ is a vector containing the cost per second per VM in each region:

$$\underbrace{\langle \mathrm{N}, \textsc{Cost}^{VM}\rangle}_{\text{VM cost per s}} * \underbrace{\textsc{volume} \div \textstyle\sum_{v\in V} \mathrm{F}_{s,v}}_{\text{Transfer time}} \tag{4.3}$$

Linear reformulation of the objective As written, the objective in Equation 4.4a is not linear due to a product of variables between F and N. By reformulating the problem to instead consider finding a plan that provides *exactly* TPUT GOAL (instead at least), the runtime for the transfer can be reduced to a constant VOLUME ÷ TPUT GOAL.

Constraints: Cloud provider service limits

Resources are not infinite at cloud regions; providers limit the number of VMs that a user may provision and in some cases, providers may throttle the performance of ingress and egress.

Per VM ingress and egress limits AWS and GCP each throttle egress from their clouds via SDN policies. For AWS, instances with 32 cores or less are limited to 5 Gbps. For GCP, individual flows are limited to 3 Gbps and total egress is service limited to 7 Gbps. Ingress is bottlenecked by VM NIC bandwidth. We constrain the maximum ingress bandwidth per VM to $\textsc{Limit}^{ingress}$ via Constraint 4.4f and the maximum egress bandwidth per VM to $\textsc{Limit}^{egress}$ via Constraint 4.4g.

Constraining TCP connections Using parallel TCP connections is a well known approach to improve WAN performance as discussed in Section 4.4.2. Yet, bandwidth does not scale linearly with connections (Figure 4.9a). We introduce a decision variable $\mathrm{M} \in \mathbb{Z}_+^{|V|\times|V|}$ representing the number of connections between a *pair of regions* (not per VM pair). Constraint 4.4b ensures M is constrained by N and $\textsc{Limit}^{conn}$ (typically 64 per VM). We then limit the total incoming and outgoing connections with Constraints 4.4i and 4.4h.

Per-region VM limits We introduce the variable $\mathrm{N} \in \mathbb{Z}_+^{|V|}$ to denote the number of VMs per region. N must be under the global instance cap in Constraint 4.4j. The optimizer linearly scales the maximum number of egress TCP connections per region by the number of VMs provisioned in each region.

Continuous relaxation of MILP

To improve solve times, N and M are relaxed into real valued variables $\mathrm{N} \in \mathbb{R}_+^{|V|}$ and $\mathrm{M} \in \mathbb{R}_+^{|V|\times|V|}$. Rounding variables down performs comparably to randomized rounding with solutions $\leq 1\%$ from optimal. The relaxed problem has worst case polynomial time complexity [96].

Full formulation of the cost optimal solver

All variables and constants are listed in Table 4.1. The full formulation of Skyplane's optimizer is:

$$\underset{\substack{F, N\\M}}{\arg\min} \quad \frac{\text{VOLUME}}{\text{TPUT GOAL}}\left(\langle F, \textsc{Cost}^{egress}\rangle + \langle N, \textsc{Cost}^{VM}\rangle\right) \tag{4.4a}$$

subject to

$$F \leq (\textsc{Limit}^{link} \odot M) \div \textsc{Limit}^{conn}, \tag{4.4b}$$

$$\textstyle\sum_{v \in V} F_{s,v} \geq \text{TPUT GOAL}, \tag{4.4c}$$

$$\textstyle\sum_{u \in V} F_{u,t} \geq \text{TPUT GOAL}, \tag{4.4d}$$

$$\textstyle\sum_{u \in V} F_{u,v} = \sum_{u \in V} F_{v,u} \quad \forall v \in V - \{s, t\}, \tag{4.4e}$$

$$\textstyle\sum_{u \in V} F_{u,v} \leq \textsc{Limit}_v^{ingress} * N_v \quad \forall v \in V \quad , \tag{4.4f}$$

$$\textstyle\sum_{v \in V} F_{u,v} \leq \textsc{Limit}_u^{egress} * N_u \quad \forall u \in V \quad , \tag{4.4g}$$

$$\textstyle\sum_{v \in V} M_{u,v} \leq \textsc{Limit}^{conn} * N_v \quad \forall u \in V \quad , \tag{4.4h}$$

$$\textstyle\sum_{u \in V} M_{u,v} \leq \textsc{Limit}^{conn} * N_u \quad \forall v \in V \quad , \tag{4.4i}$$

$$N_v \leq \textsc{Limit}^{VM} \quad \forall v \in V \tag{4.4j}$$

4.5.2 Throughput maximizing overlay paths

Directly solving for a throughput maximizing path under a cost ceiling is non-trivial as we cannot use the linear reformulation of the cost objective. We can approximate a solution by solving for the minimum cost transfer plan at a range of many throughput goals. The result of this procedure is a Pareto frontier curve (as shown in Fig. 4.9c). A throughput maximizing solution can be extracted from this curve. The quality of approximate solution will depend on how many samples are used. A single AWS `c5.9xlarge` instance can evaluate 100 samples in under 20 seconds.

4.6 Implementation of Skyplane

We implemented Skyplane in Python 3. Skyplane's planner uses the proprietary Gurobi library to solve MILP instances (used in our evaluation), but the Coin-OR library can be used instead to avoid this dependency. Our implementation currently supports the three major cloud providers: Amazon Web Services, Microsoft Azure, and Google Cloud Platform.

We use `m5.8xlarge` instances on AWS, as smaller VM sizes were subject to burstable networking performance, which we wished to avoid [10, 13]. For consistency, we used `Standard_D32_v5` instances on Microsoft Azure and `n2-standard-32` instances on Google Cloud.

A user initiates a transfer from their application with the *Skyplane client*. The client provisions VMs in each region according to the transfer plan and runs the *Skyplane gateway* program on each VM. The gateway is responsible for actually reading from source object stores, relaying data through overlay regions and writing to destination object stores.

While transfer time is dominated by network throughput, the time to spawn gateway VMs contributes to the transfer latency. To minimize unnecessary bloat in VM images, we use compact OSes such as Bottlerocket [154] and package dependencies via Docker.

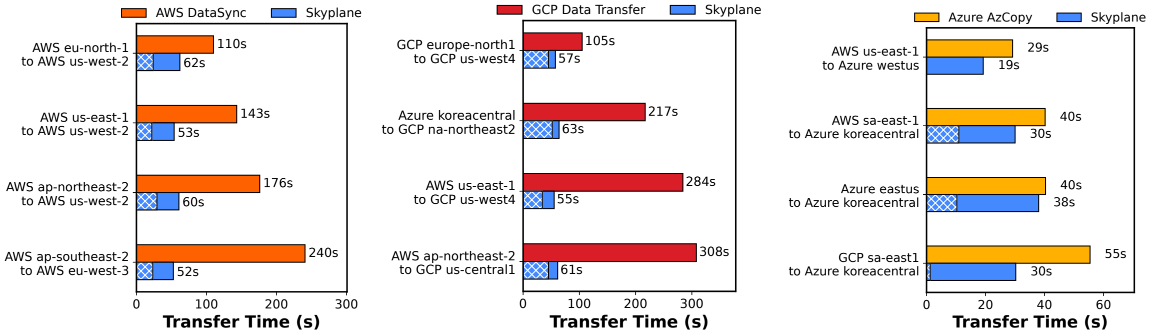
Skyplane assumes that objects are broken up into small *chunks* of approximately equal size. Applications can often do this without significant burden; for example, machine learning applications store data as `TFRecords`, which are easy to split into small chunks. This allows Skyplane to read and write data quickly from and to cloud object stores, by issuing many read/write operations in parallel to different chunks.

