Global Poverty Monitoring Technical Note 43

CACHE

A Generalized Mechanism for Storing and Replaying Any Stata Command

R.Andrés Castañeda and Damian Clarke

May, 2025

Keywords: Cache; Stata programming; computational efficiency

WORLD BANK GROUP
Development Data Group
Development Research Group
Poverty and Equity Global Department

Abstract

In this paper we describe the Stata program cache, which allows for the full output of any Stata command to be cached to disk, enabling easy recovery of command output in the future without the need for re-computation. The cache program interacts with any native Stata or user-written command, allowing for caching of any elements returned by Stata commands, including matrices, scalars, graphs, data and frames, as well as command output itself. This command is useful for improving programming practices and efficiency, particularly in cases where the underlying Stata commands are computationally intensive and slow to run.

All authors are with the World Bank. Clarke is also affiliated with the University of Chile and University of Exeter. Corresponding author: acastanedaa@worldbank.org. The authors would like to thank Christoph Lakner, Daniel Mahler, and Nishant Yonzan for comments and useful suggestions. The authors gratefully acknowledge financial support from the UK government through the Data and Evidence for Tackling Extreme Poverty (DEEP) Research Programme.

1 Introduction

Despite advances in Stata's analytical capacity, increasingly complex datasets and statistical algorithms often require significant computation time. Thus, tools that enhance computational efficiency can significantly reduce processing time and resource consumption. In this article, we introduce one such general programming tool which can greatly improve the efficiency of programming in Stata: `cache`.

`cache` is a prefix command which allows for the output of any Stata command to be cached, allowing them to be retrieved later without re-executing the underlying statistical computations. `cache` can be used as a precursor to *any* native or user-written Stata program, and automatically saves all output including elements in return lists, graphs, console output, and data and frames if such elements are altered by the command of interest. If `cache` finds that a requested command has been previously run, rather than re-running the original command, it reloads all required elements into return lists, graphical output, data and frames, and re-echoes the original command output.

The `cache` utilities work by examining both the exact command typed by the user, as well as the data signature of all data in memory. Thus, if either the command or the data used as an input to a command (or both) has not been previously executed, cache runs and caches the command. Otherwise, if the command-data combination has been previously cached, results are reloaded nearly costlessly in terms of computation.

`cache` is programmed to work as efficiently as possible, saving only the minimum elements required on the user's system such that command output can be loaded in the future without re-running the command. At times, specifically if a command alters data and requires saving of the altered data to the user's system, the footprint of cache may be somewhat large. For this reason, `cache` has a suite of options to manage this footprint, such as requesting for data (and, if relevant frames) *not* to be saved, as well as a sub-command to simply remove all cached information from local drives.

While not previously available as a general-usage command in Stata, the idea of caching command output is a widely-available tool in other languages and computational architectures. For example, similar functionalities exist as `DiskCache` (among others) in Python (Jenks 2023), `R.cache` and `memoise` in R (Bengtsson 2023; Wickham et al. 2021) and `CachedCalls` in Julia, and operating systems generally cache the output of processes for reloading in the future even without user input (see e.g. Bottomley 2004). The development of `cache` brings this functionality to Stata, integrating seamlessly with its command structure while preserving Stata's native environment.

While the primary advantage of these procedures lies in reducing computational time and resource consumption for time- or resource-intensive Stata commands, `cache` also benefits users operating in net-aware Stata. When used with commands requiring an internet connection, such as `netuse`, `cache` allows subsequent executions to proceed even without connectivity, provided the command was successfully run once with an active connection. This highlights an additional advantage of `cache` beyond computational efficiency.

The `cache` command is potentially of relevance to all Stata users who wish to increase the efficiency of their programming processes, or incorporate relatively simple elements into their code to decrease the computational burden of their commands or programs. In this sense, `cache` is a complement to other user-written general purpose programming commands such as the ftools and gtools suite of packages (Correia 2016; Caceres Bravo 2018), the `require` package (Correia and Seay 2024), tools to facilitate Stata's interaction with github (Haghish 2020) and processing (Vega Yon and Quistorff 2019), as well as the many efficiency boosting programming capacities of Stata and Mata described in (among other places) the Stata Programming manuals and Baum (2016).

In what remains of this article, we first provide a brief background on the principles of caching, as well as the mode of usage of `cache` in Section 2. We then describe `cache`'s Syntax and stored elements in Section 3. We provide a number of illustrations of the usage of `cache` in Section 4, and provide closing discussion, including a suggestion of when users may (and may not) gain from the `cache` routines in Section 5.

2 A Brief Background

In what follows, Section 2.1 briefly outlines the concept of caching. Section 2.2 outlines the precise nature of caching implemented in the `cache` command. And Section 2.3 describes how `cache` tests whether commands are identical to previously issued commands.

2.1 Caching

Disk caching temporarily stores data for faster future access. The concept of caching has roots in computer science and is crucial for improving the efficiency of systems that require repeated access to the same data. Initially, caching relied on RAM for frequently accessed data, but as datasets grew, disk-based caching became essential for handling data too large to fit in memory. This technique enables computers to "remember" prior computations, reducing the need to repeat costly operations, such as complex calculations or data retrieval, thus improving overall performance.

