Assignment 3 (CS 798: Multicore programming—Fall 2018)

Due date: December 11

Submit: a single ZIP file on Piazza, including a PDF file in your ZIP.

Readme

- There are a lot of instructions in this assignment, but that's because we are covering a lot of small topics in it. The long instructions should **not** necessarily translate into a lot of work.[1]
- I'm using this assignment as a last teaching opportunity on a few simple topics. In particular, I suggest reviewing my **tips for using bash to run experiments** at the end of this document before you get into the nitty gritty of running your experiments.
- Download the starting C++ code from `http://tbrown.pro/multicore/a3code.zip`.
- You will need hardware transactional memory (HTM) support for this assignment. (See A2.)
- In this assignment, unless I state otherwise, you are going to use **jemalloc** to perform fast scalable allocation for your data structures. (Instructions below.) **I am including a compiled shared library file `libjemalloc.so.2` for 64-bit Linux.** If you do your development on a non-Linux system, simply develop without using jemalloc, then use jemalloc when you run your final experiments on the gpu1-3 servers.
- You will use DEBRA to **reclaim memory** for your data structures. (Instructions below.)
- All of the data structures you implement in this assignment should be integrated into the provided `benchmark_set.cpp`, which can be used to perform any necessary experiments. It is somewhat similar to the hash table benchmark from assignment 2. Run `benchmark_set` with no arguments for usage instructions.
- To make it easier to review and run your code, this time I'm asking you to use *specific file names*, and *specific algorithm names* (for the argument `-a` to `benchmark_set`).

Using jemalloc

Compiling your program. Simply compile as usual. It is *not necessary* to compile with jemalloc. You can replace the default glibc `malloc`/`free` and C++ `new`/`delete` calls *dynamically* with jemalloc's implementations by using `LD_PRELOAD`.

Running with jemalloc. To use jemalloc in your program, make sure you are running in a folder that contains `libjemalloc.so.2`, and use `LD_PRELOAD` as follows. If you would normally run "`./myprogram [args]`", you should run "`LD_PRELOAD=./libjemalloc.so.2 ./myprogram [args]`".

Sanity checks. One way to test if `LD_PRELOAD` is working is to assign an invalid value to `LD_PRELOAD` and see if you get an error message. When I run "`LD_PRELOAD=nonsense.so ls`", I see the error: "`ERROR: ld.so: object 'nonsense' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.`" If you don't see this error, it's a reasonable

[1] I bet this assignment took me way longer to prepare than it will take you to solve. :)

bet that things are working as expected. (Seeing a performance difference between running with-/without jemalloc is also a good sign.)

If you are interested to build jemalloc yourself (e.g., to play with options for memory leak checking, or debug output), see instructions at the end of this document.

1 KCAS-based unbalanced external BST

A paper entitled "Verifying Concurrent Data Structures Using Data-Expansion" developed a method for (relatively) easily proving linearizability of searches in some data structures. You can get it here: `https://link.springer.com/chapter/10.1007/978-3-319-26850-7_11`. (Don't worry, you shouldn't really need to read it.)

In this question, we are going to consider Listing 1 of that paper (page 4), where the authors present a sort of *proto-algorithm* for an external binary search tree. The algorithm specifies that small blocks of code should be performed *atomically*, but does not specify *how* one should synchronize threads to achieve atomicity. Interestingly, they prove that completely synchronization-free searches are correct in this proto-algorithm, *regardless of how atomicity is achieved* (as long as the atomic blocks appear to be atomic to all threads).

1.1 Implementation

I have provided a *single-threaded* implementation of their tree (in which the "atomic" blocks are trivially atomic, because there is only one thread) in the file `trees/unbalanced_seq.h`.

Note that a special `NodeHandler` class is implemented in `trees/unbalanced_seq.h`, as part of the class `UnbalancedTreeSequential`. The `NodeHandler` class implements a clean, uniform interface for traversing the tree. This interface is used by a powerful `TreeStats` class in `tree_stats.h`, which computes a variety of statistics about the tree and prints them at the end of each trial. These stats are enabled by compiling with argument `-DCOLLECT_OPTIONAL_TREE_STATS` (at `Makefile:5`).

Create a KCAS-based implementation of this BST in file `trees/unbalanced_kcas.h`. I should be able to run your algorithm in `benchmark_set` with argument `-a unbalanced_kcas`.