To mitigate the impact of straggler connections, Skyplane dynamically partitions data across TCP connections as they become ready to accept more data. This is in contrast to tools like GridFTP [7], which assign data blocks to connections in a round-robin fashion. The downside is that, for plans that use multiple overlay paths, the amount of data sent on each path may deviate from the targets computed at planning time, which could cause the actual cost of transferring data to deviate from the cost predicted by Skyplane's planner.

To avoid overflowing buffers at relay regions, Skyplane uses hop-by-hop flow control to stop reading data from incoming TCP connections when a VM's queue of chunks reaches capacity. Bufferbloat-type problems [59] are not a concern for Skyplane, with regard to queued chunks, as we pipeline transfers to optimize for throughput instead of latency.

4.7 Evaluation

To evaluate Skyplane, we investigate transfer time and price. We will sometimes use transfer throughput as a proxy for transfer time. In our price calculations, we include both instance cost and egress cost.



(a) AWS DataSync comparison (b) GCP Storage Transfer comparison (c) Azure AzCopy comparison

Figure 4.6: **Comparison to cloud transfer systems**: The thatch pattern in each bar represents the storage I/O overhead.

4.7.1 Experimental setup

We evaluate Skyplane with 20 AWS regions, 24 Azure regions and 27 GCP regions. For all experiments, we use public IP addresses attached to the VMs for transferring data. In some cases, one can achieve better performance for intra-cloud overlay hops by using private IP addresses assigned to each VM. For GCP this yields higher performance; for AWS and Azure it may yield higher performance, but requires peering virtual networks which incurs additional fees.

Furthermore, Azure and GCP allow one to select *network tiers* to control whether data is transferred via the cloud provider's network or via the public Internet. The Skyplane prototype utilizes external IPs over standard network tiers. That said, Skyplane is not incompatible with optimizations like VPC peering or hot-potato routing tiers to reduce cost and improve performance which we leave to future work. We use the CUBIC congestion control protocol in experiments.

4.7.2 How much faster is Skyplane than existing data transfer solutions?

Existing cloud providers offer data transfer tools such as AWS DataSync, GCP Storage Transfer, and Azure AzCopy for low-cost transfers of bulk data into their respective clouds. These tools do not disclose what mechanisms they use to transfer data—for example, the number of VMs and TCP connections (if any) used for a transfer, or the QoS (if any) associated with the network traffic. When evaluating Skyplane, we restrict Skyplane to use at most 8 VMs in each region. This is conservative; for example, on equalizing $/GB for some routes, Skyplane could provision *up to 262 VMs* per region within DataSync's service fee. Moreover, while these services only support data transfer *into* their respective clouds, Skyplane supports data transfer between every region pair.

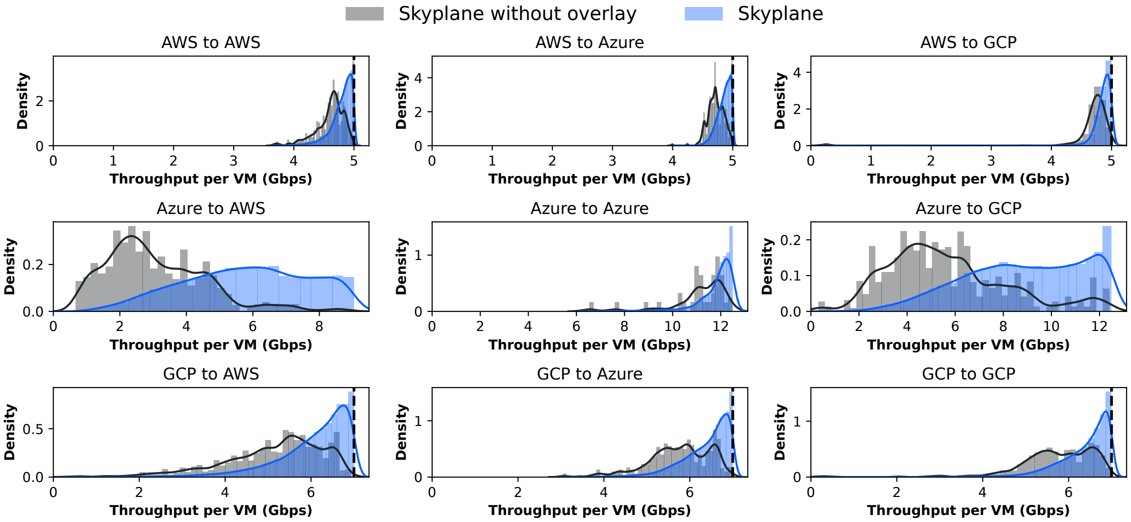


Figure 4.7: **Ablation of predicted overlays**: Overlay routes improve throughput per VM instance. We visualize the distribution of predicted throughput by the planner with all optimizations enabled (Skyplane) and with all optimizations except for overlay routing (Skyplane without overlay). The AWS and GCP egress limits are displayed with a dashed line.

We consider transferring the training and validation set for ImageNet [45]. We specifically use the `TFRecords` as generated by Google as part of the Cloud TPU benchmark example [45]. We evaluate flows between regions within a single cloud (intra-provider) and between clouds (inter-provider). We expected that data transfer within each cloud provider (e.g., between AWS's `us-east-1` and AWS's `us-west-1`) to perform well as they have full visibility into their networks and can utilize private interfaces with higher performance than over public API. For example, Azure Blob Storage throttles per-object reads for third-party VMs[126]. Our experiments did observe this behavior. However, Skyplane benefits from parallelizing the transfers.

We compare against AWS DataSync, GCP Storage Transfer and Azure AzCopy in Fig. 4.6. We evaluated Skyplane with a cost budget cap that is lower than the service fee for cloud transfer services in all our experiments. For each source-destination pair, we additionally measured the time to transfer procedurally-generated data using Skyplane; this allows us to break out the overhead of reading and writing to cloud storage as a "thatched" region in each bar. Skyplane significantly outperforms AWS DataSync and GCP Cloud Transfer in all configurations. In certain cases, Azure AzCopy performs about as well as Skyplane. We chose the `koreacentral` region because we expected the greatest improvements from the overlay in that region; however, storage

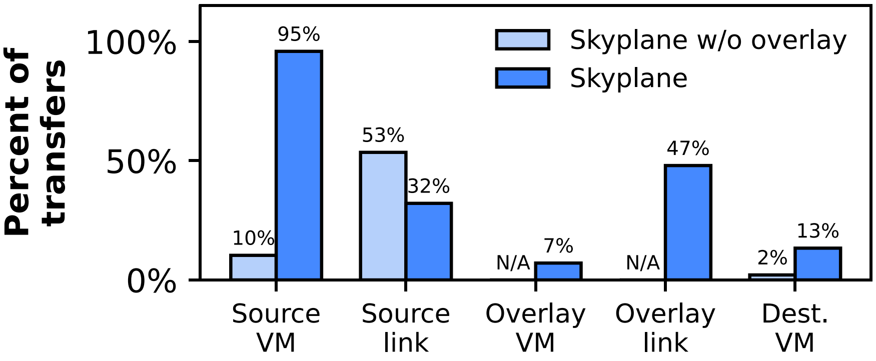


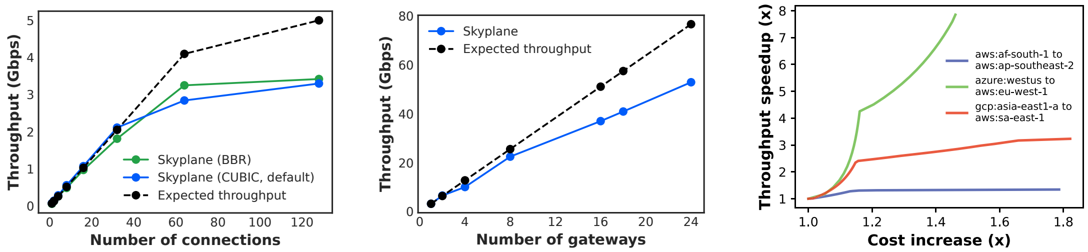
Figure 4.8: **Transfers bottlenecked at each location**: For transfers in Fig. 4.7, we visualize what percentage of transfers were bottlenecked at various locations. Enabling the overlay shifts bottlenecks from the network to the VM.

overheads (the "thatched" regions of the bars), not networking overheads, dominated the runtime. It is possible that AzCopy avoids the Azure Blob Storage I/O overhead that dominates Skyplane's transfer time by leveraging Azure's `Copy Blob From URL` API call to download data directly into the servers running Azure Blob Storage [18].

