Analyzing Linux on a Supercomputer

Spinellis, Diomidis

DOI
10.1109/MS.2024.3512732

Publication date
2025

Document Version
Final published version

Published in
IEEE Software

Editor: Diomidis Spinellis
dds@aueb.gr

Analyzing Linux on a Supercomputer

Diomidis Spinellis

THE C AND the C++ programming languages rely on a versatile but archaic and easily abused feature called a *preprocessor*. Before the source code is seen by the compiler proper, the preprocessor manipulates the source code according to special directives embedded in the source code. These directives may cause some parts of the code to vanish (conditional compilation), may insert the contents of another file in the specified place (file inclusion), and may also substitute a token or a series of tokens that resemble a function call with a specified code (macro replacement). The preprocessor boosted the C programming language's performance and portability, but its naive processing of the source code, without taking into account the language's syntax as well as scope and type semantics, is nowadays making it a liability.

To study the issue and possible countermeasures, I decided to analyze the usage of the C preprocessor by the modern Linux kernel using the CScout refactoring browser.[1] However, when I commenced the analysis, I came across a big problem. As the CScout's data structures quickly swelled to contain the kernel's millions of identifiers, its performance dropped from the thousands of lines per second it can normally process to a measly 29. This meant that processing the 20 million lines of the selected kernel configuration would take eight days—and many more if I had to restart the process a few times to fix errors or fine-tune it.

CScout is designed and constructed to use efficient data structures and algorithms, so optimizing it further would be difficult. Consequently, I followed the easy optimization method: throw money at the problem. Here I describe how I shrank the Linux kernel analysis execution time from several days to hours by running the code on an (expensive) supercomputer.

Of Supercomputers and Supercars

Early supercomputers could execute sequential programs much faster than other computers available at the time. They did that through fast (but pricey and power-hungry) electronics as well as through more sophisticated hardware architectures that offered features such as pipelining, instruction prefetching, and floating-point instructions. Nowadays, thanks to Moore's law, our laptops can execute a sequential program (such as CScout) almost as fast as a modern supercomputer.

So how do modern supercomputers justify their eye-watering price tags? By offering myriad computing cores: 11 million for *El Capitan*, the one currently at the top of the league. In that, modern supercomputers resemble a supertanker more than a supercar (Figure 1). What a modern supercomputer lacks in sequential program execution speed it offers by its ability to execute in parallel a huge number of tasks.

And how can one program such a supercomputer? Most modern supercomputers (and all top-500 ones) run versions of Linux together with diverse specialized programs that handle things such as distributed storage provision, software package installation, and resource management. A frequently used system to allocate access to computer nodes through queues and to manage the starting, stopping, and monitoring of executed jobs is the Slurm Workload Manager.[2] Both Slurm and the commands that launch it are based on the Unix shell, so with that knowledge at hand, one can easily put a supercomputer to work.

Divide and Conquer

To utilize a supercomputer's immense power, the challenging part is dividing the work at hand among the supercomputer's many processing cores. For the Linux kernel analysis I was undertaking, I decided to split

Digital Object Identifier 10.1109/MS.2024.3512732
Date of current version: 25 February 2025

the processing of the roughly 23,000 source code files into 32 jobs, each processing about 750 files on a separate node. (The number of jobs was dictated by the supercomputer resources I was allocated when I applied for its use.)

CScout operates by reading a specification file detailing the source code it needs to process. Then, it can output analysis results, offer a web interface, or store the results as 21 tables in a relational database. For the Linux kernel analysis, I adopted the database option, planning to write code to merge the 32 generated databases into a single one. A CScout tool, *csmake*, can create the required specification file by monitoring the *make*-based compilation process.

Automatically generated files are often easy to process because they have a regular structure that allows the processing to be done with regular expressions and a simple state machine rather than with full-fledged parsing. In this case, all that was required for splitting the specification file were 45 lines of *awk* code.