- Use the lock-free KCAS algorithm provided in `kcas/kcas_reuse_impl.h`.
- You can use last assignment's silly array-based data structure as an example of how to actually use this KCAS implementation.
- Be careful to make sure all fields that can be modified by KCAS are
  (a) of type `volatile casword_t`,
  (b) only read using `readPtr` (if the field *contains* a pointer) or `readVal` (otherwise), and
  (c) only initialized with `writeInitPtr` (if writing a pointer) or `writeInitVal` (otherwise).
- Make sure it passes checksum validation for a variety of inputs!
- My recommendation: first implement and test with a *single thread*, then test multi threaded.

1.2 Memory reclamation

In this part of the question, you will reclaim memory for your KCAS-based algorithm using DEBRA. Although we didn't cover memory reclamation in detail during the lectures, we did cover it in student presentations, and it is an important part of actually programming concurrent data structures. So, this assignment provides an opportunity to learn how this can be done in practice.

- I have included code for a `record_manager` class in `recordmgr/record_manager.h`. This can be used to allocate objects and reclaim them using DEBRA when it is safe to do so.
- Instructions for using the `record_manager` appear below.

Creating a record_manager To use a `record_manager` to reclaim memory for your data structure, start by adding `#include "record_manager.h"` to your data structure's header file, and adding a pointer to a `record_manager` object to your data structure.

A `record_manager` has three mandatory template arguments followed by any number of optional ones, so a pointer to a `record_manager` looks like: `record_manager<Reclaim, Alloc, Pool, ObjectType1, ObjectType2, ...> * mgr;`

In this declaration, `Reclaim` is a class type that specifies how objects will be reclaimed, `Alloc` is a class type that specifies how objects will be allocated, and `Pool` is a class type that specifies how (or if) objects will be pooled and reused instead of being freed.

`ObjectType1`, `ObjectType2`, ... are the object types that the record_manager will reclaim. So, assuming your data structure uses nodes of type `Node`, and there are no other data structure objects that you want to reclaim, your declaration would look something like: `record_manager<Reclaim, Alloc, Pool, Node> * mgr;`

So what should your `Reclaim`, `Alloc` and `Pool` arguments be? I suggest `reclaimer_debra<>`, `allocator_new<>` and `pool_none<>`. So, your final declaration is: `record_manager<reclaimer_debra<>, allocator_new<>, pool_none<>, Node> * mgr;`

In the constructor of your data structure, you would then perform: `mgr = new record_manager <reclaimer_debra<>, allocator_new<>, pool_none<>, Node>(maxThreads);`

Note that `maxThreads` should be an *upper bound* on the number of threads you will run (to avoid your memory layout changing drastically based on the number of threads you run, which might affect your experimental results). All `record_manager` functions take a thread ID (in the range 0 ... maxThreads - 1) as an argument. (If the main thread is running alone, and it needs to access `record_manager` functions, it can simply use a "dummy" thread ID of 0.)

Safely accessing nodes *At the start* each data structure operation (contains, insert or delete), before accessing any nodes in the data structure, you must invoke `mgr->startOp(threadID);`. *At the end* of each data structure operation, after you are finished accessing the data structure, you must invoke `mgr->endOp(threadID);`. Between these two points you running in a *guarded section* (meaning your accesses to the data structure are guarded so that nodes you are able to reach will not be reclaimed while you might be using them).

There is a *shortcut* possible here: you can simply perform `auto guard = mgr->getGuard(threadID);` at the beginning of a data structure operation. This function calls `startOp` for you, and returns a *guard* object that will automatically call `endOp` when it goes out of scope.[2]

[2]If you have a *retry loop* where you start your operation over from the beginning in each iteration of the loop,

Allocating nodes The `record_manager` offers an `allocate` function that you can use to allocate nodes. But, since the `record_manager` is designed to allow you to allocate and reclaim *many* types of objects, you have to tell it what kind of object you want to allocate. This is done with a template parameter, as follows: `auto newNode = mgr->template allocate<Node>(threadID);`

Reclaiming nodes Once you unlink a node `x` from the data structure, you can safely reclaim it by invoking `mgr->retire(threadID, x);`. The node will not immediately be freed. Rather, it will be freed only once no thread can access it.

To *immediately* free node `x` (useful when you know no other thread can possibly have a pointer to it), you can call `mgr->deallocate(threadID, x);`.