Caching became widely used in the early days of computing to speed up the performance of both hardware and software systems. One of the earliest uses of caching was in the form of hardware memory caches, which were implemented to address the performance bottleneck between the CPU and main memory. For example, the concept of a cache memory was first introduced by Wilkes (1965) to speed up access to data in computers. By the 1970s, as personal computing grew in popularity, caching methods expanded to include disk-based storage. In the 1980s and 1990s, operating systems and database management systems began to incorporate disk caching techniques to manage growing data storage needs. Technologies like virtual memory (introduced in the 1960s) and file system caching (widely used in modern operating systems) allowed for efficient data retrieval from disk storage, effectively extending the idea of memory-based caching to much larger datasets (Denning 1968; Tanenbaum 2009).

Beyond operating systems, many programming languages offer built-in caching mechanisms to store command outputs and avoid redundant computations. In particular, when commands are computationally intensive, the output of commands within languages can be saved to dedicated areas of the hard-disk implying that they can then be re-loaded, rather than re-computed, in the future. Such procedures are available in the routines indicated in the introduction in languages such as R, Python and Julia, and are widely available across other languages and within programs such as web-browsers, streaming platforms and word processing software.

In the social sciences, researchers working with large datasets or complex computational models can benefit from caching to disk. This approach allows for more efficient processing by minimizing the need to reload data from external sources or recompute results that have already been calculated. For example, when conducting large-scale surveys or simulations, caching can speed up the analysis process by storing intermediate results on disk, enabling researchers to pick up where they left off without having to start from scratch each time. For instance, in Stata, caching within a `do` file ensures that when the file is rerun, previously computed results are retrieved instantly, avoiding redundant execution. By integrating caching into their regular workflow, social scientists can improve the scalability and speed of their data analyses, saving time and resources while maintaining accuracy in their work.

2.2 `cache`'s Disk-Based Caching

As illustrated in Figure 1, the primary function of `cache` is storing command output. However, it also includes sub-commands like `cache clean` and `cache list`, which manage stored results. The full syntax and functionality of these commands are discussed below. Thus, prior to entering caching procedures, the `cache` command checks whether a sub-command has been issued, and if so, simply completes the requested auxiliary procedure. Provided that no such sub-command has been issued, `cache` continues to the caching routine for the command.

This caching routine is described in the lower elements of Figure 1, beginning with 'Cached?'. Specifically, if `cache` finds that a command has not previously been run, it will (a) run the full command, (b) save the output of the command to the cache directory, and finally (c) terminate with the command output in the Stata console. However, if cache finds that a command *has* been previously run, it will simply (a) load all elements of the command which had previously been saved to disk, and (b) terminate with the command output in the Stata console. Thus, regardless of whether a command has been run or not, a user will be presented with the full command output as the end point of their call to `cache`, however potentially at a substantially lower computational cost if a cached version of command output already exists.

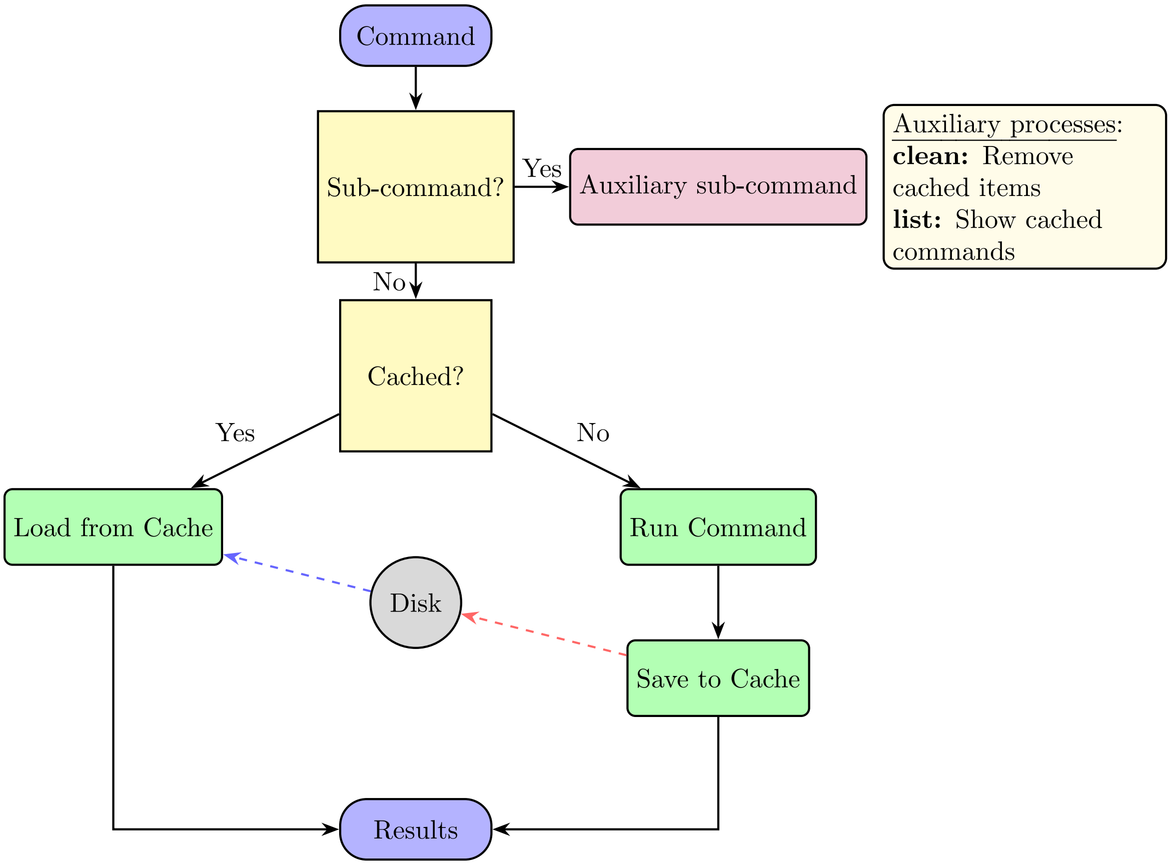

Figure 1: A Schematic Representation of the `cache` Command

2.3 Testing for Cached Commands

How `cache` determines if commands have been previously issued

In order for `cache` to work, it must determine whether an identical command has been previously cached or not. To determine whether this is the case, two specific conditions must be met. Firstly, the command input as typed must be identical. For example, if a user writes:

```
cache: reg y x1 x2, cluster(w)
```