4.7.3 How much faster are the overlay paths?

The planner optimally explores the trade-off between improved throughput and cost for cloud data transfers. We explore solving for the optimal transfer path between all pairs of clouds regions between all cloud providers. We evaluated 22 AWS regions, 23 unrestricted Azure regions and 27 GCP regions which leads to 5,184 possible replication routes. It would be too expensive to transfer a large amount of data along each path in order to measure the empirical achieved throughput; therefore we use the planner to generate a plan and compare the resulting plan with the direct path, both in terms of expected throughput and cost. We compute predicted costs for transferring a 50 GB dataset between each possible source and destination. We report the speedup relative to Skyplane with a direct connection between each set of instances. The baseline is itself an ablation of Skyplane and it generally outperforms existing cloud transfer services to begin with (see §4.7.2).

The results are shown in Fig. 4.7. For each pair of source and destination clouds, we show distribution of predicted throughputs across region pairs, both with Skyplane's planner restricted to the direct path and allowing Skyplane's planner to use overlay paths. The results show that Skyplane's overlay routing meaningfully improves achievable throughput between cloud regions.



(a) TCP connections versus throughput

(b) Number of gateway VMs versus throughput

(c) Predicted planner throughput versus cost

Figure 4.9: **Skyplane ablations**: We evaluate the impact of parallel TCP connections, parallel gateway VMs and overlay cost.

Note that transfers out of AWS cannot exceed 5 Gbps and transfers leaving GCP cannot exceed 7 Gbps due to these cloud providers' caps on egress bandwidth.

4.7.4 Where are transfer bottlenecks?

To understand how the overlay improves throughput, we characterize the fraction of transfers that are bottlenecked at each location. In Fig. 4.8, we visualize the percentage of transfers from §4.7.3 that were bottlecked at a VM in the source region, the network link leaving the source region, a VMs in optional overlay regions, a network links leaving an overlay region, and a VM in the destination region. We consider a particular location to be a bottleneck if utilization is over 99%. Multiple locations may simultaneously be a bottleneck for one transfer.

For Skyplane with overlay routing disabled, the network link from the source to the destination region is the most common bottleneck for transfers. In a small set of cases, the source VM is a bottleneck for the transfer. Generally, the direct path is not fast enough to saturate the maximum egress bandwidth limit for a VM. The overlay shifts source link bottlenecks by reducing the number of transfers bottlenecked by the source link by 32%. The bottleneck shifts to the source VM or in some cases a network link leaving an overlay region.

4.7.5 Skyplane microbenchmarks

Impact of parallel TCP connections Fig. 4.9a shows the impact of varying the number of parallel TCP connections used to transfer data between VMs. For this experiment, the source VM was located in AWS `ap-northeast-1` and the destination VM was located in AWS `eu-central-1`. Skyplane transfers 32 GB of synthetic, procedurally-generated data in these experiments to avoid

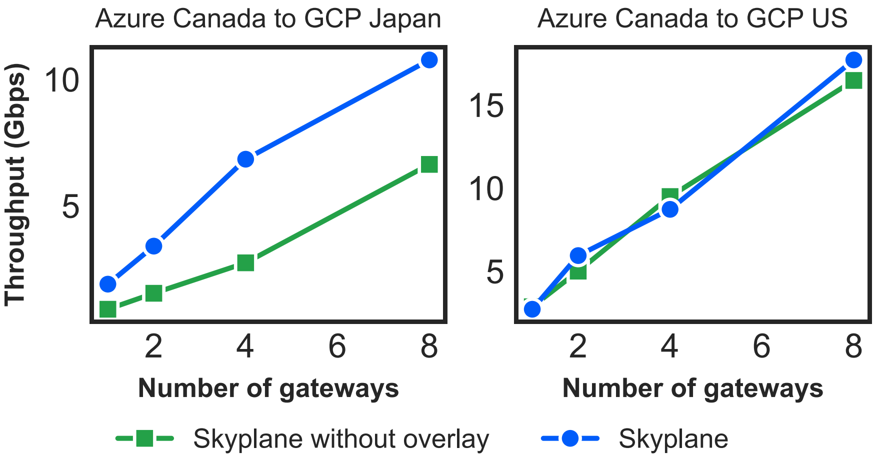


Figure 4.10: **Scaling VMs versus overlay**: In situations where the direct path is slow, the overlay is faster than simply scaling the number of VMs used alone.

incurring object store I/O overheads and thereby isolate network performance. The black dashed line shows the expected throughput, assuming that bandwidth scales linearly with the number of parallel TCP connections up to AWS' 5 Gbps egress cap. The blue line shows Skyplane's achieved throughput, and the green line uses Skyplane's achieved throughput using the BBR congestion control algorithm (used only this experiment). For this experiment, the source VM was located in AWS `ap-northeast-1` and the destination VM was located in AWS `eu-central-1`. Skyplane's achieved throughput plateaus below the 5 Gbps egress cap at 64 connections.

Impact of parallel VMs Fig. 4.9b shows the impact of using multiple VMs in each region to achieve higher aggregate throughput. The black dashed line shows the expected throughput, assuming that bandwidth scales linearly with the number of VMs. Although Skyplane's performance is significantly less than the expected throughput for a large number of gateways, the graph shows that using parallel VMs is an effective way for Skyplane to scale its aggregate bandwidth. Additionally, using parallel VMs is a particularly valuable tool in the context of inter-cloud transfers, as Skyplane can use multiple VMs in one cloud provider to circumvent the egress limit. For example, for an overlay hop from an AWS region to an Azure region, one may allocate many instances in AWS but few in Azure, to account for AWS' egress cap.

Trade-off between cost and throughput Fig. 4.9c shows the impact on overlay path throughput as the price budget is varied. We adjusted the cost budget afforded to the planner (x-axis),

Table 4.2: **Comparison with academic baselines**: Skyplane outperforms RON's path selection heuristic implemented in Skyplane [15].

Method	Time	Throughput	Cost
GCT GridFTP [7, 17] (1 VM)	133s	1.03 Gbps	$1.40
Skyplane (1 VM, direct)	**73s**	**1.71 Gbps**	$1.40
Skyplane w/ RON routes (4 VMs) [15]	21s	6.02 Gbps	$2.27
Skyplane (cost optimized, 4 VMs)	32s	3.88 Gbps	$1.56
Skyplane (throughput optimized, 4 VMs)	**16s**	**8.07 Gbps**	$1.59

and plot the throughput predicted by the planner for the output plan (y-axis). We show three routes where the overlay benefits are considerable (Azure `westus` to AWS `eu-west-1`), good (GCP `asia-east1-a` to AWS `sa-east-1`) and minimal (AWS `af-south-1` to AWS `ap-southeast-2`). As the cost budget increases, Skyplane uses increasingly complex overlay topologies, adding new overlay paths as the instance limit (1 VM, in this case) is saturated in each region. Each elbow in the plot (e.g. 1.2× for the Azure to AWS route) represents a point where Skyplane adds a new overlay route via a faster but more costly region. At some point, the planner cannot increase throughput further as the overlay network is saturated.