With the split specification files at hand, I submitted a Slurm job to run an array of 32 tasks on corresponding supercomputer nodes. Next is an excerpt from the job's specification shell script.

```
#!/bin/sh
#SBATCH --job-name=cscout
#SBATCH --array = 1—32
#SBATCH --ntasks = 1
#SBATCH --cpus-per-task = 1
#SBATCH --mem = 128G
#SBATCH --time = 1-00:00:00
filebase=$(printf tasks/file-%04d
$SLURM_ARRAY_TASK_ID)

cscout -s sqlite ${filebase}.cs 2>${filebase}.err |
sqlite3 ${filebase}.db 2>${filebase}-sqlite.err \
 >${filebase}-sqlite.out
```

The Slurm-specific SBATCH comments configure the job to run in a set of 32 nodes, each providing 128 GB of RAM (CScout processing can require a lot of RAM) for a maximum run time of one day. The **filebase** variable is set to contain the name of each task (for example, **tasks/file-0013**), which is then used to specify the names of the CScout specification file, the generated database, and the error logging files. Thus, CScout is set to run on each task's specification file, piping SQL commands to the **sqlite3** command to populate the database.

While the 32 analysis tasks were running, I started working on the merging code, naively expecting to have it completed by the time the job would finish. Reasoning that the merging would also be an expensive operation, I decided to follow a binary tournament merge strategy[3]: pairwise merge the 32 databases into 16, then the 16 into eight, then into four, two, and finally, into a single one (see Figure 2).

Most table merging operations were based on temporary tables that mapped entity keys (such as those for identifiers, files, macros, or functions) of the two databases into a common numbering scheme. The process ensured that entities that existed in both databases (for example, a commonly included header file) would share the same key. With the renumbering table at hand, the merging of two CScout tables from two databases involved creating a new table containing only a single instance of each common entity mapped to the shared identifier. (See the example in Figure 3 for the source code files entity.) Furthermore, all entity keys provided as foreign keys were renumbered to follow the shared numbering scheme. This operation involved 21 SQL scripts comprising about 700 lines and another 1,860 lines of RDBUnit[4] tests.

The merging code assumes that the merged tables also contain the temporary key mapping tables. As this is not the case for the initial set of 32 tables, I employed a small trick to avoid the cost of crafting special code and tests to populate them with mapping tables referring only to a single table rather than two. This involved initially merging each of the 32 tables with an empty database using the same schema.

The following Unix Bash function illustrates the heart of the merging process.

(a) (b)

FIGURE 1. (a) and (b) Supercomputers resemble a supertanker more than a supercar. [Source: Ford GT image credit Ruben de Rijcke (https://commons.wikimedia.org/w/index.php?curid=67643343), CC BY-SA 4.0.]