and has previously written precisely the same command, this is potentially a match for command output to be loaded from the cache directory, rather than being recomputed. However, an identical command as typed does not necessarily imply that output will be identical, given that this also depends on the contents of data in memory. For example, if two distinct datasets are loaded, each with variables `y`, `x1`, `x2` and `w`, the above command would be valid in both datasets, but would result in distinct command output.

For this reason, `cache` considers two commands identical if they meet the following criteria:

1. The contents of the commands as typed is identical; and
2. The datasignature of data in memory is identical.

Each time `cache` is invoked, it generates a unique command signature. This signature is composed of two elements: (i) the SHA1 hash of the command, where redundant whitespace has been removed before hashing, created using Mata's `hash1` function, and (ii) the unique datasignature of data in memory, obtained via Stata's `datasignature` command. The generated signature is then compared against previously cached commands. If a match is found, `cache` reloads the stored output; otherwise, it executes the command, caches the output, and saves the new signature for future reference.

To see this in a simple way, consider if we load Stata's `auto` dataset, and wished to run a simple regression. If we take the hash of the syntax for this regression below, as well as the datasignature of the data in memory, we see that we are returned two elements:

```
. sysuse auto, clear
(1978 automobile data)

. mata: hash1("reg price weight length")
  2053929229

. datasignature
  74:12(71728):3831085005:1395876116
```

However, now imagine that we slightly altered the command, we wish to run, without altering the data in memory. In this case, the datasignature will remain identical, but the hash of the command will not, signifying that the input as typed of the previously run command and this new command is not identical:

```
. mata: hash1("reg price weight length, r")
  3489264427

. datasignature
  74:12(71728):3831085005:1395876116
```

And imagine now that we decide to make some change to our data, but revert to the original command. Here, while we will return to the identical hash as that previously generated, the change in the datasignature reflects the fact that *even though* the command as typed is identical, the command output will not necessarily be identical, because the data it is run on is distinct.

```
. replace price = 15000 if price>15000 & price!=15000
(1 real change made)

. mata: hash1("reg price weight length")
  2053929229

. datasignature
  74:12(71728):2359930702:1142297751
```

Finally note that if we now go back and re-load our `auto` data and consider an identical command as to that we had typed in step 1, we are now in a case where both the command input *and* the data are identical, implying that this command, and the command issued in step 1 above are identical.

```
. sysuse auto, clear
(1978 automobile data)

. mata: hash1("reg price weight length")
  2053929229

. datasignature
  74:12(71728):3831085005:1395876116
```

Two Implications

While these two elements allow us to check whether two commands are identical, and are used internally in `cache` for this reason, it does imply two particular considerations. Firstly, if commands are not typed identically, cache will not recognise them as identical, even if Stata users know they are identical. For example, consider the following two commands issued with the `auto` dataset:

```
regress price weight length
    reg price weight length
```

While Stata users will immediately see that these commands are identical given the usage of command auto-completion in Stata, `cache` will *not* view these commands as identical, given that they are typed differently. This is of course relatively innocuous, given that users can simply adopt a systematic style of referring to commands and using complete variable names consistently. What's more, at worst, `cache` will behave in an overly conservative way, simply caching redundant copies of these commands if a user requests for both versions of the command to be cached.

However, a second, and more important implication is that considerable care should be issued with commands which refer to external datasets, such as the following command:

```
merge 1:1 varlist using myfile.dta
```

Because `cache` tests the datasignature of data in memory as well as the hash of the command as typed this may cause `cache` to treat commands as identical even if they operate on different versions of `myfile.dta`. For this reason, any commands which rely on *external* data should be issued carefully, either by users who are sure that external files are indeed identical, or, ideally by using options which exist in `cache` to add datasignatures for other files. Examples of such procedures are documented in Section 4.4.

Hash Collisions

A final point to note is that while the use of `hash1` function allows commands to be stored and referenced in a more efficient way, it opens up the (relatively slim) likelihood of a hash collision. A hash collision refers to the (unlikely) event that two distinct strings are assigned precisely the same hash. Stata's `hash1` command generates 2^{32}, or around 4.3 billion distinct hashes. While it is very unlikely that any two specific strings will be assigned an identical hash, as the number of cached commands increases, the likelihood of such a clash increases. Indeed, the likelihood of a hash collision becomes substantial (over 50%), when approximately 2^{16} (or around 65,000) distinct elements have been hashed.[1] For this reason, `cache` includes a built in check to confirm if a matching hash does indeed refer to the same command, and, on the off chance that a matching hash is a hash collision, `cache` will issue an error message, indicating to the user that they should slightly modify their syntax, for example by writing `reg` instead of `regress`.

