Cybertron: Pushing the Limit on I/O Reduction in Data-Parallel Programs

Tian Xiao[1,2] Zhenyu Guo[2] Hucheng Zhou[2] Jiaxing Zhang[2] Xu Zhao[1,2]
Chencheng Ye[3,2] Xi Wang[2,4] Wei Lin[2] Wenguang Chen[1] Lidong Zhou[2]

[1]Tsinghua University [2]Microsoft Research
[3]Huazhong University of Science and Technology [4]MIT CSAIL

Abstract

I/O reduction has been a major focus in optimizing data-parallel programs for big-data processing. While the current state-of-the-art techniques use static program analysis to reduce I/O, Cybertron proposes a new direction that incorporates runtime mechanisms to push the limit further on I/O reduction. In particular, Cybertron tracks how data is used in the computation accurately at runtime to filter unused data at finer granularity dynamically, beyond what current static-analysis based mechanisms are capable of, and to facilitate a new mechanism called *constraint based encoding* for more efficient encoding. Cybertron has been implemented and applied to production data-parallel programs; our extensive evaluations on real programs and real data have shown its effectiveness on I/O reduction over the existing mechanisms at reasonable CPU cost, and its improvement on end-to-end performance in various network environments.

1. Introduction

MapReduce-style [11] data-parallel programs for big-data processing often involve multiple stages of computation separated by cross-stage data transfer (e.g., between a map stage and the subsequent reduce stage). A user-defined function is used to describe the data-processing logic for each stage. Examples of such data-parallel systems include Hadoop [2], Dryad [16], Pig [22], Hive [25] and SCOPE [32]. The execution of data-parallel programs is often I/O intensive and incurs a significant amount of cross-stage I/O, which has been a major target for optimizations [13, 14, 17, 21, 28, 30, 31].

High-level SQL-like languages [22, 25, 27, 32] have been proposed for data-parallel programs, so that relational-algebra based database optimizations, such as early filtering and projection, can be applied. The user-defined functions in the data-parallel programs often limit the effectiveness of these optimization techniques because of the inherent diversity and complexity of the processing logic in those functions.

The more recent work [13, 17] proposes to apply static program analysis to user-defined functions. Static program analysis can help identify the fields that are not used by a user-defined function or extract from the imperative code the relational operations, such as projection and selection, to enable deeper relational optimizations even with user-defined functions, resulting in fewer records and fields being transmitted. PeriSCOPE [14] further rewrites user-defined functions by partitioning it and moving one part across the network to minimize the cross-stage I/O through *smart cut*.

Even though the static approach has been proven effective, our experiences with real production data-parallel jobs have revealed the limitations of this approach. First, we observe that user-defined functions often contain data-dependent *loop* and *conditional* processing logic, where the value of one data portion in a data entry influences the use of other portions in further data processing; for example, the number of ';'s in a string determines the number of iterations of a loop that does further data processing; a field A might not be used if another field B has a certain value V. Analyzing such constructs to identify optimization opportunities is fundamentally beyond the capabilities of static program analysis. Second, static approaches such as smart cut involve estimating data sizes in order to find an optimal point. The accuracy of such estimation (e.g., on the size of a string) is again fundamentally limited by the static approach, leading to suboptimal choices. Finally, we find that in many cases even for data that are explicitly used in a user-defined function the actual raw values might not be necessary. Instead, any value that satisfies certain constraints and leads to the same execution result can be used. Collecting the constraints, and replacing the raw value with a different value that satisfies the collected constraints and

```
b123-456,107,WEB,ACTION,2013-01-20.12:00:00,b123,ACTION,ip=192.168.0.1;scheduler=611;flags=0x4C0;action=b123,
MUID,b123-456,ANID,b123-456,EXT_DATA,len=33;data=v3/atz.prospectID/atc1.productID
```

Figure 1. A sample log entry generated by a production on-line advertisement system.

is more friendly to data encoding, can often reduce network I/O.

These limitations makes the static partitioning approach insufficient because the optimal partition can vary on different input data. In addition, an optimal partition for certain data may even increase network I/O for other data. So static approaches such as smart cut must be conservative and miss optimization opportunities. A possible workaround is to prepare multiple versions of partitioned code and decide which to use by investigating data at runtime. However, the number of code versions can be huge for complex user-defined functions, making this approach feasible only for small programs.

We therefore propose Cybertron, a novel dynamic approach to I/O reduction for data-parallel programs, which leaves the user-defined function untouched while exploiting different opportunities for different data. Specifically, Cybertron generates a shadow program that runs on the data sender side, which incorporates runtime mechanisms that are capable of understanding string operations, dynamically inspects which data fields are unused, collects the constraints that the data must satisfy in order for the code to produce the same result as on the original data, and constructs new data with more efficient encoding while preserving correctness. Through a combination of static analysis and runtime techniques, Cybertron reduces data size with respect to a user-defined function in two *execution-equivalent* ways. First, Cybertron attempts to exclude data regions by pre-executing conditions that would govern their use in the computation, which is called *unused-data elimination*. Second, Cybertron extracts constraints that govern how the computation will use data regions to enable a more space-efficient *constraint based data encoding*. Cybertron further introduces the key technique of *constraint concretization* to enable efficient constraint based encoding and decoding without expensive constraint solving.

The paper makes the following contributions. First, Cybertron represents a new class of I/O-reduction optimizations that incorporates dynamic mechanisms, which is beyond the capabilities of the state-of-the-art techniques and is shown to be effective on production data-parallel programs. Second, Cybertron introduces a novel concept of constraint based encoding based on the notion of execution equivalence. Third, the paper describes a practical realization of Cybertron that has been implemented and extensively evaluated on both its effectiveness on I/O reduction and its impact on end-to-end performance in various network environments.

The rest of the paper is organized as follows. Section 2 illustrates several examples that motivate our work. Section 3 describes the detailed design. Implementation details are the subject of Section 4, followed by our evaluations in the context of some MapReduce programs extracted from production systems in Section 5. We survey the related work in Section 6, and conclude in Section 7.

WorkflowName	ReportKey	Payload
`Build`	`StartTime`	`1/1/2013 1:00:00 AM`
`Build`	`FileName`	~~`some/long/path`~~
`Build`	`EndTime`	`1/1/2013 2:00:00 AM`
`Build`	`FileName`	~~`another/long/path`~~

Table 1. Three fields in four rows from an internal workflow log. Confidential fields are filled with fake values.

2. Motivating Examples

This section illustrates the I/O reduction opportunities beyond the capability of the current static-analysis based approaches, using real job snippets from a production MapReduce cluster. The opportunities motivate a hybrid approach with both static and dynamic techniques, leads to the novel concept of execution-equivalent encoding, and helps to shape the design of Cybertron.

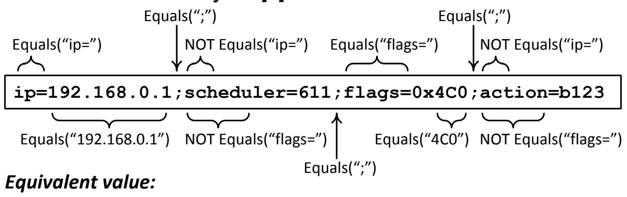
Field dependencies and conditionally unused data. Data in a field may be only used by a user-defined function when the value of another field in the same row satisfies a certain condition. For example, Table 1 shows four internal workflow log entries with the same `WorkflowName` from our production environment. A reducer aggregates the logs by `WorkflowName`, computes the duration between `StartTime` and `EndTime`, and counts the number of related files. Whether data in field `Payload` is used or not depends on the string in field `ReportKey`. In this example, the concrete file names in the 2nd and 4th row are not used. To remove such conditionally unused data, it is necessary to check at runtime whether `ReportKey` equals "`StartTime`" or "`EndTime`", while a static approach can only conservatively consider that the `Payload` field is always used by the user-defined function.