Is it better to use VMs to form overlay paths or parallelize the direct path? Given a limited number of VMs (§4.4.3), a natural question is whether it is better to use those VMs to form overlay paths or to parallelize the direct path. In Fig. 4.10, we evaluate Skyplane with and without the overlay enabled for various numbers of VMs in the context of an inter-continental transfer and an intra-continental transfer. For the inter-continental transfer, using the VMs with overlays enabled provides a 2.08× geomean speedup compared to using those VMs to parallelize the direct path. However, for the intra-continental transfer, there is little benefit to using VMs in overlay paths (1.03× geomean speedup).

4.7.6 Comparison against academic baselines

In Table 4.2, we compare Skyplane with RON [15] and the community-maintained fork [17] of GridFTP [7] for a 16 GB data transfer from Azure `East US` to AWS `ap-northeast-1`. To isolate network throughput from I/O overheads, we benchmark the transfers without object stores (VM to VM only).

We use the open-source GCT fork of GridFTP [17]. Although GCT GridFTP theoretically supports striped transfers across multiple machines, we were unable to find a supported noncommercial implementation. To make a fair comparison, we run both GCT GridFTP and Skyplane with a single VM per region. Skyplane is 1.6× faster than GCT GridFTP.

We implement RON's path selection heuristic in Skyplane to compare overlays between RON and Skyplane. Our results show that Skyplane has better cost and throughput than RON. Skyplane with routes from RON's path selection heuristic achieves 3.5× higher throughput than Skyplane with a single VM but at 62% cost overhead. Skyplane's planner instead finds overlay paths with up to 4.7× higher throughput than the direct path within a 14% cost overhead.

4.8 Related Work

Skyplane builds on the overlay network literature [15, 31, 160]. As discussed in §4.1, Skyplane adapts classical overlays to the cloud setting, accounting for the price of network bandwidth and leveraging the elasticity of cloud resources. CRONets [31] briefly discusses cost, but focuses on comparing cloud-based options to private leased lines. Unlike Skyplane, it does not discuss how to manage the cost of cloud resources. Lai et al. [111] find relay regions improve throughput in AWS when utilizing a single TCP connection but find the 2 Gbps instance NIC limit from their chosen instance class limits the benefit of overlay paths. CloudCast [149] examines the use of triangle overlays in the cloud to reduce network latency while Skyplane examines throughput.

Several existing efforts [58, 119, 146] aim to optimize bulk data transfers by reducing the amount of data transferred. Such techniques are complementary to Skyplane; one can first apply these techniques to reduce the amount of data to transfer, and then apply Skyplane's techniques to transfer that reduced data efficiently. Unlike Skyplane, these works do not use cost when selecting the network path to use for a transfer.

Another line of research aims to improve bulk data transfers by improving resource management. GridFTP [7] is a tool for wide-area transfers that techniques such as using multiple machines and TCP connections. GridFTP sends all data over the direct path and does not utilize overlays. Khanna et al. [98] explore application of network overlays to GridFTP but do not consider elasticity and egress price in the cloud. Other solutions, like PSockets [161], also use parallel TCP connections for high bandwidth. Pied Piper [26] also explored how cloud resource elasticity could be used to improve cloud data transfers, but utilize a different mechanism than Skyplane.

There have been decades of improvements and optimizations at the transport layer to make TCP perform better in large-BDP settings within TCP itself [8, 28, 33, 71], while others concern operating system support for TCP [43, 51, 117]. Improvements to TCP are complementary to Skyplane. CodedBulk [168] uses network coding to complete bulk-transfer multicast jobs quickly [168].

Another set of research [35, 175, 182] investigates how to schedule urgent and non-urgent bulk transfers to meet a transfer's deadline. None of these techniques consider the cost of transferring data in the cloud.

Traffic engineering (TE) systems, like Google's B4 [80, 89] and BwE [108] and Microsoft's SWAN [79], Cascara [157], and BlastShield [105], are used internally by cloud providers to navigate the cost-performance trade-off in their wide-area networks. The precise nature of the trade-off differs from Skyplane in two ways. First, TE systems consider costs in terms of the *bandwidth* provisioned (e.g., the cost of installing long-distance cables [89], or the 95th percentile bandwidth for peering links [157]). In contrast, Skyplane considers cost from the perspective of a cloud customer, where the cost depends on the volume and not bandwidth of data transferred. Second, TE systems like Cascara [157] assume a static topology and aim to reallocate bandwidth to save cost, with a global view of a single provider's network. Skyplane optimizes a single user's transfer, with the ability to use overlay regions in multiple cloud providers' networks.

Skyplane has similarities to Content Delivery Networks (CDNs) [160], most notably in that both make use of overlay networks. However, Skyplane's focus is different from CDNs. CDNs focus on caching objects near users, in order to provide low network latency. In contrast, Skyplane focuses on transferring large amounts of data quickly, with a focus on achieving high bandwidth rather than low network latency such as in workloads like ML training and database replication. CDNs are more suitable for workloads where popular objects need to be replicated to many regions so that geo-distributed users can access them with low network latency.

VM migration [42, 70, 90, 110] aims to balance VM downtime and bandwidth consumed when transferring data. Supercloud [90] uses a network of vSwitches in an overlay that maintains TCP connections upon migration, not to provide high bandwidth at low cost.

Some existing research efforts and commercial products focus on bulk transfer jobs that are not time-critical. For example, Laoutaris et al. [112] propose techniques to reduce the cost of transferring data for delay tolerant applications.

Cloud providers provide services for bulk transfer, such as AWS Snowball [155], Azure Data Box [19], and GCP Transfer Appliance [139], that have users ship their data via physical drives via the postal service. For sufficiently large transfers, these services may allow data to be transferred into the cloud datacenter more quickly than using the Internet.

4.9 Conclusion

This paper explores how to efficiently transfer data between cloud regions using cloud-aware overlay networks. Our key observation is that principles from overlay networks can be applied to the cloud setting to identify high-quality network paths that lead to fast transfer times. However, adapting principles from overlay networks to the cloud setting requires consideration of cloud resource pricing, most notably the egress fees associated with network bandwidth. Skyplane manages the trade-off between performance and cost when performing bulk data transfer. It works by accepting a user- or application-provided constraint on performance and solving a mixed integer linear program (MILP) to obtain the optimal data transfer plan. Skyplane can reduce the time to transfer data by up to 5.0× at minimal additional cost.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Rachee Singh, for their helpful feedback. We also thank Asim Biswal, Jason Ding, Daniel Kang, Vincent Liu, Xuting Liu, and Anton Zabreyko. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF GRFP Award DGE-1752814, and gifts from Amazon, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, and VMWare.

Chapter 5

Conclusion

This dissertation aims to address two key challenges when building large language models: robustness and scalability. Chapter 2 addresses robustness through the lens of sensitivity to input perturbations. ContraCode introduced a new method to learn code representations robust to label-preserving transformations. Chapter 3 introduced Checkmate, an algorithm to train neural networks beyond GPU memory capacity limits with optimal rematerialization. Chapter 4 considers another aspect of the scalability problem: the management of large pre-training datasets. Skyplane is a system for bulk data transfer between cloud object stores, thereby enabling ever larger pre-training datasets when training models in the cloud.