3 The cache command

The syntax of the `cache` command is as follows:

`cache` [*subcommand*, `options`][: `command`]

where the `cache` command can be optionally combined with a number of sub-commands,

---

[1] This is a problem referred to as the birthday paradox, as it is similar to the increasing likelihood that two individuals share a birthday as the number of people in a room increases. In the case of birthdays, the likelihood of a clash is greater than 50% with only 23 people in a room.

or alternatively issued as a prefix to any native or user-written Stata command. `cache` must either be employed as a stand-alone command and issued with a subcommand, or used as a prefix to a valid Stata command. Thus, in practice, one of the following two modes of usage should be employed:

`cache` *subcommand*[`, options`]

`cache` [`, options`]`:  command`

Valid options and sub-commands are discussed in turn below, followed by a description of objects returned by `cache`.

Options

`dir(`*string*`)` specifies the directory where cached contents of commands will be saved to be restored later. If not specified, a subdirectory of the current working directory named `dir(_cache)` is used by default.

`project(`*string*`)` allows for sub-folders within `dir` if further control of cached contents is desired. This permits cached commands to be stored in groups of related commands.

`prefix(`*string*`)` By default, all cached contents of a command will be saved with a prefix of `_ch` followed by the hash of the command as typed, along with the data signature of data in memory. The prefix option will replace `_ch` with the indicated string.

`nodata` If nodata is specified, cache will save all command returns, but will not save data if any changes in data are detected. This may be desired if data in memory is very large, and the command issued makes innocuous changes to the data that are not required in the future. Note that by default, any regression-based commands will save resulting data, given that a variable is generated to store information about the estimation sample via `e(sample)`. Thus, care should be taken if `nodata` is used with regression-based commands, as the use of `e(sample)` will not be available.

`datacheck(`*string*`)` By default `cache` tests data in memory and the command syntax, and determines that a command has been cached if data in memory is identical and the command as typed is identical. However, at times external data files may be called which have identical names, but altered contents. In cases such as this, the `datacheck` option can be used to indicate that `cache` should also ensure that any external data files necessary for the command are also included when generating a unique command identifier, or checking whether an identical command has previously been cached. As many data files can be indicated in `datacheck` as desired, and the name of each Stata data file should simply be separated by white space.

`framecheck(`*string*`)` Allows for identical behaviour as in `datacheck`, but now by testing the precise contents of any frames indicated in `framecheck`. As many frames can be indicated in `framecheck` as desired, and the name of each frame should simply be separated by white space.

`clear` Allows command implementation to proceed even if this would overwrite unsaved changes in data (similar, for example, to use, clear).

`hidden` By default `cache` returns all stored results, including hidden results as standard stored results, and so hidden results will be made visible following `cache`. If you would like hidden elements to stay hidden, the **`hidden`** option should be specified. Further details related to hidden stored results can be found in the Stata manual for the return command.

`replace` Forces `cache` to re-run the command and re-cache results, even if a previously cached version of command output has been found. Such an example may be useful if commands are re-issued and command behaviour has changed.

`keepall` Indicates that elements stored by previous commands in `e(return)` and `s(return)` lists should not be cleared prior to invoking the command requested with `cache`. By definition, `cache` will store all elements returned in **`ereturn`**, **`sreturn`** or **`return`** lists following an issued command. The default usage is to clear **`ereturn`**, **`sreturn`** and **`return`** lists prior to issuing the requested command, so that only elements from the issued command will be returned in future calls to the command from cache. However, this implies that elements of return lists not used by a specific command will not hold over from previous commands, for example if an r-class command is issued, results from previous e-class commands will be cleared to avoid being saved by `cache`. If such behaviour is undesired, the **`keepall`** option should be used so that any previous return lists are maintained in memory.

Subcommands

`clean` Indicates that all contents of cached commands should be permanently removed from the cache directory. Typing `cache clean` will result in a confirmation message to which the user must respond "y" if all contents of the cache directory should be cleared. This option should be used with care, as any contents of the cache directory will be permanently deleted. To avoid issues with potential loss of important information, the `clean` sub-command **does not** work recursively. For example, if various projects are located within a main cache subfolder, these must be cleaned individually by using the `dir()` and/or `project()` options.

`list` Lists the full history of cached commands available for re-loading in the current cache directory.

Global control of options

If users wish to temporarily or permanently alter the behaviour of `cache`, a series of global options can be defined which override `cache`'s default behaviour. Specifically, the following global options can be set, and these will be directly recognised by `cache` without requiring the use of command options.

`global cache_replace replace`	Automatically activates the replace option, overwriting the cache each time.
`global cache_on off`	Bypasses caching entirely (effectively ignoring the `cache:` prefix if present).
`global cache_prefix string`	Define a prefix for saving cached contents, overriding the default `_ch` used in the `prefix` option with any `_string` defined by the user.
`global cache_dir dir_name`	Define a default location for saving cached contents, overriding the default `_cache` directory with any `dir_name` defined by the user.
`global cache_project dir_name`	Define a default location within the cache directory to store cached output with any `dir_name` defined by the user.

While such global options are unlikely to be frequently used, this allows users to permanently set preferences for caching behaviour by defining such elements in their `profile.do` file, or setting such globals in Stata sessions, allowing for temporary overrides to default behaviour.

Returned Objects

`cache` is itself an `rclass` command, and along with all elements returned from the requested command, additionally stores the following in `r()`:

Scalars:

`r(cached_command)`	The precise command which has been cached.
`r(call_hash)`	The hash of the command as issued by the user.
`r(datasignature)`	The `datasignature` of the data prior to command being invoked.
`r(cmd_hash)`	The hash used by `cache` to uniquely identify the command.

Beyond `cache`-specific returns, cache loads or re-loads all elements into `e()`, `r()` and/or `s()` from the issued Stata command.