Ad hoc string parsing and constraint based encoding. We observed that log data, which is a major class of data in production settings, tends to be semi-structured or even unstructured in nature. A user-defined function consuming such log data usually uses various string operations to locate and extract those portions of interest. What matters to the result of the processing can often be summarized as a set of constraints, not necessarily the actual raw values. It is often possible to construct an alternative piece of data that satisfies the same set of constraints, while allowing a more efficient encoding. Such constraint based encoding preserves

Code:

```
string flags = null;
string ip = null;
foreach (string token in fields[7].Split(";")) {
  if (token.StartsWith("flags="))
    flags = token.Substring(8);
  if (token.StartsWith("ip="))
    ip = token.Substring(3);
}
```

Essential constraints on fields[7]:



Equivalent value:

```
ip=192.168.0.1;$$$$$$$$$$$$$;flags=$$4C0;$$$$$$$$$$$
```

Figure 2. A code snippet to extract the IP address and flags from the 8th field (`fields[7]`, marked by underline) in the log in Figure 1. Both the original string and the alternative string satisfy the same set of constraints that matter to the execution; replacing the original string with the "equivalent value" allows more efficient encoding, while preserving correctness. Some constraints are not shown due to space limitation.

correctness of the computation because all the necessary constraints are satisfied, while offering opportunities for I/O reduction through better encoding. Collecting the constraints imposes challenges for static program analysis and requires dynamic information.

Figure 1 shows an example entry from a log generated by a production on-line advertisement system to record events related to impressions, clicks, conversions, and so on. The log entry consists of comma-separated fields and some fields have nested structures. The first 8 fields (the first line in the figure) are mandatory, while the rest are optional properties as key-value pairs, with key and value occupying two fields separately. Figure 2 shows a code snippet in a mapper to extract the IP address and flags from the 8th field (`fields[7]`, marked by underline in Figure 1). The constraints that matter to the execution of the code are marked on the data shown in Figure 2. (We omit the constraints `NOT Equals(";")` for all the non-`";"` positions.) One can construct an equivalent value for the string that satisfies these constraints. Replacing the original value with this equivalent one does not change the mapper's output, but can bring I/O reduction because the new string can clearly be encoded more efficiently. Note that the iteration count of the loop depends on the concrete value of `fields[7]` and may vary for different log entries. Understanding such a loop is beyond the capability of static program analysis in previous approaches.

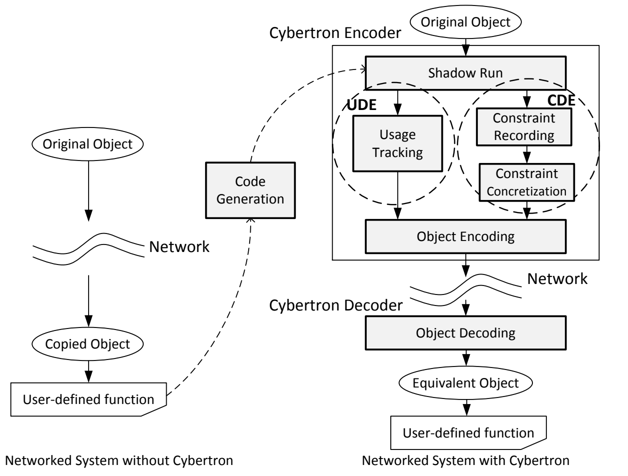


Figure 3. Cybertron Overview.

3. Execution-Equivalent Encoding

At the core of Cybertron is the notion of *execution-equivalent encoding*, where Cybertron encodes the input data to a user-defined function in such a way that the encoded data is *equivalent* to the original input with respect to the user-defined function, thereby preserving correctness, and yet is smaller in size than the original one. The left part of the Figure 3 shows a typical setting, where a stream of objects is transferred through the network; for example, for feeding from a storage system to mappers or for transmitting key-value pairs from mappers to reducers. Such data transfer is often necessary due to the inherent computation paradigm (e.g., between mappers and reducers) or due to the isolation between storage and computation resources in certain settings [13]. Cybertron can be applied on both mappers and reducers given the subsequent user-defined function that consumes the data.

3.1 Overview

Figure 3 depicts the high-level workflow in Cybertron to realize execution-equivalent encoding. The key to Cybertron is to figure out how a given user-defined function uses the input data. This is done through a *shadow program* that Cybertron derives from the given user-defined function in the *code generation* step. The shadow program provides a conservative estimate on what in the input data matters to the given user-defined function. The encoding step of Cybertron executes the shadow program in a *shadow run*, which does *usage tracking* and *constraint recording*. Usage tracking identifies the unused portions of the input data and is basis for *unused-data elimination (UDE)*. Constraint recording tracks the constraints that the input data must satisfy to preserve the correctness of the user-defined function. Cybertron further applies *constraint concretization* to generate a synthesized value that satisfies the constraints, but can be more efficiently encoded. This encoding process is *constraint based data encoding (CDE)*. Finally, Cybertron applies *object encoding*

```
1  // Input object type
2  struct Anchor {
3    string targetUrl;
4    string anchorInfo;
5    int    extraScore;
6  }
7
8  // Consumer program
9  class AnchorReducer {
10   void Reduce(string key, Anchor obj) {
11     string[] tokens = obj.anchorInfo.Split(';');
12     string anchorType = tokens[0];
13     string scoreStr = tokens[1];
14     string url = tokens[2];
15     string anchorText = tokens[tokens.Length - 1];
16     string domain = GetDomain(url);
17     int score = Int32.Parse(scoreStr);
18     if (_dict.ContainsKey(domain)) {
19         _dict[domain] += score;
20     } else {
21         _dict[domain] = score;
22         _textSample[domain] = anchorText;
23     }
24     if (anchorType.Equals("external"))
25         _dict[domain] += obj.extraScore;
26   } // Emitting code is omitted
27   string GetDomain(string url) {
28     if (url.Contains("://"))
29         url = url.Substring(url.IndexOf("://") + 3);
30     if (url.StartsWith("www."))
31         url = url.Substring(4);
32     return url.Split('/')[0];
33   }
34   Dictionary<string, int> _dict = ...;
35   Dictionary<string, string> _textSample = ...;
36 }
```

Figure 4. A sample user-defined function.

as the last step before transferring the object and *object decoding* after receiving it.

Figure 4 shows a sample user-defined function with the input object definition extracted from a production MapReduce job. The user-defined function is a reducer that receives all the `Anchor`s linking to the same target URL to compute the sum of the anchor scores for those sharing the same domain name. An `Anchor` object has three fields: `targetUrl`, `anchorInfo`, and `extraScore`. The former two are of the string type, and the last one is an integer. For each object, `AnchorReducer` splits the `anchorInfo` field into tokens (line 11) and gets the `anchorText`, `domain` and `score` (line 15-17). An extra token `anchorType` is used to judge whether the `extraScore` field should be added to the score (line 24, 25).

Figure 5 shows the encoding and decoding process on an input entry that has 70 bytes. In the shadow run, usage tracking infers that fields `targetUrl` and `extraScore` are unused: `targetUrl` is always unused, while `extraScore` is *conditionally unused* because whether it is used or not depends on the value of `anchorType`. In addition, `anchorText` (line 15) is extracted from `anchorInfo` at a position known only at runtime, imposing challenges to static approaches such as smart cut. Constraint recording collects the necessary constraints on each field to guarantee execution-equivalence. Figure 5 shows the constraints discovered for field `anchorInfo`. The first constraint

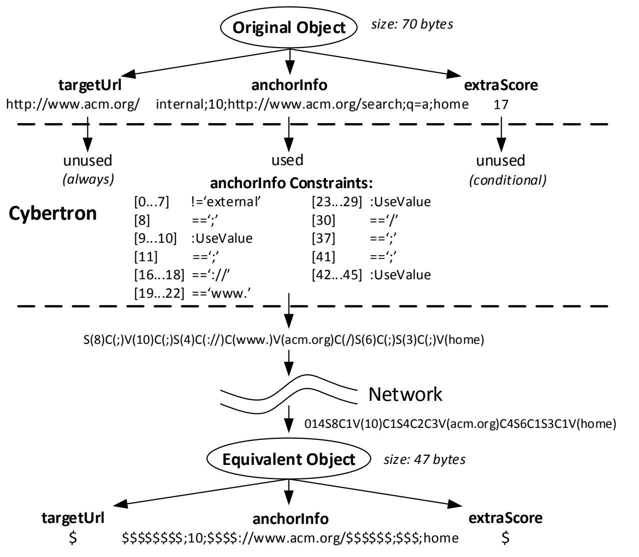


Figure 5. Cybertron on a sample object.