There is considerable exciting future directions in both robustness and scalability. While not addressed in this thesis, I previously investigated methods to improve compositional generalization in large language models. This direction remains challenging for state-of-the-art models. Moreover, large language models exhibit hallucination where ungrounded text can be generated containing unfactual information. Explicit memory mechanisms for large-language models remain a promising approach to address this problem. Finally, the scalability of large language models remains a ever present challenge. I am particularly excited about methods to enable training these large models without massive distributed clusters. This would enable academic researchers, such as myself during my time at Berkeley, to train large language models without the need for expensive large-scale cloud infrastructure.

Bibliography

1. M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". en. In: 2016. (Visited on 07/23/2019).

2. W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. "A Transformer-based Approach for Source Code Summarization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 2020, pp. 4998–5007. DOI: 10.18653/v1/2020.acl-main.449. URL: https://aclanthology.org/2020.acl-main.449.

3. M. Allamanis. "The Adverse Effects of Code Duplication in Machine Learning Models of Code". In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2019. Association for Computing Machinery, Athens, Greece, 2019, pp. 143–153. ISBN: 9781450369954. DOI: 10.1145/3359591.3359735.

4. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. "A survey of machine learning for big code and naturalness". In: *ACM Computing Surveys (CSUR)* 51:4, 2018, p. 81.

5. M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao. "Typilus: Neural Type Hints". In: *Programming Language Design and Implementation (PLDI)*. 2020. arXiv: 2004.10657 [cs.PL].

6. M. Allamanis, H. Peng, and C. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. Ed. by M. Balcan and

K. Q. Weinberger. Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016, pp. 2091–2100. URL: http://proceedings.mlr.press/v48/allamanis16.html.

7. W. Allcock, J. Bresnahn, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. "The Globus Striped GridFTP Framework and Server". In: *Supercomputing*. 2005.

8. M. Allman, D. Glover, and L. Sanchez. *Enhancing TCP Over Satellite Channels using Standard Mechanisms*. RFC 2488. 1999.

9. U. Alon, S. Brody, O. Levy, and E. Yahav. "code2seq: Generating Sequences from Structured Representations of Code". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=H1gKYo09tX.

10. Amazon Web Services. *Amazon EC2 instance network bandwidth*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html. 2022.

11. Amazon Web Services. *AWS DataSync: online data transfer and migration*. https://aws.amazon.com/datasync. 2022.

12. Amazon Web Services. *EC2 On-Demand Instance Pricing*. https://aws.amazon.com/ec2/pricing/on-demand/. 2022.

13. Amazon Web Services. *General purpose instances*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html#general-purpose-network-performance. 2022.

14. M. C. an. "Evaluating Large Language Models Trained on Code". In: *ArXiv preprint* abs/2107.03374, 2021. URL: https://arxiv.org/abs/2107.03374.

15. D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. "Resilient Overlay Networks". In: *SOSP*. ACM, 2001.

16. M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics". In: *CIDR*. 2021.

17. G. authors. *Grid community toolkit*. https://github.com/gridcf/gct. 2022.

18. M. Azure. *Copy Blob From URL*. https://docs.microsoft.com/en-us/rest/api/storageservices/copy-blob-from-url. 2022.

19. M. Azure. *Microsoft Azure Data Box*. https://azure.microsoft.com/en-us/products/databox/. 2022.

20. M. Azure. *Scalability and performance targets for Blob storage*. https://learn.microsoft.com/en-us/azure/storage/blobs/scalability-targets. 2022.

21. B. S. Baker. "A program for identifying duplicated code". In: *Computing Science and Statistics*, 1992.

22. S. Bansal and A. Aiken. "Automatic Generation of Peephole Superoptimizers". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. Association for Computing Machinery, San Jose, California, USA, 2006, pp. 394–403. ISBN: 1595934510. DOI: 10.1145/1168857.1168906.

23. O. Beaumont, J. Herrmann, G. Pallez, and A. Shilova. *Optimal Memory-aware Backpropagation of Deep Join Networks*. Research Report RR-9273. Inria, 2019.

24. T. Ben-Nun, A. S. Jakobovits, and T. Hoefler. "Neural Code Comprehension: A Learnable Representation of Code Semantics". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. 2018, pp. 3589–3601. URL: https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html.

25. S. Benton, A. Ghanbari, and L. Zhang. "Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages". In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 47–50.

26. A. Bergman, I. Cidon, I. Keslassy, N. Rotman, M. Schapira, A. Markuze, and E. Zohar. "Pied Piper: Rethinking Internet Data Delivery". In: *CoRR*. 2018. URL: https://arxiv.org/abs/1812.05582.

27. P. Bielik and M. T. Vechev. "Adversarial Robustness for Code". In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 896–907. URL: http://proceedings.mlr.press/v119/bielik20a.html.

28. D. Borman, B. Braden, and V. Jacobson. *TCP Extensions for High Performance*. RFC 7323. 2014.

29. P. Briggs, K. D. Cooper, and L. Torczon. "Rematerialization". In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI '92. New York, NY, USA, 1992, pp. 311–321. (Visited on 07/22/2019).

30. S. R. Bulo, L. Porzi, and P. Kontschieder. “In-place Activated BatchNorm for Memory-Optimized Training of DNNs”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 2018, pp. 5639–5647.

31. C. X. Cai, F. Le, X. Sun, G. G. Xie, H. Jamjoom, and R. H. Campbell. “CRONets: Cloud-Routed Overlay Networks”. In: *ICDCS*. IEEE, 2016.

32. A. Canziani, A. Paszke, and E. Culurciello. “An Analysis of Deep Neural Network Models for Practical Applications”. In: 2016. arXiv: 1605.07678. (Visited on 09/01/2019).

33. N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. “BBR: congestion-based congestion control”. In: *CACM*. 2017.

34. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. “Register allocation via coloring”. In: *Computer Languages* 6:1, 1981, pp. 47–57. (Visited on 07/22/2019).

35. B. B. Chen and P. V.-B. Primet. “Scheduling deadline-constrained bulk data transfers to minimize network congestion”. In: *CCGrid*. IEEE, 2007.

36. L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”. In: 2016. arXiv: 1606.00915. (Visited on 09/02/2019).

37. T. Chen, B. Xu, C. Zhang, and C. Guestrin. “Training Deep Nets with Sublinear Memory Cost”. In: 2016. arXiv: 1604.06174. (Visited on 02/22/2019).

38. T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton. “A Simple Framework for Contrastive Learning of Visual Representations”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 1597–1607. URL: http://proceedings.mlr.press/v119/chen20j.html.

39. X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. “Multi-View 3D Object Detection Network for Autonomous Driving”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, Honolulu, HI, 2017, pp. 6526–6534. (Visited on 09/09/2019).

40. X. Chen, H. Fan, R. Girshick, and K. He. “Improved baselines with momentum contrastive learning”. In: *ArXiv preprint* abs/2003.04297, 2020. URL: https://arxiv.org/abs/2003.04297.

41. R. Child, S. Gray, A. Radford, and I. Sutskever. “Generating Long Sequences with Sparse Transformers”. In: 2019. arXiv: 1904.10509. (Visited on 08/20/2019).

42. C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. “Live Migration of Virtual Machines”. In: *NSDI*. 2005.

43. D. D. Clark. “The Structuring of Systems Using Upcalls”. In: *SOSP*. 1985.

44. A. contributors. *Azure Storage AzCopy*. `https://github.com/Azure/azure-storage-azcopy`. 2022.

45. T. contributors. *Training ResNet on Cloud TPU*. `https://cloud.google.com/tpu/docs/tutorials/resnet`. 2022.

46. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13:4, 1991, pp. 451–490. (Visited on 07/22/2019).

47. Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. “Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context”. In: 2019. arXiv: 1901.02860. (Visited on 08/13/2019).

48. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: 2018. arXiv: 1810.04805. (Visited on 08/13/2019).

49. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://aclanthology.org/N19-1423`.

