Core Model Proposal #405: Gcamstr: String Interning in GCAM

Product: Global Change Analysis Model (GCAM)

Institution: Joint Global Change Research Institute (JGCRI)

Authors: Pralit Patel

Reviewers: Kate Calvin, Marshall Wise

Date committed: 6/25/25

IR document number: PNNL-37939

Related sector: N/A

Type of development: Feature

Purpose:

Implement and utilize the String Interning design pattern in GCAM to reduce memory usage and increase performance.

Description of Changes

Background

The String interning design pattern (or more generically interning) attempts to keep a central dictionary of data that is used through out the application. Individual instances will just reference the data from that central dictionary rather than keeping copies in each. Given GCAM's use of strings as lookup identifiers and the huge amount of redundancy, the string interning design pattern is appealing.

In fact, an early attempt has been in the GCAM code base for decades using a custom implementation called Atom. However, the design pattern was never employed in more than a couple of places and slowly got left behind. In this proposal, we instead leverage a generic implementation in the Boost Flyweights library and employ in throughout the code base as "gcamstr".

The following table comes from an early exploration of possible memory savings, exploring the number of instances of std::string used in GCAM and the distribution of the values they contain. As we can see of the millions of std::string instances, only about fifty thousand were unique values. Although, given the small size of a character in a string: one byte, the memory savings from each reduced redundancy is not huge. On the other hand, given the size of a std::string (24 bytes) compared to a gcamstr (8 bytes) which is really just a pointer, the total memory savings is significant. In addition, we would anticipate some performance (reduced run time) benefits as well due to faster equality comparisons and hashing.

```
sizeof(string::value_type): 1
sizeof(string): 24
sizeof(gcamstr): 8
Num str: 45019036
Total size of str: 1436226243
Unique str: 52314
```

String Value	Number of instances
"Tg"	11397650
"GDP_control"	5171777
"CO2"	2331759
"Cropland Management"	782748
"N2O_AGR"	578314
"N fertilizer"	543142
"biomass"	540040
"biophysical water consumption"	518120
"BC_AGR"	516890

Approach

gcamstr

As mentioned above, we leverage the Boost Flyweights to fully provide the string interning implementation. In fact the gcamstr could be implemented as simply as:

```
using gcamstr = boost::flyweight<std::string, boost::flyweights::no_tracking,
boost::flyweights::no_locking>;
```

Although we do in fact create a fully defined class for gcamstr so that we can add some convenience methods for:

- Construction and assignment from C-style strings (and string literals)
- Assignments from std::string
- Fast < and == operators taking advantage of the fact that gcamstr are unique to speed up equality checks for std::map inserts and lookups.

Caveats

As can be seen above above, we have configured boost flyweight to have gcamstr be "lean and mean". In particular, we set "No Locking" and "No Tracking". The former implies that no safe guards are in place should we attempt to intern new strings from multiple execution threads. The later implies there is no attempt to limit "bloat" in the size of the unique string dictionary.

Given the above decisions, it behooves us to include some method to debug and diagnose any coding errors. Therefore we include additional book keeping and checks which are enabled when compiled with DEBUG_STATE, adding to other runtime safety checks. This will produce reports of interned strings per model period to ensure it is not growing through out the model run. In addition trap if any new strings are being added to the dictionary while GCAM is in a mode which might be doing parallel calculations which would be unsafe. Note, converting a std::string to gcamstr could be safe during this time, so long as that string had already been added to the dictionary previously. Ideally, we still would not do this as it could reduce performance. There are a handful of instances that are still doing this, however they are infrequent enough that it didn't justify refactoring code to avoid it.

Employing gcamstr throughout the code base

Generally speaking, gcamstr can be used as a drop in replacement for std::string. Where all the standard operators such as assignment, comparison, and input / output are all defined for seamless interoperability. If for whatever reason we need access to the actual std::string we can use the .get() method to access it. We have also added an `empty()` method for checking if the gcamstr is represented by the empty string for consistency with the std::string. When then changed checks of the form `mName == ""` to `mName.empty()` for consistency. Although the former check should still work through implicit conversion of "" to gcamstr.