The task to complete *Implement* memory reclamation in your KCAS-based BST, and then *verify that there are no memory leaks* (or errors) using `valgrind`.

- I recommend testing with a single thread first, then moving onto multi threaded execution.
- I also suggest using `assert()` calls wherever you can to verify that the invariants you *think* are true actually are true.

1.3 Default vs fast allocator

Compare the performance of your KCAS-based BST (with memory reclamation) with (a) the default allocator, and (b) jemalloc. How much does jemalloc improve performance / scaling?

1.4 Does this approach generalize to an *internal* BST?

The paper above gives an algorithm for a linearizable *external* BST with synchronization-free searches and atomic modifications. However, they say nothing about whether this algorithmic idea would work for an *internal* BST that is implemented using similar techniques.

Suppose we were to implement an *internal* BST by making its searches (and the search part of updates) synchronization-free, and then performing its updates atomically (using marking, as in the external BST, to avoid scenarios where operations modify deleted nodes).

A more detailed algorithm A `contains` operation is simply a *sequential* BST search that stops as soon as it finds the key it is looking for, or when it hits a leaf. Failed `insert`s and `delete`s are essentially the same as `contains`. A successful `insert` performs a sequential BST search, ending at a leaf, then performs the following *atomically*:

1. If the leaf is marked, restart the whole operation
2. Create a new node and insert it under the leaf

A successful `delete` performs a sequential BST search, ending at a node `x`, then performs the following *atomically*:

1. If `x` or `parent(x)` is marked, restart the whole operation
2. If `x` is a leaf, mark it and unlink it

---

then you can probably put this line *inside* the retry loop (at the start of the loop body). Just make sure (a) you only access nodes when a *guard* is in scope, and (b) you start accessing the data structure from the root in a guarded section. It is *not* permitted to remember pointers to nodes deep in the data structure, start a guarded section, and then access those nodes.

3. If `x` has one child, mark it and unlink it
4. If `x` has two children, mark it, find the successor, change the appropriate child pointer of `parent(x)` to replace `x` with its successor, and unlink the successor at the bottom of the tree

Short answer questions

1. Is this algorithm correct?
2. If not, describe an execution where a search behaves incorrectly.
3. If it turns out that searches are incorrect, is the incorrect behaviour limited to searches, or could it affect updates as well?
4. Could you hope to catch errors in this implementation using checksum validation (the same kind we did in assignment 2)?

2 TLE-based unbalanced external BST

I have included a simple but powerful TLE library that I wrote for this assignment in `tle.h`. You can (and should) use this, instead of rolling your own TLE implementations. The relevant class in `tle.h` is `TLEGuard`, which functions somewhat like the guard object created by my `record_manager`'s `getGuard` operation. `TLEGuard` is an RAII (resource acquisition is initialization) implementation of TLE, which means once you simply create a `TLEGuard` object on your stack, you are automagically executing inside a TLE critical section until that `TLEGuard` object goes out of scope.

If you want to manually end the TLE critical section *before* the `TLEGuard` goes out of scope, you can invoke `TLEGuard::manually_exit`. After invoking `TLEGuard::manually_exit`, if you'd like, you can reuse the same `TLEGuard` object to start a *new* TLE critical section by invoking `TLEGuard::manually_enter`.

The constructor for the `TLEGuard` requires that you provide it with a pointer to a `TLEData` object, which stores the global TLE critical section lock, as well as per-thread statistics for commits/aborts/fallback executions. So, to use the `TLEGuard` in your data structure, you should create a `TLEData` object as part of your data structure (and have each thread provide a pointer to this `TLEData` object whenever it declares a new `TLEGuard` on its stack).[3]

The destructor for `TLEData` prints detailed transaction statistics (commits, aborts and the reasons for them, fallback executions).

1. Starting from the sequential BST in `trees/unbalanced_seq.h`, produce a basic **TLE-based** BST in file `unbalanced_tle.h`. I should be able to run this algorithm in `benchmark_set` with argument `-a unbalanced_tle`.
   What should be executed atomically:
   `contains`: atomic { search }
   `insert`: atomic { search & modification to the tree }
   `erase`: atomic { search & modification to the tree }

2. Reclaim memory with DEBRA. When combining TLE and DEBRA memory reclamation, you want to ensure that DEBRA's `auto guard = mgr->getGuard(tid)` is performed **outside** of

[3]For this assignment, I could have just created one TLEData object and hardwired it into the TLEGuard class, but it makes more sense conceptually to have one TLEData per data structure, since there is no need to protect *all data structures* in a program using *the same TLE critical section.*

the TLE transaction (to avoid including the overhead of memory reclamation in the TLE critical section).