50. C. Dong, C. C. Loy, K. He, and X. Tang. “Image Super-Resolution Using Deep Convolutional Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38:2, 2016, pp. 295–307.

51. P. Druschel and L. L. Peterson. “Fbufs: A High-bandwidth Cross-domain Transfer Facility”. In: *SOSP*. 1993.

52. J. Feng and D. Huang. “Cutting Down Training Memory by Re-fowarding”. In: 2018. (Visited on 03/11/2019).

53. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: https://aclanthology.org/2020.findings-emnlp.139.

54. R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy. "A public unified bug dataset for Java". In: *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. 2018, pp. 12–21.

55. S. Floyd and K. Fall. "Promoting the Use of End-to-End Congestion Control in the Internet". In: *Trans. Networking*, 1999.

56. J. J. Forrest, S. Vigerske, T. Ralphs, H. G. Santos, L. Hafer, B. Kristjansson, J. Fasano, E. Straver, M. Lubin, rlougee, jpgoncal1, H. I. Gassmann, and M. Saltzman. *COIN-OR Branch-and-Cut solver*. 2019. DOI: 10.5281/zenodo.3246628. (Visited on 10/03/2019).

57. Forrester/Virtustream. *A Clear Multicloud Strategy Delivers Business Value*.

58. S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch. "McCAT: Multi-cloud Cost-aware Transport". In: *EuroSys Poster Track*. 2014.

59. J. Gettys and K. Nichols. "Bufferbloat: Dark buffers in the Internet". In: *CACM*, 2012.

60. A. Gholami, A. Azad, P. Jin, K. Keutzer, and A. Buluc. "Integrated model, batch, and domain parallelism in training neural networks". In: *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM. 2018, pp. 77–86.

61. *GNU Project - Free Software Foundation (FSF)*. (Visited on 09/09/2019).

62. A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse. "The Reversible Residual Network: Backpropagation Without Storing Activations". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 2214–2224. (Visited on 02/22/2019).

63. D. W. Goodwin and K. D. Wilken. "Optimal and Near-optimal Global Register Allocation Using 0–1 Integer Programming". en. In: *Software: Practice and Experience* 26:8, 1996, pp. 929–965. (Visited on 06/25/2019).

64. Google Cloud. *All networking pricing | Virtual Private Cloud | Google Cloud*. https://cloud.google.com/vpc/network-pricing. 2022.

65. Google Cloud. *Network Bandwidth | Compute Engine Documentation | Google Cloud*. https://cloud.google.com/compute/docs/network-bandwidth. 2022.

66. Google Cloud Platform. *Storage Transfer Service*. `https://cloud.google.com/storage-transfer-service`. 2022.

67. A. Griewank and A. Walther. “Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation”. In: *ACM Transactions on Mathematical Software* 26:1, 2000, pp. 19–45. (Visited on 09/06/2019).

68. A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves. “Memory-efficient Backpropagation Through Time”. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS’16. Curran Associates Inc., USA, 2016, pp. 4132–4140. (Visited on 08/20/2019).

69. L. Gueguen, A. Sergeev, B. Kadlec, R. Liu, and J. Yosinski. “Faster Neural Networks Straight from JPEG”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 3933–3944. (Visited on 07/23/2019).

70. K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan. “You Can Teach Elephants to Dance: Agile VM Handoff for Edge Computing”. In: *SEC*. 2017.

71. S. Ha, I. Rhee, and L. Xu. “CUBIC: A New TCP-Friendly High-Speed TCP Variant”. In: *SIGOPS*. 2008.

72. R. Hadsell, S. Chopra, and Y. LeCun. “Dimensionality reduction by learning an invariant mapping”. In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’06)*. Vol. 2. IEEE. 2006, pp. 1735–1742.

73. K. He, H. Fan, Y. Wu, S. Xie, and R. B. Girshick. “Momentum Contrast for Unsupervised Visual Representation Learning”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. IEEE, 2020, pp. 9726–9735. DOI: `10.1109/CVPR42600.2020.00975`. URL: `https://doi.org/10.1109/CVPR42600.2020.00975`.

74. K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

75. V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis. “Deep learning type inference”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 152–162.

76. O. J. Hénaff. “Data-Efficient Image Recognition with Contrastive Predictive Coding”. In: *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event.* Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 4182–4192. URL: http://proceedings.mlr.press/v119/henaff20a.html.

77. D. Hendrycks, M. Mazeika, S. Kadavath, and D. Song. “Using Self-Supervised Learning Can Improve Model Robustness and Uncertainty”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.* Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett. 2019, pp. 15637–15648. URL: https://proceedings.neurips.cc/paper/2019/hash/a2b15837edac15df90721968986f7f8e-Abstract.html.

78. D. Hernandez, J. Kaplan, T. Henighan, and S. McCandlish. *Scaling Laws for Transfer.* 2021. arXiv: 2102.01293 [cs.LG].

79. C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nandury, and R. Wattenhofer. “Achieving High Utilization with Software-Driven WAN”. In: *SIGCOMM.* 2013.

80. C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat. “B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google’s Software-Defined WAN”. In: *SIGCOMM.* 2018.

81. G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2017, pp. 4700–4708.

82. Z. Huang, W. Xu, and K. Yu. “Bidirectional LSTM-CRF models for sequence tagging”. In: *ArXiv preprint* abs/1508.01991, 2015. URL: https://arxiv.org/abs/1508.01991.

83. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search”. In: *ArXiv preprint* abs/1909.09436, 2019. URL: https://arxiv.org/abs/1909.09436.

84. A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry. “Adversarial Examples Are Not Bugs, They Are Features”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada.* Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. B. Fox, and R. Garnett. 2019, pp. 125–136. URL: https://

proceedings.neurips.cc/paper/2019/hash/e2c420d928d4bf8ce0ff2ec19b371514-Abstract.html.

85. S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International Conference on Machine Learning*, 2015. (Visited on 07/19/2019).

86. S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. “Summarizing Source Code using a Neural Attention Model”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 2016, pp. 2073–2083. DOI: 10.18653/v1/P16-1195. URL: https://aclanthology.org/P16-1195.

87. A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. “Gist: Efficient Data Encoding for Deep Neural Network Training”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. IEEE Press, Piscataway, NJ, USA, 2018, pp. 776–789. (Visited on 07/30/2019).

88. P. Jain, X. Mo, A. Jain, A. Tumanov, J. E. Gonzalez, and I. Stoica. “The OoO VLIW JIT Compiler for GPU Inference”. In: *arXiv preprint arXiv:1901.10008*, 2019.

89. S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. “B4: Experience with a Globally-Deployed Software Defined WAN”. In: *SIGCOMM*. 2013.

90. Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. “Supercloud: Opportunities and Challenges”. In: *SIGOPS*. 2015.

91. Z. Jia, S. Lin, C. R. Qi, and A. Aiken. “Exploring Hidden Dimensions in Accelerating Convolutional Neural Networks”. en. In: *International Conference on Machine Learning*. 2018, pp. 2274–2283. (Visited on 09/01/2019).

92. Z. Jia, M. Zaharia, and A. Aiken. “Beyond Data and Model Parallelism for Deep Neural Networks”. In: *SysML Conference*, 2018, p. 13.

93. R. Joshi, G. Nelson, and K. Randall. “Denali: A Goal-Directed Superoptimizer”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Association for Computing Machinery, Berlin, Germany, 2002, pp. 304–314. ISBN: 1581134630. DOI: 10.1145/512529.512566.

94. A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. “Pre-trained Contextual Embedding of Source Code”. In: *ArXiv preprint* abs/2001.00059, 2020. URL: https://arxiv.org/abs/2001.00059.

95. S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. “Calendaring for Wide Area Networks”. In: *SIGCOMM*. ACM, 2014.