Given the interoperability of std::string and gcamstr, we replace almost all occurrences of std::string with gcamstr. Including member variables, and function arguments that pass around region or sector names, for instance. Areas which were not converted to gcamstr include static

XML names and string literals as those already use their own form of string interning of sorts provided by the C++ compiler and there is no use in mixing these pools.

Refactor MarketLocator

As mentioned above, a co-benefit of switching to gcamstr are potential performance boosts in comparing or hashing strings. In particular the processes of translating a Market name and region to an internal location, which happens with such frequency that it is very performance critical. The current Core GCAM uses a two layer hash map: region name -> good name -> market index. Simply switching to gcamstr should provide a performance boost due to faster hashing. Although, we explored some alternative structures as well to see which performs the best and found that ultimately have a single layer of hash map (region name, market name) -> market index where the names are of type gcamstr performs the fastest and happens to simplify the code which is nice as well. The MarketLocator code has been refactored accordingly.

Map Type	String Type	Number of levels	Time (s)
objects::HashMap	std::string	2	104
std::map	gcamstr	1	233
std::unordered	gcamstr	2	36
std::unordered	gcamstr	1	30

Code cleanup

Atom / AtomRegistry

The old custom implementation of string interning is of course no longer needed, as we instead use an "off the shelf" implementation, so we remove it from the code base.

HashMap

The MarketLocator had been using a custom implementation of a hash map as it was implemented prior to std::unordered_map being part of the C++ standard. Now that it is and we have refactored MarketLocator, there is no need to keep the custom implementation any longer.

StringHash

Used in the above HashMap. The gcamstr can just be hashed by simply using the unique address of the interned std::string. That being said, we could also leverage the standard string hashing implementation which has since been included in the C++ standard.

InputFinder

Used to be used in old SGM code and has not been needed in a long time so we remove it.

AgSupplySector / Tech Remove "subsidy"

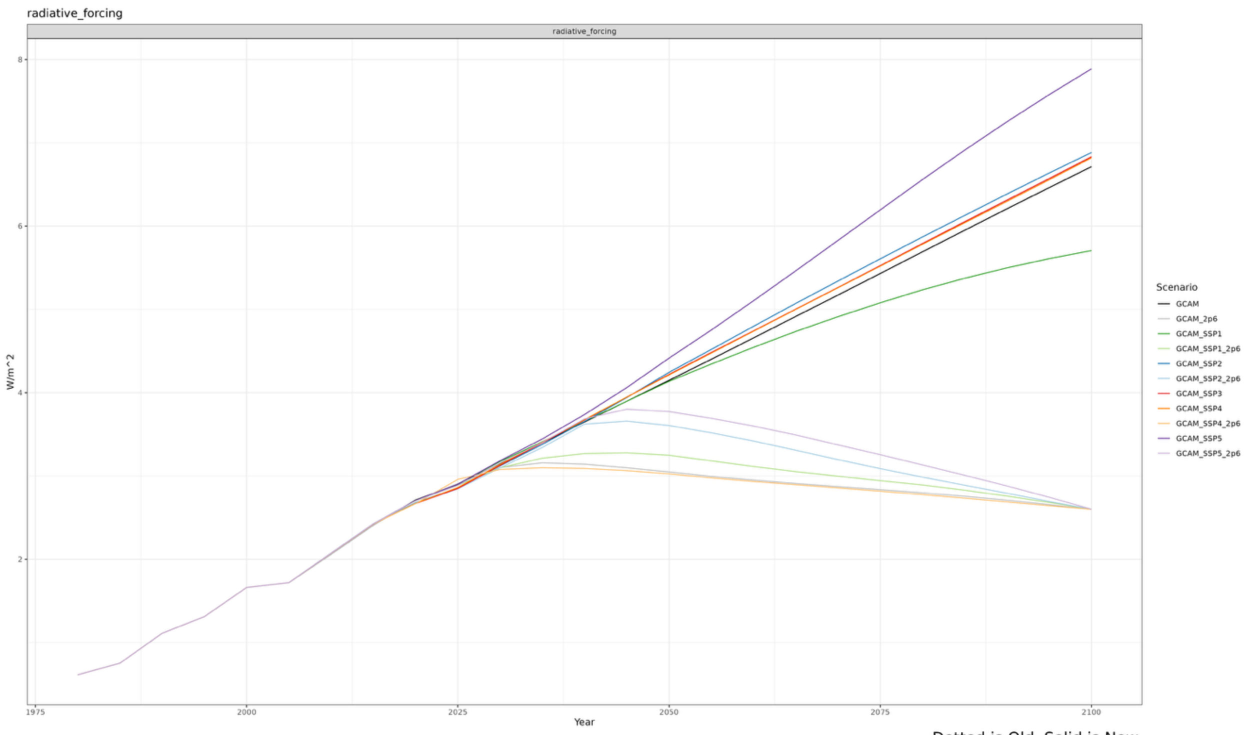
Remove the "subsidy" parameter which is has not been used for a very long time and can easily be replicated using more "standard" methods such as including an `input-subsidy" or negative "non-energy-cost".

