Acc++: Accelerator for C++ STL

phase 1 progress report

Chen Chen, Yifan Hong, Yunhan Hu, Zirui Tao

March 4, 2020

1 Introduction

The phase 1 report consists of trails of profiling tools, profiling analysis on real applications, and proposed ideas for acceleration, which follows from the chronological order of project progress.

2 Profiling tools

2.1 GNU gprof

The GNU profiler gprof [1] uses a hybrid approach of compiler assisted instrumentation and sampling. To gather profiling information at runtime, the program counter is probed at regular intervals by interrupting the program with the operating system interrupts. Yifan first realized that as sampling is a statistical process, the resulting profiling data are not exact but are rather a statistical sampling approximation by gprof with induced bias. In addition, as we found that, it has been reported as sometimes losing track of parent function, as mentioned in this discussion. Nevertheless, since gprof is relatively stable and widely used, **we select it as the primary profiling analysis tool** in this project. Other than these two drawbacks, gprof provides a clean representation of the histograms of time spent on each function based on its total samples. This part is primarily done by Chen Chen.

2.2 gperftools

Gperftools [2] from Google provides a set of tools aimed at analyzing and improving the performance of multi-threaded applications. We focus on their sampling-based CPU profiler, which has a very little runtime overhead, provides some nice features like selectively profiling certain areas of interest and has no problem with multi-threaded applications. Unfortunately, when we profile with gperftools, the profiler shows raw hex addresses instead of function names, which doesn't profile analysis friendly. This name mapping bug with x86-64 system is also reported by others and still not solved. Therefore we have to set it aside temporarily. This part is primarily done by Yifan Hong.

2.3 Pin

The pin tool [3] is a software instrumentation tool by Intel that allows easy, programmable profiling tools. It provides a variety of profilings such as, all of which is is implemented through a C++ class API, which handy to customize. In this project, Zirui followed through the DreamLandCoder's blog on implementing instrumentation of program that outputs the accessed memory address and values, which is accessible to Project code repository. This allows Pin to be a very powerful programming instrumentation tool to analyze almost any program behaviors. However, our goal is to analyze the performance bottleneck, the aforementioned 2.1 and 2.2 are sufficient for estimating the time utilization for each leaf function. This part is primarily done by Zirui Tao.

2.4 Valgrind

Valgrind is a powerful software development tool that can be used to analyze memory errors, function calls, cache usage, stack and heap management, and multi-threaded contention. In this project, we use the callgrind, recording function calls and generating call graphs, and cachegrind, counting the cache misses and hits, in the Valgrind. The function of callgrind overlaps with the function of the previous tools, so it is only used to be a reference. Cachegrind is the main tool used. For the benchmarks we interest, we will record the cache usage. For the benchmarks with high hit rates, we will give up because they are hard to be improved. This part is primarily done by Yunhan Hu.

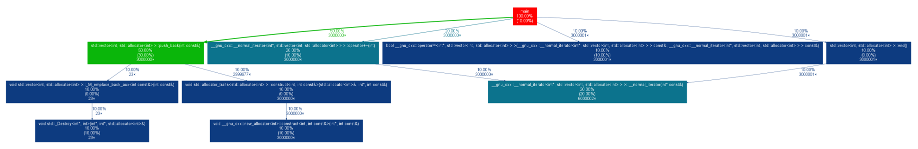

Figure 1: function DAG for simple *push_back* to *std::vector* program, cutoff at 10% runtime for visualization, figure generated by Zirui. Full figure can be access through: `https://github.com/tzrtzr000/AccPlusPlus/blob/master/ref/func_dag_t.png`

3 Profiling Analysis

In total, we examined the gprof on matrix pseudoinverse operation using Arma, Opencv and Eigen separately, originally written by Nghia Ho, which could be accessed through our code repository under benchmarks. In addition, we also examined *Fill* benchmark, provided by Wicht [4]. Due to the page limit, we instead focus on reporting the analysis case of the *Fill* benchmark.

The *Fill* benchmark analyzes the time complexity of STL C++ containers, such as vector, deque and list, upon invoking *push_back* leaf function. The size of testing elements ranges from 100,000 to 1,000,000. Table 1 shows that over 80% of system time is taken up by a GNU extension function to the Standard C++ Library, called *new_allocator*[5], which uses *global new* to manage heap memory allocation.

After determining the leaf function with the highest contribution, we begin to analyze its underlying mechanism. To isolate the analysis of *push_back* function, we write a simple program that contains *std::vector* with *push_back* function and compiled C++ source code to x86 based assembly instructions. We then manually traced the program order using *gdb* and discovered that the *new_allocator* handles the allocation and de-allocation operation of std::vector's elements, according to [5]. Additonally, we used gprof and grof2dot to convert profiling output to function calls as directed acyclic graphs(DAG), as shown in Fig. 1.