(`anchorInfo[0...7] != 'external'`) ensures that the condition for the if-statement in line 24 of the sample program is false, as in the shadow run. Any other value that satisfies the same constraints is going to yield the same result. Constraint concretization synthesizes a new value for the used field `anchorInfo` according to the collected constraints. Cybertron further encodes the new object with both the usage information from usage tracking and this new value into a compact format, which takes only 47 bytes in this case. The decoder creates a different input, but is equivalent to the given user-defined function because all the constraints on the used segments are preserved. The following subsections describe the technical details of the key steps: code generation, usage tracking, constraint recording, and concretization.

3.2 Code Generation

Code generation derives a shadow program from the given user-defined function and provides a wrapper for the input object to enable usage tracking and constraint recording when running the shadow program on the input data.

Wrapping input objects. Figure 6 shows the wrapper class for `Anchor` defined in Figure 4. Besides the original object (line 2), the wrapper contains additional fields for usage tracking and constraint recording. For instance, `fieldUsage` is a bitmap with each bit representing whether a correspondent field is used or not. We create a corresponding wrapper type `StrMeta` for each string-typed field. `StrMeta` has the same interface as `String` and implements constraint recording for each string method. For example, when a `Split(';')` method is invoked, `StrMeta` knows that certain positions in the host string must be character `';'`, while the other ranges contain any characters other than `';'`.

Deriving a shadow program. Cybertron derives a shadow program from a given user-defined function. For correctness, a shadow program must provide a conservative approximation

```
class AnchorWrapper {
  Anchor    originalObject;
  BitMap    fieldUsage;
  StrMeta   targetUrl;
  StrMeta   anchorInfo;
  // Construction
  AnchorWrapper(Anchor obj) {
    originalObject = obj;
    fieldUsage = BitMap.Empty();
    targetUrl  = new StrMeta(obj.targetUrl);
    anchorInfo = new StrMeta(obj.anchorInfo);
  }
}
```

Figure 6. Object wrapper for the example in Figure 4.

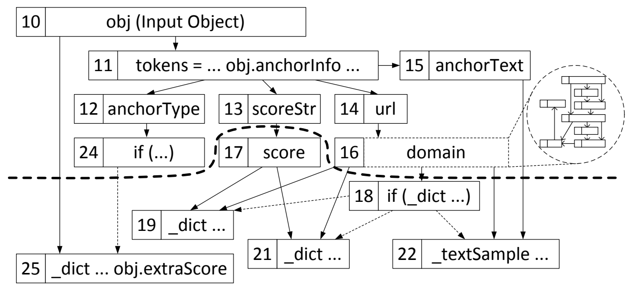


Figure 7. Dependency analysis and boundary cut.

on how input data is used in the user-defined function, such that (i) if a portion of the data is unused in the shadow program, it is unused in the original user-defined function; (ii) if a constraint matters to the original user-defined function, the constraint must be implied by the constraints derived in the shadow program.

There are cases where a pre-run cannot accurately follow the execution of a user-defined function; for example, when the execution uses environmental variables such as the time of the day. Another case is when the original user-defined function operates on a stream of input data entries and maintains states across the processing of individual data entries. Cybertron chooses to ensure that a shadow program does not include such states because accuracy in modeling such states depends on the assumption that the user-defined function processes the same stream of input data in the same order. Such an assumption does not always hold; for example, when the user-defined function is a reducer in a MapReduce program. We label those environmental variables and state variables *unsafe* variables.

Constructing a shadow program also involves trade-offs between effectiveness and overhead. In an extreme case, a shadow program can simply mark that all input data is important to the user-defined function, which is correct and incurs no overhead, but brings no network I/O saving. To maximize the opportunity for network I/O saving, the shadow program should ideally simulate the behavior of the original user-defined function closely in using the input data, as long as it does not depend on unsafe variables. Certain function calls in the original user-defined function might be excluded from the shadow program if they are expensive or hard to analyze because Cybertron's encoding involves executing the shadow program.

The process of deriving a shadow program starts with the construction of the dependency graph in a given user-defined function. Figure 7 illustrates the dependency graph for the sample in Figure 4. Each vertex in the graph is a statement with its line number in the leading square. The directed edges indicate data dependencies (solid lines) and control dependencies (dotted lines). For example, the edge from vertex 11 to vertex 12 indicates that statement 12 uses the data defined in statement 11 (`tokens[0]`). The edge from vertex 24 to vertex 25 indicates that statement 25 executes only when statement 24 evaluates true. Vertex 16 represents a function call and the dependency analysis drills down into the callee (as depicted in the dotted circle). Cybertron searches for an appropriate boundary in the dependency graph to derive a shadow program.

On the dependency graph, Cybertron marks *unsafe* statements that (recursively) depend on or define unsafe variables, i.e., those variables not purely decided by the current input object. In this case, `_dict` and `_textSample` are unsafe variables, and statements 18, 19, 21, 22 and 25 are therefore identified as unsafe. These statements cannot be selected as part of the shadow program. Starting from the input object, Cybertron traverses the graph by following data and control dependencies in the program. It marks a cut point whenever it encounters an unsafe statement. It can also choose to stop early not to include certain statements in the shadow program. For example, a string is converted into a primitive type through `Int32.Parse` on line 17. In this case, the full content of the string is required (`scoreStr`); there is no further opportunity for network I/O saving. Cybertron can therefore make this a cut point and exclude from the shadow program this statement and all the subsequent ones.

The set of cut points constitutes a *boundary cut*. The bold dotted line in Figure 7 shows one boundary cut. With the boundary cut, Cybertron selects the statements on the same side of the boundary cut as the input object vertex (vertex 10) on the dependency graph. In this case, vertices 17, 18, 19, 21, 22 and 25 are excluded.

Cybertron generates the shadow program according to this cut. Figure 8 shows the result. The differences from the original user-defined function in Figure 4 are underlined. The following changes are worth noting. First, the shadow program uses a wrapper class `AnchorWrapper`, rather than the original input object class, and replaces all `String` types to the corresponding `StrMeta` wrapper class. Second, for the *boundary* `String` variables at the boundary cut, Cybertron conservatively adds `UseValue` for all these variables to ensure their values are faithfully transmitted (e.g., for `domain` used in vertices 18, 19, 21 and 22, `anchorText` used in vertex 22, and `scoreStr` used in vertex 17), and sets usage bit for the others that directly references the field of the input object (e.g., `extraScore` used in vertex 25).

```
1  class ShadowAnchorReducer {
2    AnchorWrapper Reduce(Anchor obj) {
3      AnchorWrapper wp = new AnchorWrapper(obj);
4      wp.fieldUsage.Set(Field_anchorInfo);
5      StrMeta[] tokens = wp.anchorInfo.Split(';');
6      StrMeta anchorType = tokens[0];
7      StrMeta scoreStr = tokens[1];
8      StrMeta url = tokens[2];
9      StrMeta anchorText = tokens[tokens.Length - 1];
10     StrMeta domain = GetDomain(url);
11     scoreStr.UseValue();
12     domain.UseValue();
13     anchorText.UseValue();
14     if (anchorType.Equals("external"))
15         wp.fieldUsage.Set(Field_extraScore);
16     return wp;
17   }
18   StrMeta GetDomain(StrMeta url) {
19     if (url.Contains("://"))
20         url = url.Substring(url.IndexOf("://") + 3);
21     if (url.StartsWith("www."))
22         url = url.Substring(4);
23     return url.Split('/')[0];
24   }
25 }
```

Figure 8. Shadow program for the example in Figure 4.