3. This part of the question is about *more fine-grained* TLE critical sections. Suppose you took your TLE-based BST and modified it so that searches (and the search part of updates) are performed *before* the TLE critical section begins. In other words, we are considering the following, where the code for performing the "search" and the "modification to the tree" are the same as in the `unbalanced_tle` algorithm (and the algorithm only differs in where the TLE critical section begins/ends):
   `contains`: search (non atomic)
   `insert`: search; atomic { modification to the tree }
   `erase`: search; atomic { modification to the tree }

   Would the resulting algorithm be correct? If so, explain why. If not, give a counter example. (Hint: remember the fallback path.)

4. Design and implement a *fine-grained* HTM-based BST (possibly using a modified version of TLE) that *can* safely perform searches outside of the transaction. Use filename `unbalanced_tle_fine.h`. I should be able to run this algorithm in `benchmark_set` with argument `-a unbalanced_tle_fine`.

5. Perform a simple experiment to compare the performance of the improved algorithm (from part 3) to the basic TLE-based BST (from part 1) and your KCAS-based BST.

6. Try the KCAS-based BST with your **TLE-based KCAS** from the previous assignment (instead of the lock-free KCAS I provided here). Use file `trees/unbalanced_kcas_tle.h`. I should be able to run this algorithm in `benchmark_set` with argument `-a unbalanced_kcas_tle`. How does this compare, in terms of performance, with your other BST implementations?

3 TLE-based *balanced* tree

1. In `trees/avl_seq.h`, I provide a purely sequential AVL tree implementation (roughly based on the relaxed AVL tree of Bougé et al., which was used in the Speculation Friendly search tree and several others that we saw in class). Convert this sequential implementation into a TLE-based one in file `trees/avl_tle.h`. I should be able to run this algorithm in `benchmark_set` with argument `-a avl_tle`.
   What should be executed atomically:
   `contains`: atomic { search }
   `insert`: atomic { search & modification to the tree & rotations }
   `delete`: atomic { search & modification to the tree & rotations }

2. In part 1, you performed an update and all of its rotations in one huge TLE critical section. What about using *smaller* TLE critical sections? Recall that, after a successful insert/delete operation, rotations are performed, possibly at each level in the tree, moving upwards.
   Design an algorithm with more fine-grained transactions, in which a successful insert/delete operation should perform its search and modification in one transaction (with no rotations), and *then* perform each necessary rotation separately in its own transaction. To guarantee atomicity, after a thread starts a new transaction for a rotation, it must *validate any values*

that were read in the previous transaction to ensure they haven't changed!
Hint: one of the things guaranteed in the first transaction (where you search from the root) is that the nodes you are accessing are *not deleted* (otherwise your transaction would have aborted). After you start a subsequent transaction for a rotation, how can you validate to ensure that nodes are still not deleted?[4]
Implement this new algorithm from part 1 in file `trees/avl_tle_fine.h`. I should be able to run this algorithm in `benchmark_set` with argument `-a avl_tle_fine`.

3. Perform a simple experiment to compare the performance of your AVL tree implementations from part 1 and part 2, as well as your unbalanced BST implementations. Which tree is fastest? Explain the results.

4. Recall the Speculation Friendly search tree (SF) that was presented in class. It was implemented in a popular (but not-terribly-rigorous) benchmark called Synchrobench. In this question, you will clone Synchrobench and use it to experiment on the SF tree.

   - To clone Synchrobench, run `git clone https://github.com/gramoli/synchrobench.git`.
   - Building with `make` will create a `bin` folder that you will run in.[5]
   - To benchmark SF in Synchrobench with 50% `insert`s and 50% `delete`s on uniform random keys in range $[0, 10^6)$ for 1000 milliseconds, run command `./ESTM-specfriendly-tree -t 12 -i 500000 -r 1000000 -u 100 -f 0 -d 1000`.
     See argument details here: `https://github.com/gramoli/synchrobench`.