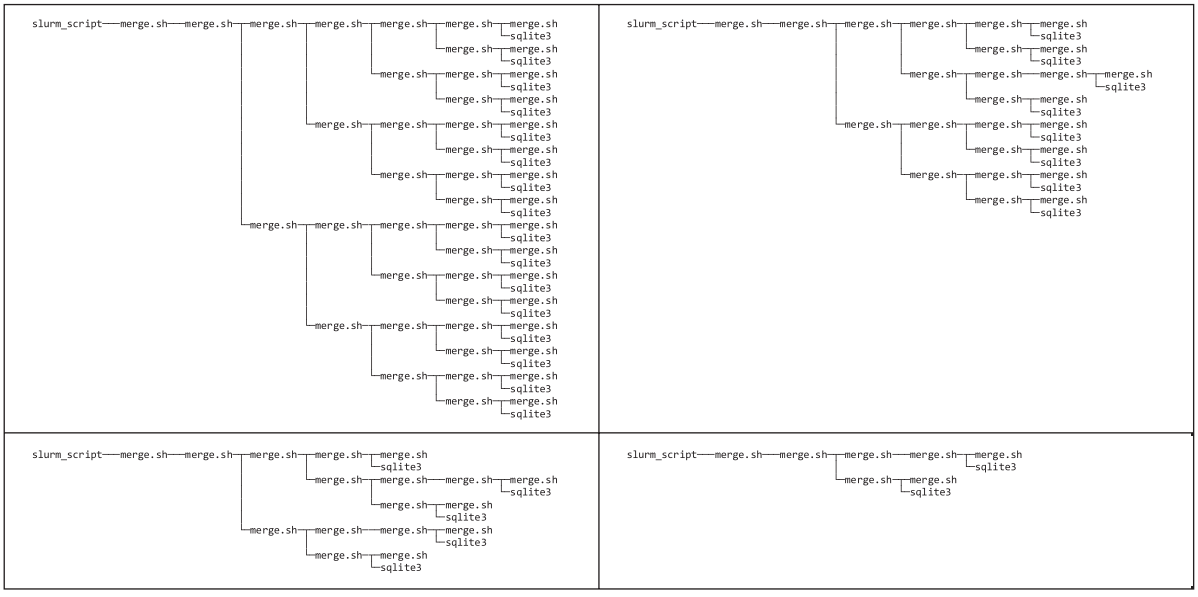

FIGURE 2. The binary tournament merge process tree at 0:00:00 (top left), 1:29:14 (top right), 2:04:50 (bottom left), and 3:57:35 (bottom right).

```
merge()
{
    local files=("$@")
    if ["${#files[@]}" -eq 2; then
        output="temp-$(get_dbid).db"
        create_empty "$output"
        merge_onto $output "${files[0]}"
        merge_onto $output "${files[1]}"
        echo $output
        return
    fi
    midpoint=$((${#files[@]}/2))
    local left=("${files[@]:0:midpoint}")
    local left_output=$(mktemp XXXXX.txt)
    merge "${left[@]}" >$left_output &
    pid_left=$!

    local right=("${files[@]:midpoint}")
    local right_output=$(mktemp XXXXX.txt)
    merge "${right[@]}" >$right_output &
    pid_right=$!

    wait $pid_left
    left_output_db=$(<$left_output)

    wait $pid_right
    right_output_db=$(<$right_output)

    merge_onto $left_output_db $right_output_db
    echo $left_output_db
}
```

The function takes as an argument a list of 2^N database files to merge (initially all 32) and outputs the name of the merged database file. The base case (in the **if** block) handles the merging of just two files, which involves creating an empty database and merging onto it the first file and then the second one. Otherwise, the function divides the files into a left and a right half set. It then recursively launches two merge operations, one for each half, which will execute asynchronously (in parallel) by terminating their invocation statement with the **&** (background execution) operator. It redirects their output into temporary files from which it will later fetch the name of the resulting database. When both operations finish, it retrieves the names of the two merged databases, merges the right one onto the left, and returns the left's database name as the result. Thus, the script initially launches a tree of processes, which at its leaves merges the 16 database pairs (Figure 3, top left). As time goes by, more and more database files are merged until just two databases remain to be merged into one. Almost like magic, the figure's process tree is initially constructed from left to right, and then the merge reduction operations take place from right to left. As all processes have the same parent, the merge operation is run on a single supercomputer node utilizing 16 cores, as specified through the Slurm invocation parameter **--cpus-per-task = 16**.

If at First You Don't Succeed…

The implementation work proceeded smoothly, with unit tests neatly demonstrating the correctness of the SQL scripts I wrote. However, when I tried merging actual databases, the results were often wrong.

The culprit was a table holding C identifier equivalence classes: a data structure indicating, for example, that the global identifier **print** occurring

in several code places in multiple databases was indeed the same one. In contrast to other merged entities for which CScout defines a single canonical representation, which can then be used for merging them across databases, identifiers lack such a representation. A CScout database groups C identifiers that are considered to be semantically equivalent through a shared key assigned to all their records. Each identifier instance is stored through the file and offset in which it appears. Identifier groups from two databases must be merged when they share at least one identifier file and location (for example, a declaration in a commonly used C header file). For example, if one database group has identifiers in file:offset locations 1:53, 2:142, and 3:1,030 and the other in locations 4:95, 2:142, 4:910, and 5:2,345, these two groups must be merged because they share the identifier in location 2:142.

Furthermore, when the results of multiple databases are merged upstream, a newly introduced database entry may require the merging of equivalence classes that were previously separate in a single database. It gets worse. The C preprocessor can dynamically create identifiers by concatenating other ones, for example, concatenating `print` and `_error` to create `print_error`. In a large software system, such as the Linux kernel, identifier splits can occur in different string offsets for the same identifier, which means that the merging may also need to take this into account and therefore resplit identifiers in the merged databases along the union of all split offsets.