Validation

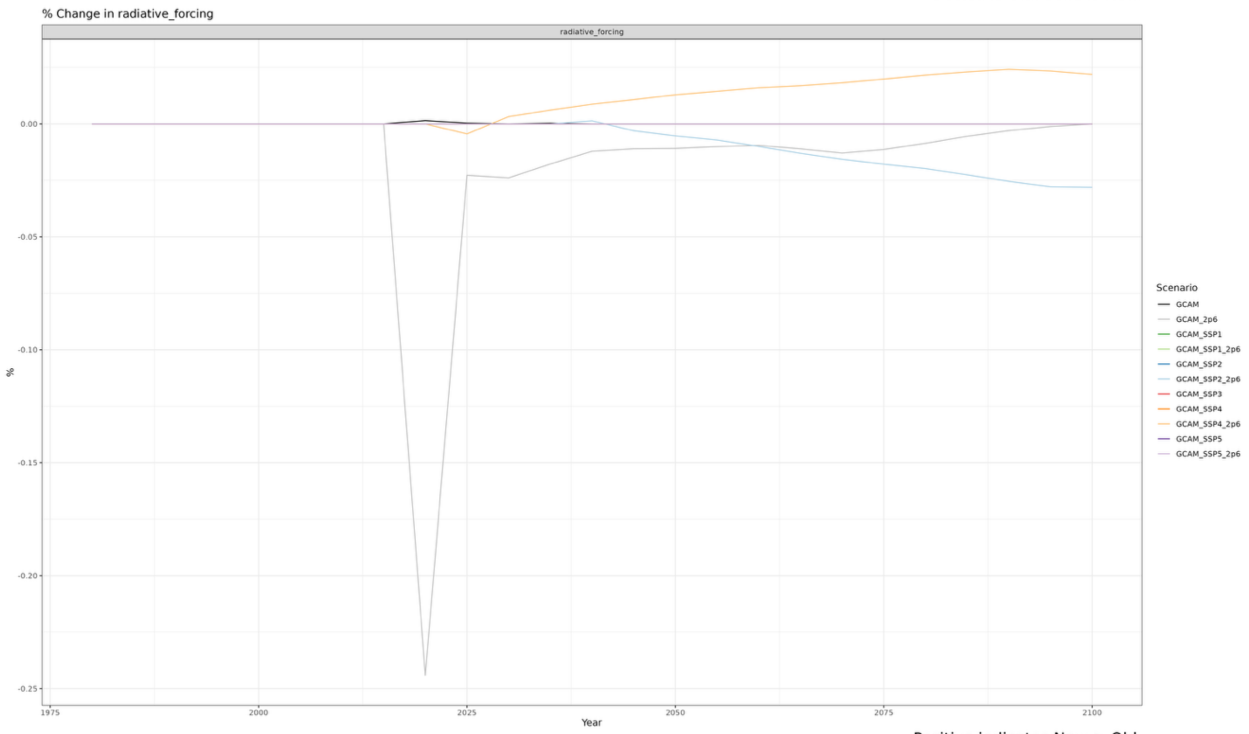
We begin by comparing memory usage and performance in the GCAM (32-region) and GCAM-USA configurations. Where we expect to see reductions in memory usage as well as modest reductions in model performance. Note, SSP and target finder scenarios have similar memory and performance savings to the GCAM scenario. A stated draw back of using the string interning design pattern is it will have slower performance while building up the central dictionary, which in GCAM would be encountered during XML Parse. However, we observe that the change in XML parse time is negligible (+/- 1%) and variability seems more related to OS level file caching.