   Compare your BST algorithms (running in `benchmark_set.cpp`) with SF (running in Synchrobench).[6] **Use jemalloc to allocate memory for Synchrobench (via `LD_PRELOAD`).**

5. In question 1.2, I asked you to verify that your KCAS-based BST with memory reclamation has no memory leaks or memory access errors by using `valgrind`. Run Synchrobench with `valgrind`. What do you think? :)

BONUS

Measure the performance of your fastest HTM-based tree and fastest KCAS-based tree using supermalloc vs jemalloc.

[4] Note: this question should not require massive changes to the TLE AVL tree code. I was able to get away with changing only 28 lines.

[5] Warning: the benchmark is poorly designed and fails to build if you specify the `-j` argument on your make command.

[6] Of course, this is not exactly a fair comparison, but it would be too much work to run a proper apples-to-apples comparison. My hope is that this will still be interesting.

Optional: Thinking about transaction splitting

I initially wanted to make this a question, but I don't want to overburden you. That said, I'll bring it up anyway (with no obligation), since it's a nice problem to think about. We talked about doubly linked lists in class, and explained how it's hard to get them right unless you use some strong primitive like KCAS or HTM. If you use KCAS, you still have to carefully prove correctness for searches, because KCAS just implements *modifications* atomically, but says nothing about searches. HTM makes it easy to reason about searches, but only if they are performed inside a transaction. However, searches in a linked list, if performed inside a transaction, will almost certainly cause the transaction to abort.

So, one thing researchers have tried doing is **splitting** transactions. This involves traversing a linked list by reading a *small number of nodes* in a transaction (say, 5 nodes), then committing the transaction and starting a new transaction. Of course, when you commit a transaction and start a new one, you lose some guarantees. For instance, when you start the new transaction, you don't know whether the nodes you read in your last transaction are still in the data structure, or have been deleted since you ended your last transaction. Intuitively, you have to add some synchronization to stitch transactions together, restoring these lost guarantees.

The question is, can you use TLE *efficiently* (in a way that causes few aborts) to implement a doubly linked list *with easy correctness arguments.*

The work you did in 2.4 and 3.2 should go a long way in terms of preparing you to actually solve this problem. Why is this question interesting after you've already solved those? In those problems, splitting transactions probably doesn't improve performance much (at least it didn't for me). In contrast, a TLE-based doubly linked list is *totally infeasible* without transaction splitting. (Overall, I think the *idea* of splitting transactions to improve performance is still very valuable.)

Optional: Tips for using bash to run experiments

When I run experiments, I like to use `bash` to run all of my trials, creating a *separate output file* for each trial (so I can go back and check my debugging output if something unexpected happens), then using the tools `grep` and `cut` to extract relevant information from each output file and process it into CSV format.

For example, consider the following output, which is the result of running one trial.

```
Cmd: ./benchmark_set -a unbalanced_tle -t 1000 -s 200000 -n 12 -i 50 -d 50
MAX_THREADS=256
totalThreads=12
keyRangeSize=200000
insertPercent=50
deletePercent=50
millisToRun=1000

prefilling round 0 ending size 100104 total elapsed time=0.234s
prefilling completed to size 100104 (within 5% of expected size 100000 ...

main thread: experiment starting...
main thread: experiment finished...

tree_stats_numInternalsAtDepth=1 2 4 8 16 32 64 127 241 434 747 1250 1875 2570 ...
tree_stats_numLeavesAtDepth=0 0 0 0 0 0 0 1 13 48 121 244 625 1180 1848 2598 ...
tree_stats_numNodesAtDepth=1 2 4 8 16 32 64 128 254 482 868 1494 2500 3750 5140 ...
tree_stats_numKeysAtDepth=0 0 0 0 0 0 0 1 13 48 121 244 625 1180 1848 2598 3189 ...

tree_stats_height=51
tree_stats_numInternals=99755
tree_stats_numLeaves=99756
tree_stats_numNodes=199511
tree_stats_numKeys=99756

tree_stats_avgDegreeInternal=2.00001
tree_stats_avgDegreeLeaves=1
tree_stats_avgDegree=1.5
tree_stats_avgKeyDepth=24.0567

Validation: sum of keys according to the data structure = 9977911171 and sum ...
sizeChecksum=99755

completedOperations=33767631
throughput=33767631

total elapsed time=1.292s
```

```
[abort breakdown]
aborts           : unexplained 94177
aborts           :  conflict 1486
aborts           :  retry conflict 304611
aborts           :  capacity 829582
aborts           :  explicit 9920
aborts           :  explicit conflict 8
total aborts     : 1239784
total commits    : 33757117
total fallback   : 10514
```