Getting this merging right was a humbling experience involving many false starts. Initially, I tried various increasingly complex approaches based on SQL statements. At some point, I had working unit tests for eight possible cases, and yet although the tests were passing, in practice, the merging failed in a few cases.

After I spent several days examining the failing cases, I realized that the problem could not be addressed by the relational algorithm I was implementing in SQL. Instead, it required a graph algorithm, namely that for finding connected components. The graph's nodes are the identifier equivalence classes. One node is created for each group of identifiers in a database that are semantically equivalent, for example, all instances of `dev_t`. Two equivalence classes (from different databases) are connected by an edge when they share at least one identifier. For both classes, the shared identifier will reside in the same file and at the same offset, for example, `dev_t` in the header file `types.h`.

The connected components algorithm can be easily implemented through a breadth-first search (BFS) along the graph's nodes. One can implement BFS using recursive SQL common table expressions. Unfortunately, the SQLite database I was using for storing the CScout results did not support the required multiple references to a recursively specified table or recursive aggregate queries.

To counter this limitation, I located and used an SQLite extension (*bfsvtab*) that exposes a performant BFS algorithm as a virtual database table. However, this approach proved too slow for two reasons. First, it required performing a BFS for every node, which required processing large amounts of data. Second, to encode into the graph node identifiers all the required information, I devised a suitable integer encoding and stored the mapping from the original equivalence classes into the graph node identifiers in temporary tables. It turned out that the SQLite relational join on half a billion elements was too slow, mainly because it performed a nested loop join rather than the more appropriate (in this case) sort-merge join. Even after I created all the required database indexes, the operation to map the graph node identifiers back into CScout equivalence classes generated only a few tens of records per second, which was unacceptably slow.

I then reasoned that the slowness was because too many processes were running on a single supercomputer node, starving it from input/output (I/O) capacity. Normally, I would use diverse performance monitoring tools to verify this hypothesis, but doing this on the nodes of the supercomputer wasn't easy. So, I simply changed the work division from launching several processes on one supercomputer node to scheduling several rounds of mutually dependent Slurm tasks, hoping that these would distribute the I/O load among multiple nodes. This didn't help for two reasons. First, all the tasks were again scheduled on the same node. Second, I hit a limit on the number of supercomputer tasks I was allowed to submit even though most were just waiting for their dependent tasks to finish.

GraphViz to the Rescue?

Back to the drawing board, I decided to calculate the graph's connected

Database 2

Id	Name
1	main.c
2	types.h

Database 5

Id	Name
1	version.c
2	compile.h
3	types.h

Map

DbId	LocalId	GlobalId
2	1	1
2	2	2
5	1	3
5	2	4
5	3	2

Merged

Id	Name
1	main.c
2	types.h
3	version.c
4	compile.c

FIGURE 3. Merging the Files tables from two database instances.

components using the *ccomps* (connected components) command-line program that is distributed with the GraphViz (Graph Visualization Software) open source software package.[5] Initiated by AT&T Labs Research and expertly designed and engineered, GraphViz has served me well on a number of occasions, so I reasoned it would do so again. However, although this approach worked fine for small examples, when I tried merging the databases resulting from the Linux kernel analysis (by that time, the analysis had long finished), *ccomps* appeared to hang.

Hypothesizing that *ccomps* was hitting a fault, I started creating a minimum working example to demonstrate it. I did this by repeatedly cutting the number of input lines by half to find what part of the input was causing it to hang. Surprisingly, at some point, while I was taking notes on the process, the program finished. It turns out that *ccomps* wasn't in fact hanging; it was just very slow. I verified this by timing and plotting a few input sizes. Through the elapsed time plot I realized that it experienced an exponential slowdown as the input size increased.

Digging deeper into the rabbit hole, I then started debugging *ccomps*. Guessing that the issue was expensive memory garbage collection at the end of processing, I ran the *ltrace* (library trace) program on it, expecting to see calls to *free*() memory deallocation function. No such sign. Attaching a debugger to the hung process, interrupting it, and looking at the stack also didn't help; the process appeared to be quite live, exploring the graph.