3.3 Usage Tracking

Usage tracking is performed by the input object wrapper to discover what is unused when running the shadow program. As shown in Figure 6, Cybertron uses a bitmap (e.g., `fieldUsage` in `AnchorWrapper`) in the wrapper for usage tracking. Each bit of the bitmap corresponds to a field of the input type and indicates whether or not that field is used. For instance, `anchorInfo` is used at the beginning of the `Process` method in `AnchorReducer` (line 11, Figure 4). Cybertron emits a statement to set the corresponding bit in `fieldUsage` (line 4, Figure 8). Because usage tracking is done at runtime, the bit corresponding to a field that is conditionally used will not be set if the execution does not touch the field. For example, line 15 in Figure 8 is not executed when line 25 in Figure 4 is not executed. For a field of a non-primitive data type, such as a tree or a dictionary, Cybertron chooses not to track the detailed usage information and to conservatively assume that the entire structure is used as long as any single element is accessed.

After the shadow run, Cybertron uses the resulting bitmap when encoding the data: it attaches the bitmap as the header of the serialized data, and simply skips all the unused fields. The corresponding decoder reads the bitmap first and knows which fields to skip.

3.4 Constraint Recording

Cybertron also records constraints on the input data during the shadow run. This is done using the wrappers on particular data types. Because strings are the type of 89% fields in the input data we examine and offer the most significant opportunities for size reductions, Cybertron uses the `String` type as the showcase and implements the `StrMeta` wrapper. For all other types, we use their concrete values during the shadow run.

Defining Constraints. Cybertron encodes constraints imposed by string operations as a list of *range constraints*, each denoted as (s,l,c), where the substring starting from index s with length l in the input string must satisfy data constraint c. Cybertron introduces the following three types of constraints:

- Any: the concrete value of the substring does not matter, which usually happens when the substring is unused or the later user-defined function cares only about its length.
- Pred(predicate): the substring must satisfy a certain *predicate*; for example, it must not contain a certain constant string.
- Value: the substring must be faithfully preserved, as its actual value matters to the user-defined function.

Table 2 illustrates the range constraints for some popular string operations. For example, a range constraint of (0, this.Length, Any) ensures that the length of host string remains unchanged, but the content does not matter. A more complicated case is `Split`, which divides the host string into substrings by the given separator v. Cybertron ensures that the separator is present at exactly the same positions as in the host string by introducing range constraints ($s_i - v$.Length, v.Length, Value) for $i = 2,\ldots,n$, where n is the number of resulting substrings and s_i are their start positions. Cybertron also introduces (s_i, $s_{i+1} - s_i - 1$, Pred(!value.Contains(v))) for $i = 1,\ldots,n$ where s_{n+1} is the host string length plus 1, guaranteeing that the separator does not appear elsewhere.

For operations returning substring(s) of the host string, such as `Substring` and `Split`, Cybertron tracks the data ranges with regard to the original input string at runtime. For example, `url` on line 19 in Figure 8 is derived from the results of the `Split` operation on line 5. So when `Contains` on line 19 returns true, (s, 3, Value) is added to `anchorInfo` where s is the correspondent start position of "`://`" in `anchorInfo`.

Range constraints imposed by a string operation may differ depending on the return value. `Equals` and `IndexOf` shown in Table 2 are two examples. For `IndexOf`, when the return value ≥ 0, the range constraints are similar to that with `Split`; otherwise, the range constraint makes sure that the given parameter string v does not appear in the host string.

3.5 Constraint Concretization

With the range constraints that the shadow run collects, a naive scheme is to encode the constraints directly in the encoding process and to use a constraint solver in the decoding stage to come up with an input that satisfies these constraints (and hence is equivalent to the original input with respect to the given user-defined function.) For the scheme to work effectively, Cybertron must find an efficient encoding of the constraints for size reduction, while at the same time coming up with a specific constraint solver that is efficient to be used in decoding. Two properties of the collected constraints make this task particularly challenging.

String operation	Return	Range constraint
int Length()	-	(0, this.Length, Any)
string[] Split(string v)	-	($s_i - v$.Length, v.Length, Value) for $i = 2, \ldots, n$
		(s_i, $s_{i+1} - s_i - 1$, Pred(!value.Contains(v))) for $i = 1, \ldots, n$
		where $s_i \ldots s_n$ are start positions of the resulting substrings,
		and s_{n+1} = this.Length+1.
string Substring(int *start*, int *len*)	-	-
string ToLower()	-	(0, this.Length, Value)
bool Equals(string v)	true	(0, this.Length, Value)
	false	(0, this.Length, Pred(value != v))
int IndexOf(string v)	r (≥ 0)	(r, v.Length, Value), (0, $r + v$.Length-1, Pred(!value.Contains(v)))
	r (< 0)	(0, this.Length, Pred(!value.Contains(v)))

Table 2. Range constraints for common `String` operations. Constraints for certain operations depend on the return value. "this" represents the host string for the current operation, and "value" represents the substring of the correspondent range in the host string specified by the current constraint.

First, the ranges from different range constraints may partially overlap. *Normalizing* the constraints in a reasonable way is non-trivial. For example, one might attempt to transform the existing set of constraints into one on non-overlapping ranges. This requires *safe decomposition* of constraints into those on smaller ranges, i.e., any strings satisfying the decomposed constraints must satisfy the old constraint as well. This is doable for Any and Value constraints, but difficult for Pred ones. For example, the constraint (0, 6, Pred(!value.Contains("xy"))) cannot be simply decomposed into two sub-constraints (0, 3, Pred(!value.Contains("xy"))) and (3, 3, Pred(!value.Contains("xy"))), because a string "abxycd" satisfying the two sub-constraints fails the original one, which violates the safety requirement of decomposition.

Second, there can be multiple range constraints discovered for the same range. Note that the types of the range constraints have a total order Any $\preceq$ Pred $\preceq$ Value; that is, a Value constraint is stronger than any Pred constraints discovered for the same range, which in turn can always satisfy any Any constraints. When constraints with different types are discovered for the same range, Cybertron can always select the strongest one as the only constraint for the range. The real difficulty lies in resolving two Pred constraints. For example, strings satisfying (4, 2, Pred(!value.Contains("x"))) may not satisfy (4, 2, Pred(!value.Contains("y"))).

In general, solving a set of Pred constraints is difficult. Cybertron gets around this problem by removing the need for any constraint solving during decoding. This is done by introducing a constraint concretization process, where Cybertron essentially solves the constraints with an alternative input during encoding. That alternative input satisfies all the constraints and can be more efficiently encoded than the original input.

The key to constraint concretization is to solve all the Pred constraints by assigning concrete values to each position in a string. To simplify, for any Pred constraint not of types `(value != v)` or `(!value.Contains(v))`, where `v` must be a constant string, Cybertron conservatively upgrades it to the Value type. This simplification is justified because our experiences show that those two types are the dominant Pred constraints and retain the most potentials for size reductions. Others (e.g., those for `Compare`) are rare. Cybertron then picks a special character σ that is not used in any parameters in the Pred constraints, and does the following for each position: if the position is covered by a Value constraint, Cybertron assigns the character from the original input string at this position. Otherwise, it must be covered by only Any or Pred constraints, and Cybertron assigns the special character σ to this position.

Constraint concretization produces an assignment that satisfies all the constraints: for any Value constraint on a certain range, the whole range must have been filled with the original characters, thereby satisfying the constraint. For any Pred constraint, which must be of the types `(value != v)` and `(!value.Contains(v))`, each position in the range is filled with either the one in the original input data if that range is also covered by a Value constraint, or σ otherwise. Because the original input data satisfy the constraint and σ is not in `v`, the newly constructed string must satisfy the constraint. Any Any constraint is trivially satisfied.