4 Illustrations of cache usage

In the sections below we provide a number of illustrations of the usage of, and performance of, the `cache` command.

4.1 A simple illustration of `cache`

As a first example, consider a regression using Stata's `auto` dataset as discussed in section 2. Provided this has not previously been cached, `cache` will run the command as normal, with elements saved into the `ereturn`, `return` and `sreturn` lists:

```
. cache: reg price weight length if foreign==0
```

```
 Note: Command is not cached. Implementing cache for future.

      Source |       SS           df       MS      Number of obs   =        52
-------------+----------------------------------   F(2, 49)        =     29.28
       Model |   266348916         2   133174458   Prob > F        =    0.0000
    Residual |   222845885        49   4547875.2   R-squared       =    0.5445
-------------+----------------------------------   Adj R-squared   =    0.5259
       Total |   489194801        51  9592054.92   Root MSE        =    2132.6

------------------------------------------------------------------------------
       price | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
      weight |    6.19526    1.10254     5.62   0.000     3.979623    8.410897
      length |  -120.5376   38.24525    -3.15   0.003    -197.3943   -43.68088
       _cons |   9163.626   4381.408     2.09   0.042     358.8579     17968.4
------------------------------------------------------------------------------

. return list

macros:
          r(call_hash) : "_ch2819172053"
      r(datasignature) : "74:12(71728):3831085005:1395876116"
           r(cmd_hash) : "_ch3830495101"

matrices:
              r(table) :  9 x 3

. ereturn list

scalars:
                  e(N) =  52
               e(df_m) =  2
               e(df_r) =  49
                  e(F) =  29.28278638568423
                 e(r2) =  .5444639140801126
               e(rmse) =  2132.574781614523
                e(mss) =  266348915.9325744
                e(rss) =  222845884.7597332
               e(r2_a) =  .5258706044507295
                 e(ll) =  -470.8242328036174
               e(ll_0) =  -491.2675217154787
               e(rank) =  3

macros:
            e(cmdline) : "regress price weight length if foreign==0"
              e(title) : "Linear regression"
          e(marginsok) : "XB default"
                e(vce) : "ols"
             e(depvar) : "price"
                e(cmd) : "regress"
         e(properties) : "b V"
            e(predict) : "regres_p"
              e(model) : "ols"
          e(estat_cmd) : "regress_estat"

matrices:
                  e(b) :  1 x 3
                  e(V) :  3 x 3
               e(beta) :  1 x 2
```

```
functions:
            e(sample)

. sreturn list

macros:
         s(width_col1) : "13"
              s(width) : "78"
```

Now imagine that some other commands are run (for example `sum price` below), such that elements in the return lists have changed, and you wish to re-gain access to these previous elements returned from the regression command. Rather than re-running the regression, we can simply call `cache` once again, and it will recover all previously returned elements rather than re-running the command:

```
. sum price

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
       price |         74    6165.257    2949.496       3291      15906

. return list

scalars:
                  r(N) =  74
              r(sum_w) =  74
               r(mean) =  6165.256756756757
                r(Var) =  8699525.974268788
                 r(sd) =  2949.495884768919
                r(min) =  3291
                r(max) =  15906
                r(sum) =  456229

. cache: reg price weight length if foreign==0
Command was cached.  Recovering previous output.

      Source |       SS           df       MS      Number of obs   =        52
-------------+----------------------------------   F(2, 49)        =     29.28
       Model |   266348916         2   133174458   Prob > F        =    0.0000
    Residual |   222845885        49   4547875.2   R-squared       =    0.5445
-------------+----------------------------------   Adj R-squared   =    0.5259
       Total |   489194801        51  9592054.92   Root MSE        =    2132.6

------------------------------------------------------------------------------
       price | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
      weight |    6.19526    1.10254     5.62   0.000     3.979623    8.410897
      length |  -120.5376   38.24525    -3.15   0.003    -197.3943   -43.68088
       _cons |   9163.626   4381.408     2.09   0.042     358.8579     17968.4
------------------------------------------------------------------------------
```

We can see above that all command output is also echoed to the terminal which is loaded from a log of the previously cached command. We can also inspect that return lists and see that along with specific elements which are included in the `return list` by cache, all other elements returned by Stata's regress command are incorporated into the `ereturn` and `sreturn` lists:

```
. return list

scalars:
              r(level) =  95
        r(PT_k_ctitles) =  1
       r(PT_has_cnotes) =  0
       r(PT_has_legend) =  0

macros:
          r(call_hash) : "_ch2819172053"
      r(datasignature) : "74:12(71728):3831085005:1395876116"
           r(cmd_hash) : "_ch3830495101"

matrices:
              r(table) :  9 x 3
                 r(PT) :  3 x 6

. ereturn list

scalars:
                  e(N) =  52
               e(df_m) =  2
               e(df_r) =  49
                  e(F) =  29.28278732299805
                 e(r2) =  .5444639325141907
               e(rmse) =  2132.57470703125
                e(mss) =  266348912
                e(rss) =  222845888
               e(r2_a) =  .5258706212043762
                 e(ll) =  -470.82421875
               e(ll_0) =  -491.2675170898438
               e(rank) =  3

macros:
          e(estat_cmd) : "regress_estat"
              e(model) : "ols"
            e(predict) : "regres_p"
         e(properties) : "b V"
                e(cmd) : "regress"
             e(depvar) : "price"
           e(_r_z__CL) : "t"
       e(_r_z_abs__CL) : "|t|"
                e(vce) : "ols"
          e(marginsok) : "XB default"
        e(marginsprop) : "minus"
              e(title) : "Linear regression"
            e(cmdline) : "regress price weight length if foreign==0"

matrices:
                  e(b) :  1 x 3
                  e(V) :  3 x 3
               e(beta) :  1 x 2

functions:
             e(sample)

. sreturn list

macros:
              s(width) : "78"
           s(width_col1) : "13"
```