Then, I downloaded and compiled the source code (this is the beauty of open source software) and started adding print statements to narrow down the area of code that was causing the slowdown. Through this process, I found that it was code that listed any missing edges after the connected components had been output. Because the program had output half a million components, each node was probably checked against all of them, resulting in the extreme slowdown. Given that for efficiency reasons I fed to *ccomps* only edges associated with connected components, I found and used the program invocation option that eliminated that step. After that change, *ccomps* finished in seconds.

With running and unit-tested code at hand, I ran the process on the whole output, and 12 hours later, I had a single merged database file. Full of excitement, I started working on it, but I soon realized that the results I obtained weren't quite what I expected. And yet, my manual verification of the file's contents typically yielded the expected results.

Feeling uneasy with the outcome, I devised a scheme for stress-testing the merging code in a minute rather than in 12 hours. Rather than merging 32 database files containing the analysis of 740 files each, I created eight database files containing the analysis of four files each. I also extended the database's contents to include all the analyzed code elements (not just those I needed for my study) so that I could then fully reconstitute from the database the complete source code and compare it against the original. The merging of the resultant files still performed the pairwise tournament merge but could finish in about a minute. It turned out that creating the testing setup was one of the best decisions I had made in a long time.

Between the time I created the testing setup and the time I had merge code working correctly, I fixed not fewer than 20 logical errors that caused the code to produce incorrect results. Each of them manifested itself as a discrepancy between the original Linux kernel C source code and the one reconstituted from the analysis database. With the fast testing setup, I was able to test each fix in minutes rather than wait a day for the results.

Third Time Lucky

Two of the faults caused me to completely redesign the merging method. First, I gave up on using *ccomps* when I realized that it couldn't merge groups containing parts of dynamically generated C identifiers because the underlying graph lacked the corresponding information. To overcome that issue, I decided to use CScout's tried and tested identifier equivalence class merging capabilities. Therefore, I extended CScout with an option to merge input files describing identifiers, functions, and associated tokens. The database merging process writes these files out for each database, calls CScout to merge them, and then reads back the results into the database to continue the merging.

Second, my initial design for the CScout-based merging operation, built around a state machine that processed the identifiers file, failed because I didn't take into account that the identifiers would not appear in the file in the neat order I had placed them in the 10 test files I had constructed to test the merge operation. I fixed that by abandoning this design in favor of a two-pass approach over the entire identifier set. Even though the CScout approach was much simpler than the others I had tried, I still fixed several faults mainly caused by me trying to optimize the merge process without taking into account diverse corner cases. As I fixed them, I remembered D. Knuth's pronouncement: "Premature optimization is the root of all evil."[6]

In the end, the analysis took just 32 hours of wall clock time using 374 CPU hours and 640 GiB of RAM on the supercomputer's nodes to create a single 27-GiB database file with the results. The laborious implementation process, which took several weeks, taught me many new things regarding the processing of complex large datasets. I hope that sharing the adventure through this column has made you, dear reader, wiser in a less painful way.

References

1. D. Spinellis, "CScout: A refactoring browser for C," *Sci. Comput. Program.*, vol. 75, no. 4, pp. 216–231, Apr. 2010, doi: 10.1016/j.scico.2009.09.003.
2. M. A. Jette and T. Wickberg, "Architecture of the Slurm Workload Manager," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2023, pp. 3–23, doi: 10.1007/978-3-031-43943-8_1.
3. D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Reading, MA, USA: Addison-Wesley, 1973.
4. D. Spinellis, "Unit tests for SQL," *IEEE Softw.*, vol. 41, no. 1, pp. 31–34, Jan./Feb. 2024, doi: 10.1109/ms.2023.3328788.
5. J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz—Open source graph drawing tools," in *Proc. Int. Symp. Graph Drawing*, 2002, pp. 483–484, doi: 10.1007/3-540-45848-4_57.
6. D. E. Knuth, "Structured programming with go to statements," *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974, doi: 10.1145/356635.356640.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, 104 34 Athens, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

Digital Object Identifier 10.1109/MS.2025.3531670