Putting it all together. Constraint concretization solves all the constraints and creates strings with either the original characters or the special character that satisfy the constraints needed to preserve the correctness of the given user-defined function. To help with efficient encoding, Cybertron further identifies the constant strings that appear as the parameters to string operations in the user-defined function. Because those constant strings recur frequently in the input data, Cybertron introduces a constant table and each occurrence of a constant can be represented simply by its index to the table. In the end, Cybertron generates three types of constraints: Special, Value, and Const on non-overlapping ranges, where Special fills the range with the special character σ and Const is a special case of Value that matches a constant in the user-defined function. Cybertron then efficiently encodes those constraints in the encoding process.

```
1  // original program
2  if (obj.field0)
3      _str  = obj.field1;               // safe
4  idx = _str.Indexof("something");      // unsafe
5
6  // after transformation
7  if (obj.field0) {
8      _str  = obj.field1;               // safe
9      idx  = _str.Indexof("something"); // safe
10 } else {
11     idx  = _str.Indexof("something"); // unsafe
12 }
```

Figure 9. Enlarged shadow program by code transformation (highlighted with underline).

4. Implementation

Cybertron is implemented as two major components: a runtime library and a code generator, which take 1,100 and 6,900 lines of C# code, respectively. The runtime library provides the implementation of `BitMap` and `StrMeta` used by the code generator. The runtime also provides *Encode* to encode the results from usage tracking and constraint concretization, and the corresponding *Decode*. The encoding is straight-forward: a header describes which fields are used, i.e., the `BitMap`, followed by the serialized buffer for each field if the field is used. The header is omitted when static analysis in the code generator finds only statically unused fields. For each field with constraints, the field is encoded as a list of non-overlapped range constraints (from constraint concretization), each with its type and type-specific payload: length for Special constraints, length and concrete values for Value constraints, and constant id (referring to a global constant table from code generation) for Const constraints.

The code generator is built on top of ILSpy [15], an open-source .NET decompiler, to handle both .NET assembly and C# code. Given the input consumer program, the code generator transforms it into ILAst, which is the internal Intermediate Representation (IR) in ILSpy, analyzes the ILAst, and generates the target code. We report here the worthy bits from our experiences.

Merge adjacent range constraints. Cybertron merges adjacent range constraints when possible to make the encoding more efficient. Given continuous Value ranges, Cybertron merges them together to be one larger range. When the range of a Const constraint is only one byte (a common case because the separator of `Split` is usually a single character) and it is next to a Value constraint, Cybertron merges it into the neighboring Value constraint.

Make unsafe statements partially safe. Shown in Figure 9, `_str` is a global variable which records `field1` of the latest object whose `field0` is `true`. Therefore it is an unsafe variable, and the statement on line 4 is excluded from the shadow run because it uses the unsafe variable. Cybertron enlarges the shadow run scope by duplicating this statement into both the two preceding branches, as shown on lines 9 and 11, which makes statement 9 safe. The shadow program can now include statement 9 to uncover more size reduction opportunities (e.g., when `obj.field0` is `true`).

Track constraints after `ToLower` and `ToUpper`. We observe that some string operations, such as `ToLower` and `ToUpper`, are often used to sanitize input data for later computation, especially for comparison against constants. Because they may change every character in the host string, they impose a Value constraint over the whole range, which terminates the traversal for generating the shadow program as discussed in Section 3.2. If all computation on a string is done after such a string operation, we can safely apply the operation on the input data first. We can use the transformed data as the input data. This has no impact on the correctness of the computation because applying the same operation again produces the same result. This allows Cybertron to track the constraints further to uncover more size reduction opportunities.

5. Evaluation

This section reports the effectiveness of Cybertron against 10 user-defined functions in different MapReduce jobs and their input data extracted from the production environment to answer the following questions: (i) How much network I/O reduction can Cybertron achieve? (ii) What are the key factors that affect the effectiveness? (iii) How does Cybertron compare with the previous static approaches and traditional compression algorithms? (iv) How does Cybertron affect the end-to-end performance under various network conditions?

5.1 Benchmarks

We collect 5 mappers and 5 reducers in different MapReduce jobs developed and used by various production teams, including the search team, the advertisement team. and data mining teams. These jobs are written in SCOPE [32], a SQL-like declarative language with user-defined functions written in C#, similar to Pig [22] and Hive [25]. We briefly describe these jobs and their data.

RevenueByPosition (Case 1, Reducer) This job computes the total advertisement revenue in each page position, which is a predefined three-character string. Besides summing up the revenue, the reducer also aggregates several other properties about advertisement events in a customized way.

PageGem (Case 2, Reducer) This job mines commercial values from web pages containing videos of electronic products. The `PageGem` reducer in this job aggregates a list of records by URLs and, for each URL, sorts products by their ranks.

InternalSysReport (Case 3, Reducer) The input is the log generated by an internal work flow. The log contains the starting and finishing times of various events in the system. The reducer in this case aggregates the log by the work-flow name and computes the duration of each event.

HostRule (Case 4, Reducer) This job computes the statistics for each URL from the crawler's log. In each log entry, the reducer uses the first several integers to compute the result, while ignoring any subsequent long sequence of IP addresses or English words (depending on log entry types).

FollowAnalysis (Case 5, Reducer) This job processes the search-engine log to find out the clicks to a given domain *A* after a user clicks a link to domain *B*. Each record in the log contains several fields to identify a user, a URL, a query string and a timestamp. Although URLs in the log are usually very long, pointing to specific web pages, this program cares only about the domain name and the depth (number of slashes) in each of them.

FatigueMeasure (Case 6, Mapper) This job processes the log generated by a production on-line advertisement system (shown in Figure 1). It computes the fatigue level of each advertisement impression, which is the number of impressions with the same user identity and on the same site within a given time window. Then it counts the number of impressions and the number of clicks for each user, site, and fatigue level.

AnswerRelativeness (Case 7, Mapper) The input of this job is the cooked log from a question answering system. Each line contains 10 key-value pairs indicating how relevant the answer is to each domain (e.g., sports or music). This job finds out the sum of relevance in 8 out of 10 domains. Other metadata in the input is discarded.

AnswersArbitration (Case 8, Mapper) This job works for the same question answering system as in Case 7, but processes a different log that contains the detailed information of each answer arbitration event in a nested format. This job extracts only a specific attribute from those events and joins them with another data file.

DeDupQuery (Case 9, Mapper) The input log records local performance data of a machine in the search engine. The log contains machine names, queries, timestamps, process IDs, and other detailed information. This job extracts some feature words from each query and finds out queries with distinct features in every minute.

MentionGraph (Case 10, Mapper) This job extracts a *mention graph* from the Twitter data. Each line of the input consists of the user name, the timestamp, and the text of a tweet. The job creates an edge from node *A* to node *B* in the mention graph if there is a tweet by user *A* containing a word "@*B*".

5.2 Network I/O Reduction

Experimental setup. To ease the evaluation of each case, we manually download the source code and a sample of around 500MB to 1GB input data from the production cluster to a local machine. For a reducer case, we manually re-run the preceding mapper in the job to generate the reducer's input. We further apply Cybertron to generate the encoder and decoder functions, which takes less than 2 seconds for all cases. Finally, we run the encoders and decoders on the local machine to compare their performance numbers; certain instrumentation is done to collect these numbers. The evaluation successfully verifies the correctness

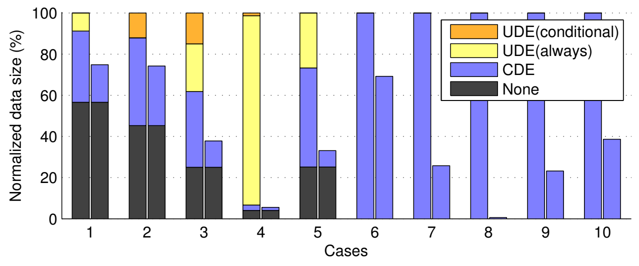


Figure 10. Overall normalized encoded sizes and the breakdown. The left bar has the breakdown before encoding, while the right bar after encoding.