The *Fill* benchmark helps us get familiar with couple analysis tools. We then proceed similar analysis on a real applicaton: matrix pseudoinverse operation benchmark, with Arma, Openvv and Eigen environments separately. By profiling real benchmarks, we discovered some accelerators design opportunities which is covered in section 4.

Leaf Function	**Time Contribution %**
new_allocator(Trivial(4096))	64.49
new_allocator(Trivial(1024))	16.37
new_allocator(std::list(4096))	2.80
new_allocator(Trivial(128))	1.35
other	14.99

Table 1: Leaf Function Contribution, table generated by Chen Chen

4 Proposed Plan

Based on the analysis on all applications mentioned at the beginning of 3, not limited to one reported, we proposed three potential ideas for accelerating C++ programs.

4.1 Memory Allocation

By tracing through assembly through gdb, we found that "insert" operation (*push_back, Fill*) generates a great number of instructions (s200) to only allocate new heap memory space, which shows opportunities to design accelerators for dynamic memory allocation and size of class computation.

Although existing literature [6] has examined the solutions on accelerating malloc function specifically on C++, it is still possible for us to find alternative design strategy for heap manager. Intuitively, the heap management process should take up a critical portion of computation time in real applications, which still worth us to understand it deeply.

4.2 Iterator Implementation

As a significant part in C++ STL, iterators are primarily used in sequence of numbers, characters etc. As shown in Fig.1, for instance, iterator functions are the second most overhead after allocation. In some particular simple benchmark the percentage of iterators can even reach 60%. Therefore, finding optimization opportunities on hardware-level for iterators would be promising and worth trying. Next step we will dive in iterators' instruction-level behaviors, trying to achieve some regular patterns, then optimize the implementation for specific usage.

4.3 Value Assignment Speculation

After the analysis, Chen noticed that the *push_back* function copies the value of the source variable and moves it into the vector container, which requires expensive copy instructions and the system may need to spend extra time to remove the reference[7] from source variables. A novel value assignment speculation scenario can be designed to release this pressure.

In C++ syntax, there exists a fundamental implementation difference between Lvalue and Rvalue assignment: move or copy values from the source to the destination. [8, 9]. In short, Lvalues' assignments maintain source values while Rvalues' assignment does not, which involves the expensive copy instructions and cheap moving pointer operation separately. While newly introduced C++ compiler specifically distinguishes the Lvalues and the Rvalues through rigid static definition, Zirui proposed that during dynamic execution, some Lvalue reference could be equivalent to Rvalue reference (as long as there is no write to either source and destination variable). By presenting a subset of Lvalue as Rvalue, the execution would be able to benefit from extra performance gain from Rvalue implementation. As discussed with Prof. Lipasti, OS already adopted the Copy on Write (COW) [10] policy, in which a copy from the instruction of duplication is deferred until writes: the source and destination would initially share the same physical address. An OS exception occurs when either one of the variables performs a write operation, in which a memory allocation is performed, followed by a value calculation and assignment. The COW policy is based on the assumption that all assigned operators tend not to be written as frequent as to be read, thus, avoiding copes in the first place could benefit from the performance and memory allocation, even the penalty for extra OS exception handling for write is expensive. Similar to branch prediction and value prediction, Zirui proposed that there may be opportunities for the predictor to predict read and write pattern for certain variables. However, in order to maintain the correctness of execution, the source value should still be efficiently "tracked", therefore a follow-up solution is to design a data buffer for hardware to handle write exception normally handled by OS. The plan for this idea is to first examine whether this idea would have a significant impact on performance or energy efficiency. If it does, then a follow up fine-grained experiments would be performed to exploit the most ideal design parameter.

References

[1] Susan L Graham, Peter B Kessler, and Marshall K Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*. ACM, 1982, vol. 17, pp. 120–126.

[2] "gperftools," https://github.com/gperftools/gperftools, [Online; accessed 22-November-2019].

[3] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*. ACM, 2005, vol. 40, pp. 190–200.

[4] Baptiste Wicht, "C containers benchmark: vector/list/deque and plf::colony," May 2017.

[5] "libstdc source documentation, https://gcc.gnu.org/onlinedocs/gcc-4.6.3/libstdc /api/a00056.html," .

[6] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks, "Mallacc: Accelerating memory allocation," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 33–45, 2017.

[7] "std::remove_reference," http://www.cplusplus.com/reference/type_traits/remove_reference/, [Online; accessed 22-November-2019].

[8] Howard E. Hinnant, Bjarne Stroustrup, and Bronek Kozicki, "A brief introduction to rvalue references," https://www.artima.com/cppsource/rvalue.html, [Online; accessed 22-November-2019].

[9] "Rvalue references: C++0x features in vc10, part 2," https://devblogs.microsoft.com/cppblog/rvalue-references-c0x-features-in-vc10-part-2/, [Online; accessed 22-November-2019].

[10] Wikipedia, "Copy-on-write — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Copy-on-write&oldid=925169638, 2019, [Online; accessed 22-November-2019].