P2329R0
Pablo Halpern <phalpern@halpernwightsoftware.com>
2021-12-13 16:24 EST

Move, Copy, and Locality at Scale

Abstract

This paper (paper, not *proposal*) explores the performance benefits and penalties of preferring move assignment over copy assignment on containers that allocate memory. A relatively simple experiment is described whereby elements within a subsystem are allocated consecutively, in contiguous memory. The elements are randomly shuffled across subsystems using either move assignment or copy assignment, then accessed repeatedly within each subsystem. Although the shuffle step is generally faster when using move assignment, especially for large elements, Preliminary results show that cache effects can more than cancel out the benefit under certain circumstances, e.g., when the overall system does not fit in L3 cache. Moreover, when the system size approaches or exceeds the size of physical memory, data within a subsystem is spread out over many more pages, sometimes resulting in a dramatic slowdown for accessing the data after it had been shuffled using move assignment vs. copy assignment. Programmers should thus be aware that preferring moves to copies does not necessarily result in a performance improvement and might, for some large programs, do the opposite.

Description

We (the author and other engineers at Bloomberg LP) designed an experiment intended to simulate a *system* composed of multiple *subsystems*. Each subsystem creates multiple data "elements," each of which allocates memory. The simulation assumes that these elements are mostly accessed (read and written) within the subsystem, but are occasionally transferred across subsystems. This transfer can be accomplished using either move assignment or copy assignment; move assignment transfers ownership of the allocated memory, whereas copy assignment copies the contents of the allocated bytes.

Under what conditions does the use of move assignment yield a performance benefit over copy assignment and vice versa? To answer this question, our simulation is parameterized on the following quantities:

1. the total size, in bytes, of the system (`systemSize`)
2. the number of subsystems in the system (`numSubsystems`)
3. the number of elements in each subsystem (`elemsPerSubsys`)
4. the size, in bytes, of each element (`elemSize`)
5. the number of times the elements are shuffled between subsystems, i.e., *churned* (`churnCount`)

6. the number of times every element in each subsystem is accessed (`accessCount`)
7. the number of times the entire churn/access cycle is repeated (`repCount`)

Note that `systemSize` is assumed to be the product of `numSubsystems`, `elemsPerSubsys`, and `elemSize`; thus, a test run supplies values for only three of the first four parameters (using a `'.'` placeholder for the fourth value), and the simulation program computes the fourth value automatically.

The entire simulation is run once using copy assignment in the churn step and then again using move assignment. Each simulated run is timed, and the times are reported as well as the quotient (expressed as a percent) obtained by dividing the move-assignment time measurement by the copy-assignment time measurement. Varying these parameters reveals patterns for when move assignment was beneficial (computed percentage significantly less than 100), detrimental (computed percentage significantly greater than 100), and insignificant, i.e., when the difference was within the noise (computed percentage close to 100).

Because the parameter space is large (7 parameters), and because some run times were long (in excess of an hour), it was not possible to cover the entire parameter space. Therefore the simulation was run within a script that varied two or three parameters at a time while holding the rest constant. In most cases, for each system size, the script allowed the number of subsystem to vary automatically in inverse proportion to the number of elements per subsystem.

We ran three tests. **Test 1** explored system sizes from below L1 cache size to several times L3 size.

parameter	type	values
`systemSize`	varying	2^{13} to 2^{25} bytes
`elemsPerSubsys`	varying	4 to `systemSize/elemSize/2` elements
`elemSize`	constant	128 bytes (2 cache lines)
`churnCount`	constant	1
`accessCount`	constant	4
`repCount`	varying	inversely with `systemSize` (min 32)

Test 2 was similar to Test 1 except that `elemSize` was not a multiple of the cache-line size.