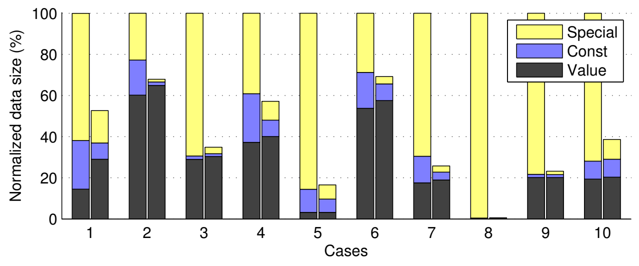


Figure 11. CDE breakdown. The left bar has the breakdown before encoding, while the right bar after encoding.

of Cybertron on these cases by running the original user-defined functions on both the original data and the data manipulated by Cybertron, and comparing their results. The local machine where the evaluation is done has 8 GB memory and 4 core 2.83 GHz Intel Xeon CPU.

Overall effectiveness. Figure 10 shows the data size reduction ratio, as well as the contributions from various techniques used in Cybertron. We further show a breakdown on the sources of size reduction, which could come from CDE, UDE(always), and UDE(conditional). "None" refers to the fields that Cybertron does not handle. For fields that can be saved by UDE(conditional), only those really unused are counted, the rest are either encoded by CDE further if they are strings, or not changed otherwise. The right bars show the same breakdowns based on the total bytes (normalized) after encoding for each technique. Note that UDE(always) and UDE(conditional) do not appear on the right bars because those portions of the data are thrown away during encoding. UDE(conditional) introduces an additional header, which should be included in Cases 1 to 5, but the total size of this header is less than 2% of the encoded size for all these cases and therefore omitted in the figure. For the mapper cases (Cases 6 to 10), only CDE works because they process raw logs with a single string in each line, while in the reducer cases (Cases 1 to 5) UDE can usually remove some unused fields.

As shown in the figure,Cybertron reduces I/O by at least 23.5% and by as much as 99.4%. In particular, Cybertron can reduce the sizes by more than 50% of the input data

Cases	1	2	3	4	5	6	7	8	9	10
`Conditionally used fields`			X							
`Input-dependent loops`						X	X			X
`Data size estimation`	X		X				X		X	

Table 3. Optimization opportunities uncovered by Cybertron while not possible with static approaches.

for most cases, among which UDE(always) and CDE are particularly effective. This is not surprising because many of these data are shared and analyzed by many big-data analysis programs, which may focus on different parts of the data and therefore leave opportunities for UDE and CDE. We examine the computations and the resulting encoding by Cybertron; manually writing a program that extracts the information with the same effect is difficult. Doing so would also make the code hard to understand and maintain. When changes are made, the manual process needs to be re-done. Effectiveness of UDE(conditional) relies largely on how frequently a certain condition is false. For example, it works well in Case 3 whose sample data is shown in Table 1. `Payload` is only used when `ReportKey` is "`StartTime`" or "`EndTime`", which is true in only 2% of the records. On the other hand, Case 1 does not have significant savings because the condition filters out only a small number of mal-formatted values.

CDE effectiveness. We also study the breakdown of CDE to understand the key factors affecting its effectiveness. Figure 11 shows the contributions from the three constraint types Special, Value, and Const. For each case, the left bar has the breakdown before encoding, while the right bar after encoding.

The largest benefit comes from the Special constraints, where the payload of the encoding contains the range length only. All cases benefit from this type of constraints. Among all ranges contributed by the Special constraints in the 10 cases, we find 84% of the total range size is contributed by the Pred constraints, while the other 16% by the Any constraints.

The Const constraints also contribute to the savings. Each Const can usually be encoded into a one-byte constant ID. The longer the constant, the higher the savings. For example, Case 1 has only one constant of size 3, causing the encoded size to be a third of the original input data size marked with Const. Cases 10 has only constants of size 1 (a single space to split words and "`@`" to identify mentions in tweets) and therefore shows no savings.

The encoded size of the Value constraints is always slightly larger than the original size, because the encoding format introduces additional overhead besides the raw value. This overhead is significant for cases like Case 1, in which all ranges of the Value constraints contain a single byte, thereby making the resulting size exactly twice of the input data size. The overhead becomes negligible when the range for Value constraints is large. For example, a distribution analysis of the range sizes for Value constraints in Case 9 reveals

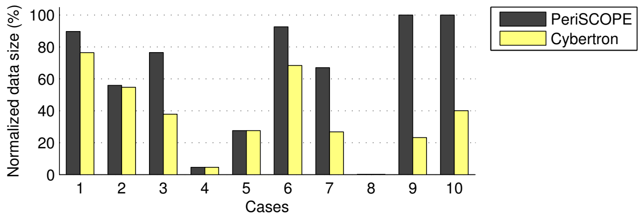


Figure 12. Comparison with a static approach. The left bar represents the resulting network I/O optimized by PeriSCOPE, while the right bar by Cybertron after PeriSCOPE; both are normalized to the original data size.

that the ranges usually contain a thousand of bytes as each range covers intermediate analysis results of a search query including a query string and many properties.

Comparison with static approaches. We also compare Cybertron with the previous static approaches to assess how much extra I/O reduction (relatively) the dynamic approach in Cybertron can achieve. PeriSCOPE [14] is a representative state-of-the-art static approach. PeriSCOPE not only uncovers the semantic of projection through *column reduction* and selection through *early filtering* in user-defined functions, but also incorporates *smart cut* to move certain processing logic in user-defined functions across different stages to reduce I/O.

We conduct the experiments as follows. For each case, we first run PeriSCOPE, which might rewrite the user-defined functions and redefine the schema of data transmitted on the network, and then apply Cybertron on the resulting user-defined functions. Figure 12 shows the resulting network I/O optimized by PeriSCOPE (on the left bars) and that further achieved by Cybertron after PeriSCOPE (on the right bars). Both bars are normalized to the original data size. In 6 out of 10 cases, Cybertron further reduces data size by a factor of 1.17x to 4.30x.

The improvement primarily comes from three opportunities uncovered by Cybertron's dynamic approach. The first two have been discussed in Section 2. Table 3 shows the cases that benefit from each opportunity. The first opportunity is to reveal conditional used fields as shown by the example in Table 1, which is extracted from Case 3. The second opportunity is in the input-dependent loops, whose iteration count is unknown statically. Analyzing such loops is beyond the capability of the static program analysis in PeriSCOPE, while Cybertron is able to track constraints imposed in each iteration dynamically. Cases 6 (the example in Figure 2), 7 and 10 benefit from this opportunity. Static approaches also suffer

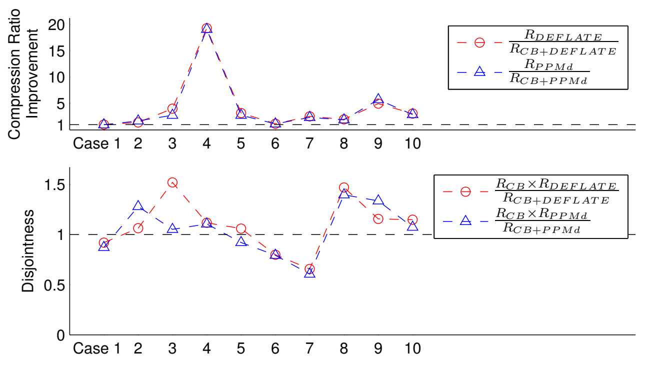


Figure 13. Compression algorithm improvement and disjointness with Cybertron. R_{CB}, R_{Algo} and $R_{CB+Algo}$ are the compression ratio (compressed size over input size) of Cybertron, *Algo*, and their combination (applying *Algo* compression after Cybertron compression), respectively. *Algo* is DEFLATE or PPMd.

from their inability to estimate data sizes accurately. This third opportunity contributes to the additional size reduction in Cases 1, 3, 7 and 9. For the example in Figure 4, the user-defined function extracts `domain` and `anchorText` from the `anchorInfo` field. When PeriSCOPE performs smart cut, it has an option to move the extraction logic across the network and transmit `domain` and `anchorText` (and something else) instead of `anchorInfo`. However, without any runtime information, it is impossible to figure out whether `domain` and `anchorText` have a smaller size than `anchorInfo`. As a result, PeriSCOPE makes a conservative decision to transmit the original field faithfully. In contrast, Cybertron is able to find out which parts in the field in each input object really matters to the computation.