Suppose this output is stored in a file called `step10001.txt`. If I want to extract the **throughput** from this file, I can simply run `grep throughput step10001.txt`, which will output any line containing the word "throughput," in this case, `throughput=33767631`. If you want to throw away the "throughput=" and keep only the number, you can use the `cut` tool, which cuts a string into fields based on a delimeter. If we pipe the output of grep to `cut -d"=" -f2` (by executing `grep throughput step10001.txt | cut -d"=" -f2`) we will get only the number: `33767631`. I can store this in a bash variable called `tput` by running `tput=‘grep throughput step10001.txt | cut -d"=" -f2‘`. Note that those quotes around the entire grep/cut command are *backquotes* (the key to the left of "1" on your keyboard).

Of course, this means we can run all of our trials in for loops, output the results to files, and parse the files into a CSV (which we can operate on easily in, say, Excel). Consider the following bash script (included in the code as `example_script.sh`):

```
#!/bin/bash
step=10000
echo step,alg,nthreads,throughput
for alg in unbalanced_tle avl_tle ; do
    for n in 1 2 4 6 8 10 12 ; do
        # increment step number
        step=‘expr $step + 1‘
        # new filename
        f=step$step.txt
        # run command and store the result in filename $f
        LD_PRELOAD=./libjemalloc.so.2 ./benchmark_set [...args...] > $f
        # extract throughput from file $f
        tput=‘grep "throughput" $f | cut -d"=" -f2‘
        # print this row in CSV format
        echo $step,$alg,$n,$tput
    done
done
```

On my system, this outputs the following. Nice!

```
step,alg,nthreads,throughput
10001,unbalanced_tle,1,4289502
10002,unbalanced_tle,2,8495246
```

```
10004,unbalanced_tle,4,15827588
10006,unbalanced_tle,6,22700139
10007,unbalanced_tle,8,27361721
10008,unbalanced_tle,10,31442914
10009,unbalanced_tle,12,36656753
10010,avl_tle,1,3678002
10011,avl_tle,2,7047561
10013,avl_tle,4,13484011
10015,avl_tle,6,18853916
10016,avl_tle,8,22613538
10017,avl_tle,10,27524055
10018,avl_tle,12,26401577
```

Why do I make my filenames step10001.txt, etc.? Because with a step counter I don't need to think too hard about filenames, and I know the output of every experiment is saved. It is easy to print the contents of one of these files (no super long filename), and it's easy to know that I am not overwriting previous files because I forgot to include some experimental parameter in the filename. (For example, if I tried to name my files `exp.$alg.$nthreads.txt`, and I later expanded my experiments to vary the size of the key range, and forgot to include the key range in the file name, after the first key range, experiments on subsequent key ranges would overwrite my earlier experiments.)

I make *rerunning* individual steps easy by linking my data to the file (by including the step number in the data), and linking the file to the command that generates it (by including the command in the first line of each data file). To see what command produced step10002.txt, I can simply run `cat step10002.txt | head -1 | cut -d":" -f2`. If I want to rerun a *range* of steps, it's easy to do so with an if-else statement in my inner bash loop. (Semantics: if step is not between 10182 and 10452, skip it, else run it.)

Why start at step10000 instead of step1? Because if you start at step1, then in a directory listing, step2 is ordered after step10 and step11, and step100, and so on. This bothers me. If I start at 10000, as long as I run less than 90000 trials, my steps are ordered properly (because ASCII order = numeric order).

Why create files at all? Because then you can save massive amounts of debug data in a flexible, human readable format, and know that you have that data available if you want to debug performance anomalies that you see in your graphs, long after the fact. Outputting and saving this debug information (ideally including as much information about the system and OS environment as possible) can not only improve your understanding of performance issues looking back, but can also guard against stupid errors that arise when you think you ran one kind of experiment, but really ran another, and simply didn't save debug data that would expose your mistake.

Why not simply emit nice CSV data from your benchmark directly? Because whenever you output new data, you have to change your CSV format, which can make any scripts or Excel spreadsheets that parse your CSV data obsolete. IMO, it's better to add a script that takes your full human-readable output and produces a CSV. With that extra abstraction layer, you can change your output as much as you like without threatening to break the CSV format (unless you really want to change the CSV format).