96. N. Karmarkar. “A New Polynomial-Time Algorithm for Linear Programming”. In: *STOC*. 1984.

97. N. Karmarkar. “A new polynomial-time algorithm for linear programming”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM. 1984, pp. 302–311.

98. G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, I. Foster, and J. Saltz. “Using overlays for efficient data transfer over shared wide-area networks”. In: *Supercomputing*. 2008.

99. J. Kim, J. K. Lee, and K. M. Lee. “Accurate Image Super-Resolution Using Very Deep Convolutional Networks”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 1646–1654. DOI: 10.1109/CVPR.2016.182.

100. M. Kim, T. Zimmermann, and N. Nagappan. “A field study of refactoring challenges and benefits”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 2012, pp. 1–11.

101. M. Kirisame, S. Lyubomirsky, A. Haan, J. Brennan, M. He, J. Roesch, T. Chen, and Z. Tatlock. “Dynamic Tensor Rematerialization”. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=Vfs_2RnOD0H.

102. D. R. Koes and S. C. Goldstein. “A Global Progressive Register Allocator”. In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’06. event-place: Ottawa, Ontario, Canada. ACM, New York, NY, USA, 2006, pp. 204–215. (Visited on 07/22/2019).

103. J. Koppel, V. Premtoon, and A. Solar-Lezama. “One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax”. In: *Proc. ACM Program. Lang.* 2:OOPSLA, 2018. DOI: 10.1145/3276492. URL: https://doi.org/10.1145/3276492.

104. D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. “Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh”. In: *SOSP*, 2003.

105. U. Krishnaswamy, R. Singh, N. Bjørner, and H. Raj. “Decentralized cloud wide-area network traffic engineering with BlastShield”. In: *NSDI*. USENIX, 2022.

106. A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. (Visited on 08/20/2019).

107. T. Kudo. “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 2018, pp. 66–75. DOI: 10.18653/v1/P18-1007. URL: https://aclanthology.org/P18-1007.

108. A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. “BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing”. In: *SIGCOMM*. 2015.

109. J. F. Kurose and K. W. Ross. “Computer Networking: A Top-Down Approach”. In: International 6th. 2013. Chap. 3, p. 308.

110. H. A. Lagar-Cavilla, J. A. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. M. Rumble, M. Satyanarayanan, and A. Scannell. “SnowFlock: Virtual Machine Cloning as a First-Class Cloud Primitive”. In: *ACM Trans. Comput. Syst.* 29:1, 2011. ISSN: 0734-2071. DOI: 10.1145/1925109.1925111. URL: https://doi.org/10.1145/1925109.1925111.

111. F. Lai, M. Chowdhury, and H. Madhyastha. “To Relay or Not to Relay for Inter-Cloud Transfers?” In: *HotCloud*. 2018.

112. N. Laoutaris, G. Smaragdakis, P. Rodriguez, and R. Sundaram. “Delay Tolerant Bulk Data Transfers on the Internet”. In: *SIGMETRICS*. 2009.

113. C. Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002.

114. Y. Liu, M. Ott, N. Goyal, and J. D. an. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: *ArXiv preprint* abs/1907.11692, 2019. URL: https://arxiv.org/abs/1907.11692.

115. Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. “RoBERTa: A Robustly Optimized BERT Pretraining Approach”. In: 2019. arXiv: 1907.11692. (Visited on 08/13/2019).

116. R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte. "Combinatorial Register Allocation and Instruction Scheduling". In: 2018. arXiv: 1804.02452. (Visited on 07/19/2019).

117. C. Maeda and B. N. Bershad. "Protocol Service Decomposition for High-performance Networking". In: *SOSP*. 1993.

118. H. Massalin. "Superoptimizer: A Look at the Smallest Program". In: *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*. ASPLOS II. IEEE Computer Society Press, Palo Alto, California, USA, 1987, pp. 122–126. ISBN: 0818608056. DOI: `10.1145/36206.36194`.

119. M. Matos, A. Sousa, J. Pereira, and R. Oliveira. "CLON: Overlay Network for Clouds". In: *WDDM*. 2009.

120. S. McCandlish, J. Kaplan, D. Amodei, and O. D. Team. "An Empirical Model of Large-Batch Training". In: 14, 2018. arXiv: 1812.06162. arXiv: `1812.06162 [cs, stat]`. (Visited on 09/09/2019).

121. S. McKenzie et al. *Babel: compiler for writing next generation JavaScript*. `https://github.com/babel/babel`. 2020.

122. C. Mendis, A. Renda, S. P. Amarasinghe, and M. Carbin. "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 4505–4515. URL: `http://proceedings.mlr.press/v97/mendis19a.html`.

123. C. Meng, M. Sun, J. Yang, M. Qiu, and Y. Gu. "Training Deeper Models by GPU Memory Optimization on TensorFlow". en. In: 2017, p. 8.

124. P. Micikevicius. *Local Memory and Register Spilling*. en. 2011.

125. Microsoft Azure. *Pricing - Bandwidth | Microsoft Azure*. `https://azure.microsoft.com/en-us/pricing/details/bandwidth/`. 2022.

126. Microsoft Azure. *Scalability and performance targets for Blob storage*. `https://docs.microsoft.com/en-us/azure/storage/blobs/scalability-targets`. 2021.

127. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. “Distributed Representations of Words and Phrases and their Compositionality”. In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.* Ed. by C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger. 2013, pp. 3111–3119. URL: https://proceedings.neurips.cc/paper/2013/hash/9aa42b31882ec039965f3c4923ce901b-Abstract.html.

128. D. Movshovitz-Attias and W. W. Cohen. “Natural Language Models for Predicting Programming Comments”. In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers).* Association for Computational Linguistics, Sofia, Bulgaria, 2013, pp. 35–40. URL: https://aclanthology.org/P13-2007.

129. Y. Nesterov and A. Nemirovskii. *Interior-point polynomial algorithms in convex programming.* Vol. 13. Siam, 1994.

130. NVIDIA. *NVIDIA Tesla V100 GPU Architecture.* en. 2017. (Visited on 07/22/2019).

131. E. Nygren, R. K. Sitaraman, and J. Sun. “The Akamai Network: A Platform for High-Performance Internet Applications”. In: *SIGOPS*, 2010.

132. J. S. Olesen. *Register Allocation in LLVM 3.0.* en. 2011.

133. A. v. d. Oord, Y. Li, and O. Vinyals. “Representation learning with contrastive predictive coding”. In: *ArXiv preprint* abs/1807.03748, 2018. URL: https://arxiv.org/abs/1807.03748.

134. J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. “Modeling TCP Throughput: A Simple Model and Its Empirical Validation”. In: *Proceedings of the ACM SIGCOMM ’98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication.* SIGCOMM ’98. Association for Computing Machinery, Vancouver, British Columbia, Canada, 1998, pp. 303–314. ISBN: 1581130031. DOI: 10.1145/285237.285291. URL: https://doi.org/10.1145/285237.285291.

135. I. V. Pandi, E. T. Barr, A. D. Gordon, and C. Sutton. *OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints.* 2020. arXiv: 2004.00348 [cs.PL].

136. A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. “Automatic differentiation in PyTorch”. In: *NIPS 2017 Autodiff Workshop.* 2017.

137. A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035.

138. S. G. Patil, P. Jain, P. Dutta, I. Stoica, and J. Gonzalez. "POET: Training Neural Networks on Tiny Devices with Integrated Rematerialization and Paging". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 17573–17583.

139. G. C. Platform. *Google Cloud Transfer Appliance*. https://cloud.google.com/transfer-appliance/docs/4.0/overview. 2022.