5.3 Interaction and Comparison with Traditional Compression Algorithms

Most data compression algorithms can also be used to reduce the volume of data on network. For example, Hadoop [2] supports compression to reduce the size of data written to and read from the distributed file system and also data transferred between mappers and reducers. The techniques in Cybertron are largely orthogonal to those in traditional compression schemes. In the following set of experiments, we compare Cybertron with traditional schemes and show the effect of combining Cybertron with traditional ones. We select two widely used and different compression algorithms, DEFLATE [12] and PPMd [8], with the implementation in 7-Zip [23]. DEFLATE compresses duplicate strings and applies Huffman encoding for further size reduction, while PPMd uses a completely different algorithm to predict bytes according to previous bytes in the uncompressed data.

Compression ratio. We first measure how Cybertron can further improve the compression ratio of a traditional compression scheme by computing

$$\frac{R_{Algo}}{R_{CB+Algo}},$$

where *Algo* is DEFLATE or PPMd and R_{Algo} and $R_{CB+Algo}$ are the compression ratio (compressed size over input size) of *Algo* and their combination (applying *Algo* compression after Cybertron compression), respectively. The upper half of Figure 13 reports the result. In most cases, Cybertron improves the compression ratio by a factor ranging from 1.4 to 19, while for Cases 1 and 6 there is only 15% improvement to 5% degradation. Such impact on compression ratio of these traditional algorithms depends on both Cybertron's own effectiveness shown in Figure 10 and the overlapped effect between DEFLATE/PPMd and Cybertron which is discussed below. The results also indicate that, even in cases where Cybertron cannot help DEFLATE/PPMd, Cybertron does not introduce any significant negative effect on DEFLATE/PPMd.

To understand how much overlapped effect that DEFLATE/PPMd and Cybertron have on compression, we further measure the *disjointness* between DEFLATE/PPMd and Cybertron, defined as follows:

$$\frac{R_{CB} \times R_{Algo}}{R_{CB+Algo}},$$

where *Algo* is DEFLATE or PPMd and R_{CB}, R_{Algo} and $R_{CB+Algo}$ are the compression ratio (compressed size over input size) of Cybertron, *Algo*, and their combination (applying *Algo* compression after Cybertron compression), respectively. When disjointness equals to 1, Cybertron and DEFLATE/PPMd are perfectly orthogonal. Disjointness below 1 indicates that Cybertron and DEFLATE/PPMd overlap with each other to some extent. Disjointness above 1 shows that one helps make the other even more effective. The lower half of Figure 13 shows the results, which reveals interesting insights about the interaction between DEFLATE/PPMd and Cybertron. We find that disjointness is above 1 in over half of the cases, an indication that DEFLATE/PPMd works even better on data compressed by Cybertron than on the original data. This is primarily because Cybertron removes hard-to-compress data in some cases. Taking Case 8 with disjointness 1.4 for an example, Cybertron manages to discover two Base64-encoded 128-bit unique IDs, which keep changing from one object to another, and are difficult to compress. Once removed, DEFLATE/PPMd becomes significantly more effective. There are also cases where the disjointness is below 1; for example, Case 7 has a disjointness of 0.6. In this case, Cybertron mainly removed event type strings and encoded domain names with constant IDs. These strings have about only a dozen different values and can be easily compressed by DEFLATE/PPMd even if not eliminated by Cybertron. In addition, as shown in the figure, the conclusion is not sensitive to whether the traditional algorithm is DEFLATE or PPMd.

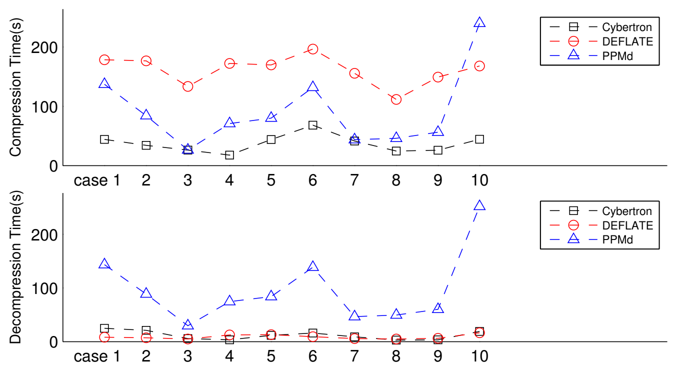


Figure 14. Compression and decompression time.

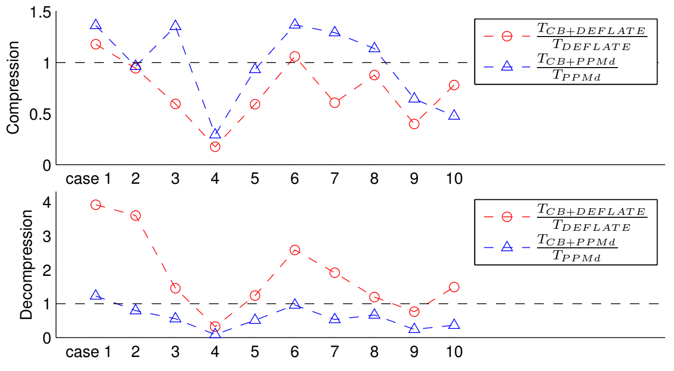


Figure 15. (De-)Compression time comparison. For each case, ratio of (de-)compression time of the combination of DEFLATE/PPMd and Cybertron to that of individual DEFLATE/PPMd is plotted.

Execution time. We also compare the execution times of these schemes and their combinations.

The upper half of Figure 14 depicts the compression time of different compression schemes. Cybertron runs $2.8x \sim 9.7x$ faster than DEFLATE and $1.0x \sim 5.4x$ faster than PPMd in compression, partially because DEFLATE/PPMd looks at a much larger data window across many objects while Cybertron focuses on one single object and with a static analysis stage providing additional knowledge (e.g., the constant table). Their corresponding decompression time is shown in the lower half of Figure 14. The difference of decompression speed of Cybertron and DEFLATE is negligible, while decompression of PPMd is much more complicated and runs $4.2x \sim 21.9x$ slower than Cybertron due to the its time-consuming prediction model.

Figure 15 shows the ratio of the (de-)compression time of the combined DEFLATE/PPMd and Cybertron scheme over the (de-)compression time of DEFLATE/PPMd. The upper part shows that in over a half of the cases, running combined Cybertron and DEFLATE/PPMd is faster than running just DEFLATE/PPMd (up to $5x$ in Case 4), which indicates that preprocessing data by Cybertron not only helps DEFLATE/PPMd gain higher compression ratio but also often save computation time. In other cases, we see an up to 18% additional execution time for DEFLATE and 37% for PPMd. Meanwhile, as shown in the lower half of Figure 15, the decompression of combination is faster than the time of the individual counterpart in 2 cases for DEFLATE and most cases for PPMd, because the compression ratio is higher due to Cybertron and there is less data to read during decompression.

5.4 Cost Benefit Analysis

Cybertron trades CPU cycles for reduction in network I/O. Specifically, it replaces serialization/deserialization with an encoding process before the network transfer and a decoding process after. Our measurement shows that the additional CPU overhead of encoding and decoding ranges from 4.0% to 63.6% and −59.1% to 14.9% compared to the execution time of the original user-defined functions, respectively. The decoding overhead is negative in many cases (1, 3, 4, 5, 8, and 9), which means the user-defined function becomes faster with Cybertron due to the reduced amount of data to be processed.