Scenario	New Max Memory	Change in Memory	Change in Full Model Time / Iteration
GCAM	6.82 GB	-16%	-7%
GCAM-USA	9.43 GB	-16%	-7%

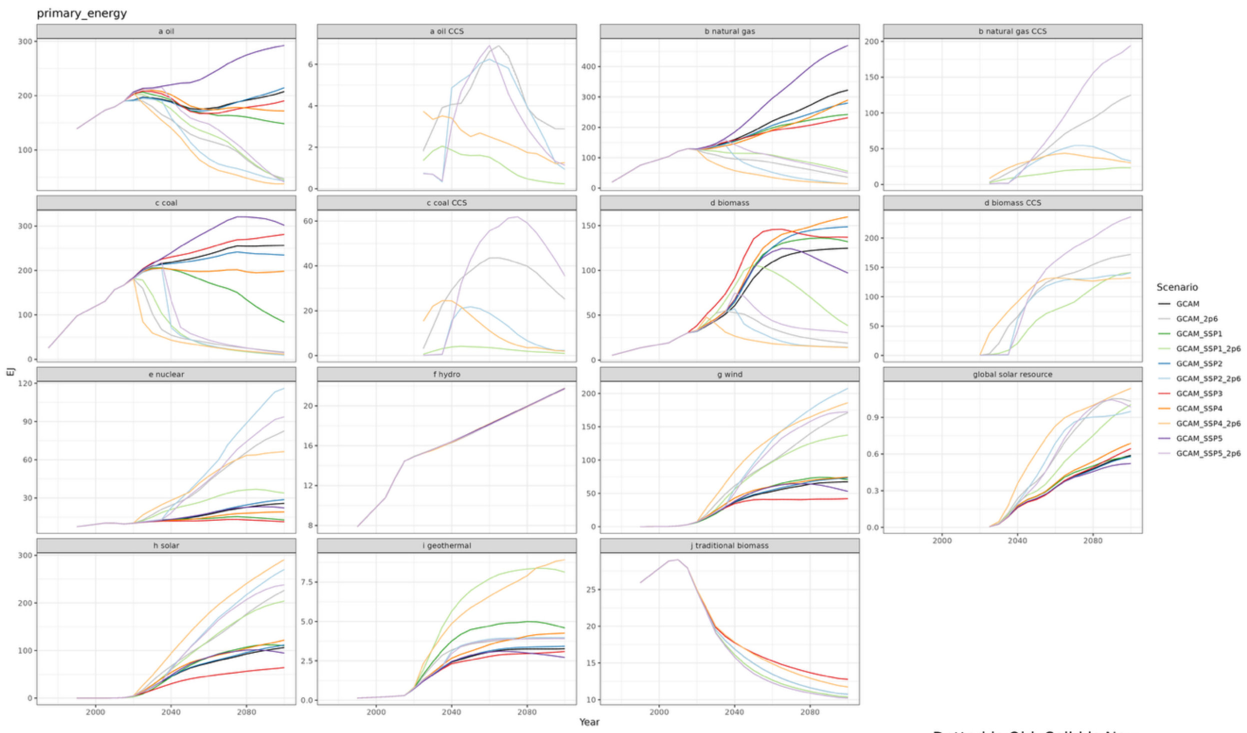
Finally, we do not expect this proposal to have any impacts on model results:



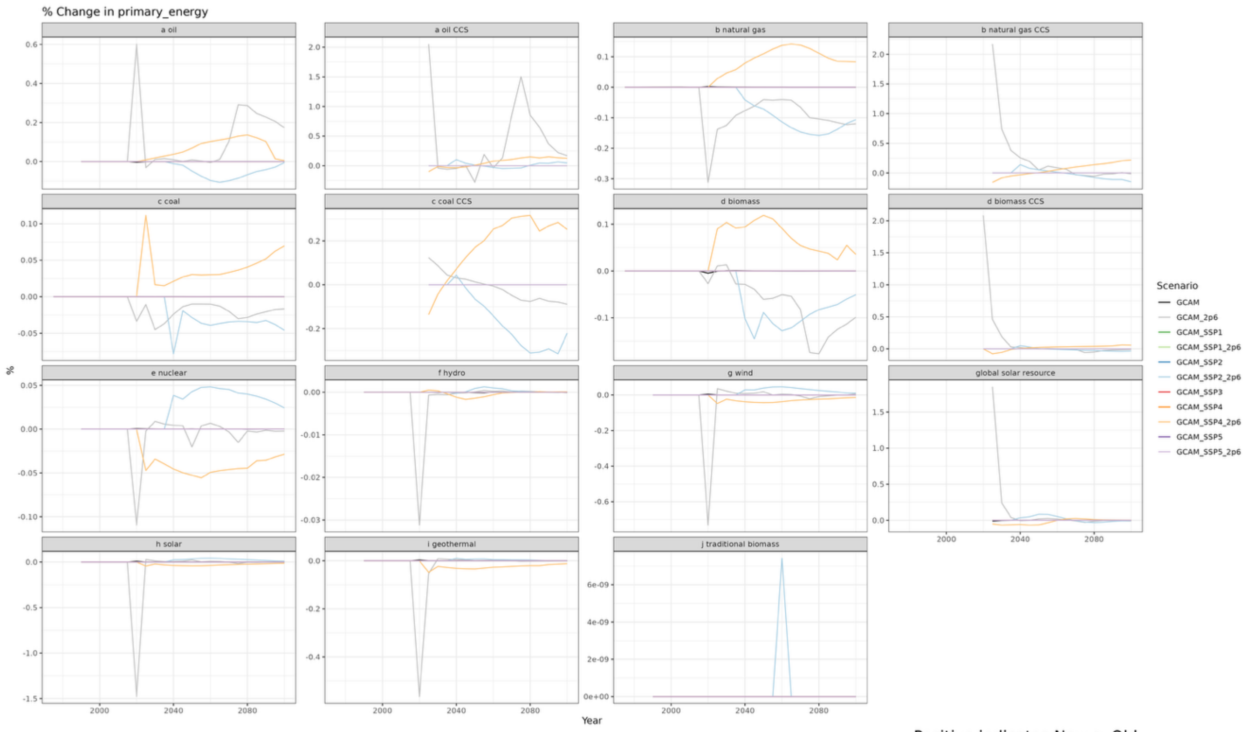
Dotted is Old. Solid is New.



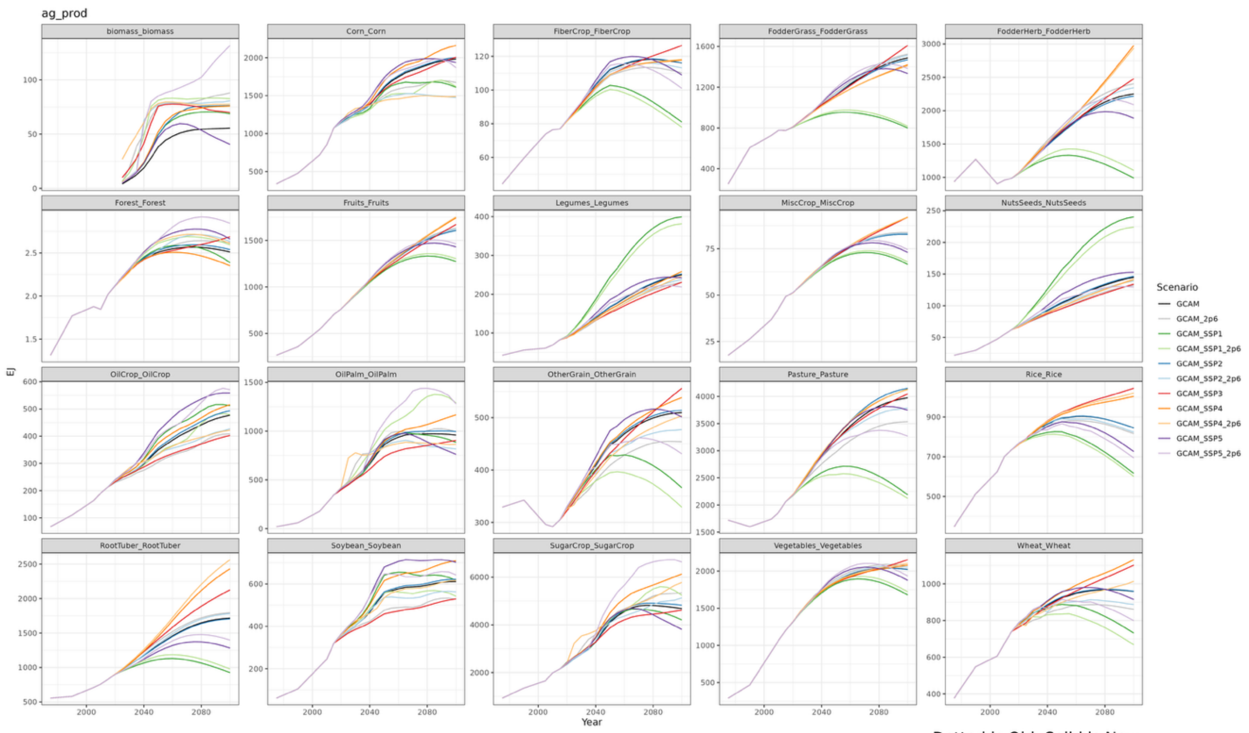
Positive indicates New > Old.