Finally, all behaviour following `cache` can proceed as if the command was run fresh, rather than loaded from the cache. For example, below we inspect the `e(sample)` function returned by regression, and confirm that `cache` has also correctly returned the behaviour of this function.

```
. count if e(sample)==1
  52

. tab foreign if e(sample)==1

 Car origin |      Freq.     Percent        Cum.
------------+-----------------------------------
   Domestic |         52      100.00      100.00
------------+-----------------------------------
      Total |         52      100.00
```

4.2 An illustration of potential time-savings

As a second example, and to see the benefits of `cache` in practice, consider a command which may take considerable time to run, such as a bootstrap procedure. While the first time it is cached the command will indeed need to run, in future calls it will run essentially instantaneously. We begin below by running a regression with the bootstrap prefix with a substantial number of bootstrap replicates:

```
. sysuse auto, clear
(1978 automobile data)

. timer on 1

. cache: bootstrap, reps(10000) dots(500): reg price mpg
 Note: Command is not cached. Implementing cache for future.
(running regress on estimation sample)

Bootstrap replications (10,000): .........5,000.........10,000 done

Linear regression                                      Number of obs =        74
                                                       Replications  =    10,000
                                                       Wald chi2(1)  =     16.83
                                                       Prob > chi2   =    0.0000
                                                       R-squared     =    0.2196
                                                       Adj R-squared =    0.2087
                                                       Root MSE      = 2623.6529

------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
       price | coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
         mpg |  -238.8943   58.23514    -4.10   0.000    -353.0331   -124.7556
       _cons |   11253.06   1379.755     8.16   0.000     8548.791    13957.33
------------------------------------------------------------------------------
```

```
. timer off 1

. timer list
   1:     10.64 /        1 =      10.6410
```

We can see that these 10,000 replicates takes approximately 10 seconds to complete, given that the command has not been previously cached. However, now that the command has been cached, we can re-run it and access all elements from cache:

```
. timer on 2
. cache: bootstrap, reps(10000) dots(500): reg price mpg
Command was cached.  Recovering previous output.
(running regress on estimation sample)

Bootstrap replications (10,000): .........5,000.........10,000 done

Linear regression                                      Number of obs =         74
                                                       Replications  =     10,000
                                                       Wald chi2(1)  =      16.83
                                                       Prob > chi2   =     0.0000
                                                       R-squared     =     0.2196
                                                       Adj R-squared =     0.2087
                                                       Root MSE      = 2623.6529

------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
       price | coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
         mpg |  -238.8943   58.23514    -4.10   0.000    -353.0331   -124.7556
       _cons |   11253.06   1379.755     8.16   0.000     8548.791    13957.33
------------------------------------------------------------------------------

. timer off 2

. timer list
   1:     10.64 /        1 =      10.6410
   2:      0.01 /        1 =       0.0150
```

We see that now, instead of 10 seconds, the command takes around 0.01 seconds—the time required to load elements of the command which have been previously saved to the cache directory.

4.3 Saving and accessing other elements

The `cache` prefix also works for commands that issue multiple graphs. As an example, consider the following command which produces two graphs (which requires the installation of `sdid` from the SSC). First, we will run the command, and examine graphs in memory (graphs will also be produced in interactive versions of Stata):

```
webuse set www.damianclarke.net/stata/
webuse prop99_example.dta, clear

cache: sdid packspercapita state year treated, vce(placebo) seed(1213) graph g1on
Command is not cached.  Implementing and caching for future.
```

```
Placebo replications (50). This may take some time.
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
..................................................     50


Synthetic Difference-in-Differences Estimator

-----------------------------------------------------------------------------
packsperca~a |     ATT     Std. Err.     t      P>|t|    [95% Conf. Interval]
-------------+---------------------------------------------------------------
     treated | -15.60383    9.87941    -1.58    0.114   -34.96712     3.75946
-----------------------------------------------------------------------------
95% CIs and p-values are based on large-sample approximations.
Refer to Arkhangelsky et al., (2021) for theoretical derivations.

graph dir
    g1_1989  g2_1989
```

We can see here that along with command output, two graphs are produced in Stata's memory, which are called `g1_1989` and `g2_1989`. Now, we will drop graphs, and re-run with `cache` and confirm that the command is printed from `cache` and graphs have been re-loaded in memory (and re-displayed in interactive versions of Stata).

```
graph drop _all
cache: sdid packspercapita state year treated, vce(placebo) seed(1213) graph g1on
Command was cached.  Recovering previous output.
Placebo replications (50). This may take some time.
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
..................................................     50


Synthetic Difference-in-Differences Estimator

-----------------------------------------------------------------------------
packsperca~a |     ATT     Std. Err.     t      P>|t|    [95% Conf. Interval]
-------------+---------------------------------------------------------------
     treated | -15.60383    9.87941    -1.58    0.114   -34.96712     3.75946
-----------------------------------------------------------------------------
95% CIs and p-values are based on large-sample approximations.
Refer to Arkhangelsky et al., (2021) for theoretical derivations.

graph dir
    g1_1989  g2_1989
```