To understand when this CPU-network trade-off is profitable, we run the benchmarks pulling data over a LAN with large data sets in an exclusive cluster over which we have full control and simulate various network environment by throttling the inbound network bandwidth on each machine from 100 Mbps to 1 Gbps (no throttling). Note that in the real production cluster, network resources are shared by many concurrent jobs. The bandwidth available to a particular job varies significantly and could be much lower than the capacity of network adapters: we have often observed effective network bandwidth under 100 Mbps from our production traces. Figure 16 plots the trend of data processing throughput in each case when the network bandwidth varies. In most cases, the network I/O becomes the bottleneck when the bandwidth is low; Cybertron tends to improve the throughput significantly due to less data transmitted on the network. Even when the bandwidth is high, many cases benefit from Cybertron because the user-defined functions have less data to process and their speedup outweighs Cybertron's overhead. The curve in each case primarily depends on the complexity of the original user-defined function and the size reduction ratio achieved by Cybertron. For example, Cybertron works well in Cases 4 and 8 because the encoded data size is so small that network I/O is almost never the bottleneck with Cybertron enabled. While in Cases 1, Cybertron is less effective because the original user-defined function is time-consuming and both the gain of I/O reduction and the CPU overhead is negligible.

These experiments show that Cybertron could noticeably improve the end-to-end performance when the network bandwidth, rather than the CPU, is the bottleneck. Such cases have been observed in a shared production cluster deployed even

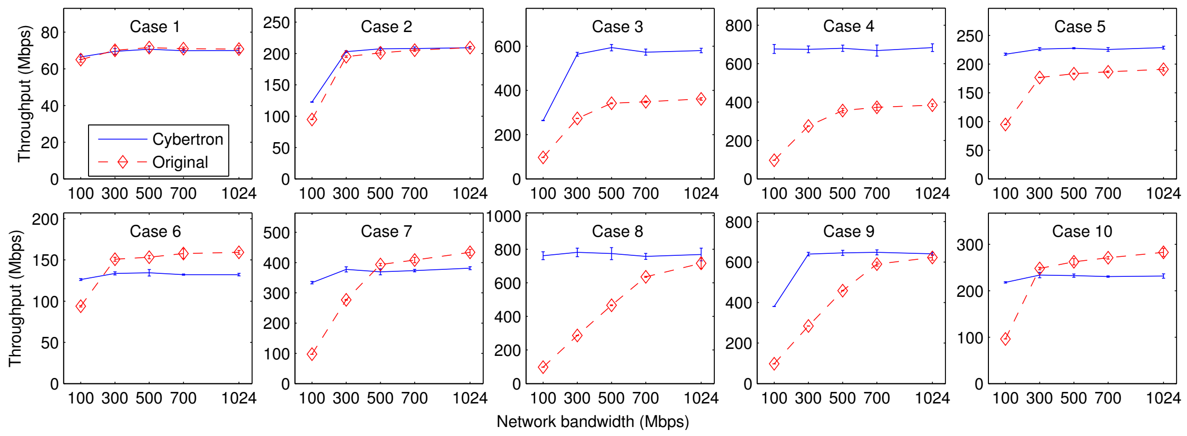


Figure 16. Data processing throughput under various network bandwidths. Each point is measured as a mean of five runs.

in a LAN setting. We have also seen the needs for data transfers across a wide area network in several cases, as echoed by others [13]; for example, when the data and computation have to be on different data centers due to security, privacy, policy, or resource constraints, or when data sources originated from multiple data centers are joined. To verify the results in the simulation, we also run some cases in a cluster in East Asia pulling data from U.S. over a WAN with a bandwidth of 213.7 ± 8.4 Mbps according to our measurements. This shows a real case where Cybertron shows a significant benefit and the measured throughput also validates the points in our controlled experiments.

6. Related Work

I/O reduction for MapReduce-style programs. MapReduce-style data parallel programs [2, 6, 11, 16, 22, 25, 27, 32] are widely used for big-data analysis; their executions tend to introduce a significant amount of I/O. Many techniques [13, 14, 17, 21, 28, 30, 31] have been developed to reduce network I/O for this computation paradigm. Most of the work involves analyzing the code of a program; for example, to enable partial aggregation [28], to eliminate unnecessary data shuffling [30], to facilitate database optimizations [17], and to apply compiler optimizations to an entire program [14]. Fundamentally, these approaches examine only the code (but not data) to make optimization decisions statically, while Cybertron leverages the computation and the data together for optimizations, and is able to resort to runtime mechanisms for more opportunities.

Computation independent data compression. In contrast to the code-analysis based approaches, most data compression algorithms such as DEFLATE [12] and PPM [8] can also be used to reduce the volume of data on network. For example, Hadoop [2] supports compression to reduce the size of data written to and read from the distributed file system and also data transferred between mappers and reducers. For those algorithms, decompression of the compressed data always returns the original data; in contrast, Cybertron ensures only execution equivalence with respect to a given computation. Specific data compression schemes have also been studied and proposed. Abadi et al. [1] discussed several compression schemes that can be integrated into columnar storage like C-Store [24] and Dremel [19]. Manimal [17] proposes to use *delta-compression*. BigTable [7] allows clients to specify the compression format applied to each file (`SSTable`) to save disk storage. These algorithms are largely orthogonal to Cybertron and can often be applied together with Cybertron for better overall network I/O reduction.

Execution equivalence and constraint solving. The idea of using constraint solving to come up with concrete instances that preserve certain equivalence has been used in other settings, such as securing software by blocking bad input [3, 9], correlating runtime logs for error diagnosis [29], and privacy-preserving bug reporting [5, 26]. When a particular input triggers a bug in a piece of software, it is ideal to send a bug report that can reproduce the bug, but without revealing user private data. Privacy-preserving bug reporting finds an input that can trigger the same bug, but with different values from the original input. Here, instead of execution equivalence, the alternative input must follow the same execution path to trigger the same bug. Symbolic execution [18] is generally used to generate test input [4, 20] that would take a particular execution path. This is achieved by collecting the conditions on the path and solving them with a constraint solver [10]. Cybertron instead uses constraint concretization to avoid the often expensive constraint solving by leveraging the fact that the input data provides a concrete instance that satisfies all the constraints.

7. Conclusion

Cybertron opens up a new avenue for optimizing data-parallel programs through a novel combination of static and dynamic mechanisms. It is the first system that establishes the feasibility and effectiveness of this approach through extensive evaluations with real programs on real data. The techniques are largely orthogonal to the existing static-analysis based approaches and have been shown to bring additional, sometimes significant, benefits. Even though the dynamic mechanisms in Cybertron incurs runtime overhead, Cybertron's design has carefully avoided expensive operations such as constraint solving to make such overhead manageable.

Acknowledgement

We thank the anonymous reviewers for their valuable and insightful comments. We also thank the production teams in Microsoft for sharing their jobs and data.

References

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[2] Apache. Hadoop. http://lucene.apache.org/hadoop/.

[3] D. Brumley, J. Newsome, D. X. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[5] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, pages 319–328, 2008.

[6] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI*, pages 363–375, 2010.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, pages 4:1–4:26, 2008.

[8] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, pages 117–130, 2007.

[10] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[12] P. Deutsch. DEFLATE compressed data format specification version 1.3. http://www.ietf.org/rfc/rfc1951.txt.

[13] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron. Rhea: automatic filtering for unstructured cloud storage. In *NSDI*, pages 343–356, 2013.

[14] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *OSDI*, pages 121–133, 2012.

[15] ILSpy. The open-source .NET assembly browser and decompiler. http://ilspy.net/.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, pages 59–72, 2007.

[17] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for MapReduce programs. *PVLDB*, 4(6):385–396, 2011.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[19] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, pages 330–339, 2010.

[20] Microsoft. PEX. http://research.microsoft.com/en-us/projects/pex/.

[21] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX ATC*, pages 267–273, 2008.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[23] I. Pavlov. 7-Zip. http://www.7-zip.org/, 2013.

[24] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: A warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[26] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, pages 131–144, 2007.

[27] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, pages 1–14, 2008.

[28] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP*, pages 247–260, 2009.

[29] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, pages 143–154, 2010.

[30] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, pages 295–308, 2012.

[31] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *ICDE*, pages 1060–1071, 2010.

[32] J. Zhou, N. Bruno, M. chuan Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. In *The VLDB Journal*, volume 21, pages 611–636, 2012.