140. T. Pohlen, A. Hermans, M. Mathias, and B. Leibe. "Full-Resolution Residual Networks for Semantic Segmentation in Street Scenes". In: *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. 2017.

141. M. Pradel, G. Gousios, J. Liu, and S. Chandra. "TypeWriter: Neural Type Prediction with Search-based Validation". In: *ArXiv preprint* abs/1912.03768, 2019. URL: https://arxiv.org/abs/1912.03768.

142. M. Pradel and K. Sen. "DeepBugs: A learning approach to name-based bug detection". In: *Proceedings of the ACM on Programming Languages* 2:OOPSLA, 2018, pp. 1–25.

143. Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica. "Low Latency Geo-distributed Data Analytics". In: *SIGCOMM*. 2015.

144. M. Punjani. "Register Rematerialization in GCC". In: *GCC Developers' Summit*. Vol. 2004. Citeseer, 2004.

145. M. R. I. Rabin and M. A. Alipour. *Evaluation of Generalizability of Neural Program Analyzers under Semantic-Preserving Transformations*. 2020. arXiv: 2004.07313 [cs.SE].

146. A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman. "Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area". In: *NSDI*. 2014.

147. O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". en. In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24574-4.

148. B. K. Rosen, M. N. Wegman, and F. K. Zadeck. “Global Value Numbers and Redundant Computations”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL ’88. ACM, New York, NY, USA, 1988, pp. 12–27. (Visited on 07/22/2019).

149. N. H. Rotman, Y. Ben-Itzhak, A. Bergman, I. Cidon, I. Golikov, A. Markuze, and E. Zohar. “CloudCast: Characterizing Public Clouds Connectivity”. In: *CoRR*, 2022.

150. F. Santos et al. *Terser: JavaScript parser, mangler and compressor toolkit for ES6+.* https://github.com/terser/terser. 2020.

151. N. Saunshi, O. Plevrakis, S. Arora, M. Khodak, and H. Khandeparkar. “A Theoretical Analysis of Contrastive Unsupervised Representation Learning”. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA.* Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 5628–5637. URL: http://proceedings.mlr.press/v97/saunshi19a.html.

152. M. Schuster and K. Paliwal. “Bidirectional recurrent neural networks”. In: *Signal Processing, IEEE Transactions on* 45, 1997, pp. 2673–2681. DOI: 10.1109/78.650093.

153. R. Schuster, C. Song, E. Tromer, and V. Shmatikov. “You autocomplete me: Poisoning vulnerabilities in neural code completion”. In: *30th {USENIX} Security Symposium ({USENIX} Security 21).* 2021.

154. A. W. Services. *Amazon Bottlerocket OS.* https://aws.amazon.com/bottlerocket. 2022.

155. A. W. Services. *AWS Snowball.* https://aws.amazon.com/snowball. 2022.

156. K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: 2014. arXiv: 1409.1556.

157. R. Singh, S. Agarwal, M. Calder, and P. Bahl. “Cost-Effective Cloud Edge Traffic Engineering with CASCARA”. In: *NSDI.* 2021.

158. J. M. Siskind and B. A. Pearlmutter. “Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation”. In: *Optimization Methods and Software* 33:4-6, 2018, pp. 1288–1330. (Visited on 08/05/2019).

159. J. M. Siskind and B. A. Pearlmutter. “Divide-and-conquer checkpointing for arbitrary programs with no user annotation”. In: *Optimization Methods and Software* 33:4-6, 2018, pp. 1288–1330. DOI: 10.1080/10556788.2018.1459621. eprint: https://doi.org/10.1080/10556788.2018.1459621.

160. R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain. “Overlay networks: An Akamai perspective”. In: *Advanced Content Delivery, Streaming, and Cloud Services*, 2014.

161. H. Sivakumar, S. Bailey, and R. L. Grossman. “PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks”. In: *Supercomputing*. ACM/IEEE, 2000.

162. M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou. “Astra: Exploiting Predictability to Optimize Deep Learning”. en. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '19*. ACM Press, Providence, RI, USA, 2019, pp. 909–923. (Visited on 09/02/2019).

163. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”. In: *SIGCOMM*. 2001.

164. J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. “Towards a Big Data Curated Benchmark of Inter-Project Code Clones”. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. ICSME '14. IEEE Computer Society, USA, 2014, pp. 476–480. ISBN: 9781479961467. DOI: 10.1109/ICSME.2014.77. URL: https://doi.org/10.1109/ICSME.2014.77.

165. V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. “Efficient processing of deep neural networks: A tutorial and survey”. In: *Proceedings of the IEEE* 105:12, 2017, pp. 2295–2329.

166. Y. Tai, J. Yang, and X. Liu. “Image Super-Resolution via Deep Recursive Residual Network”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2790–2798. DOI: 10.1109/CVPR.2017.298.

167. W. L. Taylor. ““Cloze procedure”: A new tool for measuring readability”. In: *Journalism Quarterly* 30:4, 1953, pp. 415–433.

168. S.-H. Tseng, S. Agarwal, R. Agarwal, H. Ballani, and A. Tang. “CodedBulk: Inter-Datacenter Bulk Transfers using NetworkCoding”. In: *NSDI*. 2021.

169. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008. (Visited on 07/19/2019).

170. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett. 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

171. K. Wang and M. Christodorescu. “COSET: A benchmark for evaluating neural program embeddings”. In: *ArXiv preprint* abs/1905.11445, 2019. URL: https://arxiv.org/abs/1905.11445.

172. K. Wang and Z. Su. “Learning blended, precise semantic program embeddings”. In: *ArXiv*, 2019.

173. J. Wei, M. Goyal, G. Durrett, and I. Dillig. “LambdaNet: Probabilistic Type Inference using Graph Neural Networks”. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=Hkx6hANtwH.

174. M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. “Deep learning code fragments for code clone detection”. In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2016, pp. 87–98.

175. Y. Wu, Z. Zhang, C. Wu, C. Guo, Z. Li, and F. C. M. Lau. “Orchestrating Bulk Data Transfers across Geo-Distributed Datacenters”. In: *Trans. Cloud Computing*.

176. Y. Wu and K. He. “Group Normalization”. In: 22, 2018, pp. 3–19. (Visited on 09/09/2019).

177. Z. Wu, Y. Xiong, S. X. Yu, and D. Lin. “Unsupervised Feature Learning via Non-Parametric Instance Discrimination”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 2018, pp. 3733–3742. DOI: 10.1109/CVPR.2018.00393. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Wu%5C_Unsupervised%5C_Feature%5C_Learning%5C_CVPR%5C_2018%5C_paper.html.

178. B. Yang, M. Liang, and R. Urtasun. “HDNET: Exploiting HD Maps for 3D Object Detection”. In: 2018, p. 10.

179. M. Yannakakis. “On the approximation of maximum satisfiability”. In: *Journal of Algorithms* 17:3, 1994, pp. 475–502.

180. N. Yefet, U. Alon, and E. Yahav. "Adversarial Examples for Models of Code". In: *ArXiv preprint* abs/1910.07517, 2019. URL: https://arxiv.org/abs/1910.07517.

181. N. Yefet, U. Alon, and E. Yahav. "Adversarial examples for models of code". In: *Proceedings of the ACM on Programming Languages* 4:OOPSLA, 2020, pp. 1–30.

182. H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang. "Guaranteeing Deadlines for Inter-Datacenter Transfers". In: *EuroSys*. 2015.

183. Y. Zhang, J. Jiang, K. Xu, X. Nie, M. J. Reed, H. Wang, G. Yao, M. Zhang, and K. Chen. "BDS: A Centralized near-Optimal Overlay Network for Inter-Datacenter Data Replication". In: *EuroSys*. 2018.