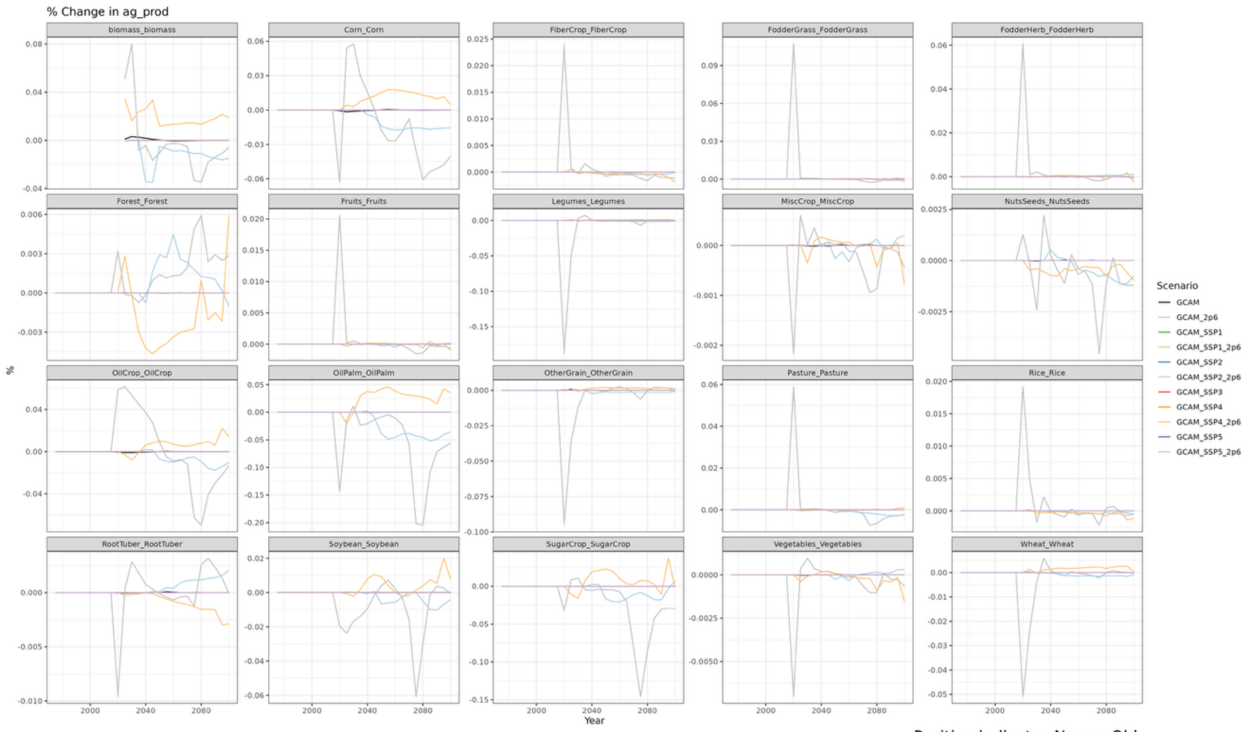
Dotted is Old. Solid is New.



Positive indicates New > Old.



Dotted is Old. Solid is New.



Positive indicates New > Old.