4.4 Caching with commands using external data files or frames

One case in which specific care needs to be paid when caching is when commands include the use of external data which is not in memory when the command is invoked. In cases such as this, `cache` has the `datacheck` (or, analogously `framecheck` in the case of frames). To see the importance of this, let's consider a very simple case of merging. Below we will generate two files with 10 observations and two distinct variables, `var1` and `var2`. The variables can be perfectly merged with the variable `ID`:

```
. clear
. set obs 10
```

```
Number of observations (_N) was 0, now 10.
. gen ID = _n
. gen var1 = rnormal()
. save myFile1.dta, replace
file myFile1.dta saved
.
. drop var1
. gen var2 = rnormal()
. save myFile2.dta, replace
file myFile2.dta saved
```

Let's now merge these files, and use the `cache` command to save the output for future. We see below that all works as expected, resulting in a file with each of these variables joined by `ID`.

```
. clear
. use myFile1
. cache: merge 1:1 ID using myFile2
 Note: Command is not cached. Implementing cache for future.
    Result                      Number of obs
    -----------------------------------------
    Not matched                             0
    Matched                                10  (_merge==3)
    -----------------------------------------

. des
Contains data from C:\Users\wb450043\Documents\cache\_cache/_ch3209895705.dta
 Observations:            10
    Variables:             4                  2 Feb 2025 23:10
--------------------------------------------------------------------------------
Variable      Storage   Display    Value
    name         type    format    label      Variable label
--------------------------------------------------------------------------------
ID              float   %9.0g
var1            float   %9.0g
var2            float   %9.0g
_merge          byte    %23.0g     _merge     Matching result from merge
--------------------------------------------------------------------------------
Sorted by: ID
```

However, imagine that we now open the using datafile of our merge and make some edit, re-saving the data *with the same name*. We do this below, simply adding another variable called `var3`.

```
. use myFile2
. gen var3 = rnormal()
. save myFile2.dta, replace
file myFile2.dta saved
```

Now, we are in a case where we can issue the same command with the same data in memory. Hence, unless instructed otherwise, `cache` will view this and the previous command as identical. Below we see that if we issue a naive implementation of cache, we will *not* generate the desired merged data with `var1`, `var2`, and `var3`, but rather, simply load the recent version of the command from the cache:

```
. use myFile1

. cache: merge 1:1 ID using myFile2
Command was cached.  Recovering previous output.
    Result                           Number of obs
    ─────────────────────────────────────────────
    Not matched                                  0
    Matched                                     10  (_merge==3)
    ─────────────────────────────────────────────

. ds
ID       var1     var2     _merge
```

As such, in cases where we are loading data from the disk and *also* wish to confirm that the data from the disk used in the command is identical, we simply need to instruct to `cache` that it should also check data on the disk, and add this to the fingerprint used to cache data. By adding the `datacheck` option `cache` correctly recognises that this is a new command, implements the command, and caches the command output for future reference:

```
. use myFile1

. cache, datacheck(myFile2): merge 1:1 ID using myFile2
 Note: Command is not cached. Implementing cache for future.
    Result                           Number of obs
    ─────────────────────────────────────────────
    Not matched                                  0
    Matched                                     10  (_merge==3)
    ─────────────────────────────────────────────

. ds
ID       var1     var2     var3     _merge
```

Such a case is potentially even more likely to occur if Stata's frames are used, as these may commonly be assigned similar names in everyday workflows. To see a case where `cache` may encounter problems if used naively (*i.e.*, without the `framecheck` option), consider a case where we have two frames in memory, `frame1` and `frame2`. Below, we will simply generate two frames which each contain 10 observations with random numbers, and append these together with the user-written `frameappend` command (Freese 2019).

```
. clear all

. frame create frame1

. frame create frame2

.
. set seed 121316

. cwf frame1

. set obs 10
Number of observations (_N) was 0, now 10.

. gen randnoise = rnormal()
```

```
.
. cwf frame2
. set obs 10
Number of observations (_N) was 0, now 10.
. gen randnoise = rnormal()
.
. cwf frame1
. cache: frameappend frame2
 Note: Command is not cached. Implementing cache for future.
. count
  20
```

We can see above that `cache` does as expected, caching the command output and implementing the command, given that this command has not previously been cached. But now let's imagine that we generate a different frame, and seek to once again append this to our original frame. If this frame is simply viewed as part of the command syntax, `cache` will re-load the cached command. However, if we correctly indicate to `cache` that this frame should be checked, we will see that all proceeds as desired.

```
. clear all
. frame create frame1
. frame create frame2
.
. set seed 121316
. cwf frame1
. set obs 10
Number of observations (_N) was 0, now 10.
. gen randnoise = rnormal()
.
. cwf frame2
. set obs 20
Number of observations (_N) was 0, now 20.
. gen randnoise = rnormal()
.
. cwf frame1
. preserve
. cache: frameappend frame2
Command was cached.  Recovering previous output.
. count
  20
. restore
.
.
. cache, framecheck(frame2): frameappend frame2
 Note: Command is not cached. Implementing cache for future.
. count
  30
```

Both `datacheck` and `framecheck` can work with arbitrarily many dataframes or datasets, provided that any files or frames are simply entered as a list with whatspace in between. The `cache` command also has a `force` option which indicates that even if a command has previously been cached, the command should be re-run and re-cached. While in practice situations such as this are likely to be relatively uncommon, it is important to note the importance of elements such as `datacheck`, `framecheck` and `force`, and employ these if ever necessary.