parameter	type	values
`systemSize`	varying	2^{13} to 2^{25} bytes
`elemsPerSubsys`	varying	4 to `systemSize/elemSize/2` elements
`elemSize`	constant	96 bytes (1.5 cache lines)
`churnCount`	constant	1
`accessCount`	constant	4
`repCount`	varying	inversely with `systemSize` (min 32)

Test 3 explored very large system sizes, exceeding physical memory, when `accessCount` greatly exceeded `churnCount`:

parameter	type	values
`systemSize`	varying	2^{32} to 2^{35} bytes
`elemsPerSubsys`	varying	8 to `systemSize/elemSize/16` elements
`elemSize`	constant	64 bytes (one cache line)
`churnCount`	constant	1
`accessCount`	constant	8
`repCount`	constant	5

Source for this micro-benchmark can be found at https://github.com/phalpern/WG21-halpern/tree/P2329/P2329-move_at_scale. The three tests are implemented in the `runtest` shell script.

Results of the above three tests are available as a set of `.csv` files in the `R0-results` subdirectory of the source repository. Each line of each `.csv` file lists the test arguments, the time (in ms) spent on the simulation using copy assignment, the corresponding time (in ms) for move assignment, and the move-assignment time as a percentage of the copy-assignment time (less than 100% if move was faster, greater than 100% if copy was faster).

Test platform

The results were generated on a MacBook Pro with the following specifications (source: GeekBench Browser):

- Model: MacBook Pro 2018 (Model ID: MacBookPro15,1)
- CPU: 6-core Intel Core i7, 2.2 GHz
- L1 Data Cache: 32KiB per core
- L1 Instruction Cache: 32KiB per core
- L2 Cache 256KiB per core
- L3 Cache 9MiB shared
- RAM: 16GiB
- Disk: 512GB SSD

Observations

Rather than publishing page after page of raw numbers, we invite the reader to view the resulting `.csv` files in the repository listed above.

Although this paper does not present a comprehensive analysis, the data yielded the following observations.

- With 128-byte elements and a 32MiB total system size, move assignment yielded up to a 2x speedup (50% run time) with a large number of small

subsystems but a 2x slowdown (189% run time) with a small number of large subsystems.

- With 64-byte elements and a 4GiB or 8GiB total system size, move assignment was typically worse than copy assignment (up to 7x worse), but even in this case, move assignment was significantly faster than copy assignment for large numbers of small subsystems.
- The results were somewhat noisy; in many cases, consecutive runs with only small parameter changes resulted in large swings. One theory is that the alignment of elements on cache lines was different between such runs. Nevertheless, certain patterns did emerge, such as the large slowdowns caused by page thrashing.

Future directions

We recognize that this experiment is not the final word on this subject. Specifically, we are considering making the following improvements.

- Run on a greater variety of hardware.
- Experiment with element sizes that are not necessarily uniform.
- When computing the total system size, take into the account that the system, subsystems, and elements are all `vector` types and have their own footprints.
- Experiment with a memory resource that allocates items in the smallest possible number of cache lines.
- Experiment with using a non-polymorphic memory allocator.
- Run experiments where subsystem accesses run in concurrent threads.
- Cover a larger portion of the parameter space.

Whether and to what degree our simulation reflects real-world programs is unclear. Measurements on real, memory-bound programs would be ideal.

Conclusions

- In a great many circumstances with non-trivial element sizes, using move assignment instead of copy assignment results in a performance improvement, sometimes dramatically so.
- However, in large systems with individual subsystems that exceed the size of the L3 cache, copy assignment can provide better spacial locality when the number of data accesses exceeds the number of transfers by a factor of 4 or more.
- The deleterious effect of non-locality on performance caused by using move assignment is even more dramatic if the total system size exceeds the size of physical memory.
- When the number of data accesses greatly exceeds the number of transfers, the benefits of move assignment are diluted, but the potential downside on performance remain larges.

- These effects are not always predictable. As always, the best approach is to measure.
- Programmers should not assume that move assignment is preferable to copy assignment in large-scale programs.