4.5 Use of cache sub-commands

Based on all of the above commands, we can examine sub-commands within cache. Below we use `cache list` which provides a list of all currently cached commands.

```
. cache list
Cached commands:
reg price weight length if foreign==0

bootstrap, reps(10000) dots(500): reg price mpg

sdid packspercapita state year treated, vce(placebo) seed(1213) graph g1on
```

The `cache` prefix stores commands on disk permanently, and so if we exit Stata and re-enter in a fresh session, provided that `cache` is pointed at the same project directory, `cache list` (and `cache` itself) will find all of the previously cached commands.

However, if we wish to clean up all cached commands and remove any saved elements from the disk, we can do so quite simply using the `cache clean` subcommand:

```
. cache clean
Warning: This will delete all files within ~/home/_cache
Do you want to continue? (y/n): . y

. cache list
Cached commands:
```

You will note that after the use of `cache clean`, the `cache list` sub-command now indicates we have no remaining commands cached.

Both `cache clean` and `cache list` assume that a user has worked with the default cache directory (`_cache`), however both of these commands can be used to provide greater control over `cache` with the `dir` and/or `project` options. If users wish to cache certain commands in groups this can be done by using directories in `dir` or `project`, and these can subsequently be cleaned up individually. For example, if commands are issued such as:

```
. cache, project(myproject): ...
```

just these commands located in the directory `myproject` can cleaned by issuing the command `cache clean, dir(myproject)`.

5 Conclusions

This article describes the `cache` command for Stata which provides a manner to save command output to disk, and replay the command without re-computation. This command can be very simply issued as a command prefix to any other Stata commands (including to commands which are themselves prefixes), allowing for considerable time-savings if commands are ever referred to in the future. As well as the time savings potentially available from caching and reloading commands, `cache` more generally increases the flexibility of net-aware Stata. By caching commands which only run when an internet connection is available (such as `webuse`), these commands can later be run even without an internet connection. This feature may be particularly useful to researchers working in the field, or in conditions with sporadic internet access.

In closing, it is worth briefly considering situations where `cache` may be productively used (and situations where *cache should perhaps not be used*). `cache` is useful for Stata commands with high computational complexity and long run-times, but which are likely to be re-computed multiple times. This will be well-suited, for example to computationally-heavy procedures in `do` files which are run repeatedly. However, there are also situations where `cache` may offer less benefits to users. For example, situations in which large data are used *and data are altered*, but commands themselves are relatively quick to run, are unlikely to be well-suited to caching. This is because (unless instructed otherwise) `cache` stores all altered elements in memory, including data files, and these datafiles may imply undesired costs in terms of memory use of a computer's hard disk (until such time as cached programs are cleared).

Overall however, `cache` offers a simple-to-use routine to increase the efficiency of Stata code, and seamless integration to all Stata commands. It may thus be viewed as a useful tool for programmers and any other analysts who wish to incorporate efficiency-boosting tools in their coding arsenal.

6 References

Baum, C. F. 2016. *An Introduction to Stata Programming*. 2nd ed. Stata Press. Paperback.

Bengtsson, H. 2023. *R.cache: Persistent Object Caching for R*. https://CRAN.R-project.org/package=R.cache.

Bottomley, J. 2004. Understanding Caching. *Linux Journal* . https://www.linuxjournal.com/article/7105.

Caceres Bravo, M. 2018. GTOOLS: Stata module to provide a fast implementation of common group commands. Statistical Software Components, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s458514.html.

Correia, S. 2016. FTOOLS: Stata module to provide alternatives to common Stata commands optimized for large datasets. Statistical Software Components, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s458213.html.

Correia, S., and M. P. Seay. 2024. require: Package dependencies for reproducible research. *The Stata Journal* 24(4): 599–613.

Denning, P. J. 1968. The Working Set Model for Program Behavior. *ACM Computing Surveys* 2(3): 323–333.

Freese, J. 2019. FRAMEAPPEND: Stata module to append frames. Statistical Software Components, Boston College Department of Economics. https://ideas.repec.org/c/boc/bocode/s458685.html.

Haghish, E. F. 2020. Developing, maintaining, and hosting Stata statistical software on GitHub. *The Stata Journal* 20(4): 931–951.

Jenks, G. 2023. DiskCache: Disk Backed Persistent Cache. http://www.grantjenks.com/docs/diskcache/.

Tanenbaum, A. S. 2009. *Modern Operating Systems*. 4th ed. Pearson.

Vega Yon, G. G., and B. Quistorff. 2019. parallel: A command for parallel computing. *The Stata Journal* 19(3): 667–684.

Wickham, H., J. Hester, W. Chang, K. Müller, D. Cook, and M. Edmondson. 2021. *memoise: 'Memoisation' of Functions*. R package version 2.0.1. https://cran.r-project.org/package=memoise.

Wilkes, M. V. 1965. Slave Memories for Control Units. *Proceedings of the IRE* 53(12): 1859–1863.

Acknowledgments

We are grateful to Christoph Lakner, Daniel Mahler and Nishant Yonzan for comments and useful suggestions. The authors gratefully acknowledge financial support from the UK government through the Data and Evidence for Tackling Extreme Poverty (DEEP) Research Programme.

The findings, interpretations, and conclusions expressed in this paper are entirely those of the authors. They do not necessarily represent the views of the International Bank for Reconstruction and Development/World Bank and its affiliated organizations, or those of the Executive Directors of the World Bank or the governments they represent.

About the authors

R. Andrés Castañeda is a Senior Economist in the Data Group of the Department of Development Economics at the World Bank.

Damian Clarke is an Associate Professor at The Department of Economics of The University of Exeter and the Universidad de Chile.