automemcpy: A Framework for Automatic Generation of Fundamental Memory Operations*

Guillaume Chatelet
Google Research
France
gchatelet@google.com

Chris Kennelly
Google
USA
ckennelly@google.com

Sam (Likun) Xi
Google
USA
xyzsam@google.com

Ondrej Sykora
Google Research
France
ondrasej@google.com

Clément Courbet
Google Research
France
courbet@google.com

Xinliang David Li
Google
USA
davidxl@google.com

Bruno De Backer
Google Research
France
bdb@google.com

Abstract

Memory manipulation primitives (`memcpy`, `memset`, `memcmp`) are used by virtually every application, from high performance computing to user interfaces. They often consume a significant portion of CPU cycles. Because they are so ubiquitous and critical, they are provided by language runtimes and in particular by *libc*, the C standard library. These implementations are heavily optimized, typically written in hand-tuned assembly for each target architecture.

In this article, we propose a principled alternative to hand-tuning these functions: (1) we profile the calls to these functions in their production environment and use this data to drive the important high-level algorithmic decisions, (2) we use a high-level language for the implementation, delegate the job of tuning the generated code to the compiler, and (3) we use constraint programming and automatic benchmarks to select the optimal high-level structure of the functions.

We compile our memfunctions implementations using the same compiler toolchain that we use for application code, which allows leveraging the compiler further by allowing whole-program optimization. We have evaluated our approach by applying it to the fleet of one of the largest computing enterprises in the world. This work increased the performance of the fleet by 1%.

CCS Concepts • **Software and its engineering → Software development techniques**; • **General and reference → Measurement**; **Performance**; *Empirical studies*; *Metrics.*

Keywords memory functions, C standard library

ACM Reference Format:
Guillaume Chatelet, Chris Kennelly, Sam (Likun) Xi, Ondrej Sykora, Clément Courbet, Xinliang David Li, and Bruno De Backer. 2021. automemcpy: A Framework for Automatic Generation of Fundamental Memory Operations. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3459898.3463904

*Our memory function implementations [14], benchmarking methodology [11, 13] and raw measurements [10] have been open sourced.

ISMM '21, June 22, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8448-3/21/06.
https://doi.org/10.1145/3459898.3463904

1 Introduction

Interacting with memory is an essential part of software development. At the lowest level the two fundamental memory operations are *load* and *store*, but developers frequently use higher level memory primitives such as *initialization*, *comparison*, and *copy*. These operations are at the base of nearly all libraries,[1] language run-times,[2] and even programming languages constructs.[3] They may be customized for some particular contexts—such as for kernel use or embedded development—but the vast majority of software depends

[1]e.g., *std::string* and *std::vector* in the C++ Standard Template Library.
[2]Garbage collectors *move* memory to reduce fragmentation, reflection APIs rely on run-time type identifier *comparison.*
[3]In some `C` implementations, passing a large `struct` by value inserts a call to libc's `memcpy`.

Table 1. Overview of `memcpy` source code language for various architectures and *libc* implementations.

	bionic	glibc	freebsd	dietlibc	uClibc	eglibc	klibc	musl
aarch64	asm	asm	asm	-	-	-	-	asm
arm	asm	asm	asm	-	-	asm	-	asm
i386	asm	asm	asm	asm	C	asm	-	asm
x86-64	asm	asm	asm	asm	asm	asm	-	asm
alpha	-	asm	-	-	-	asm	-	-
generic	-	-	-	-	-	-	C	C
ia64	-	asm	-	-	asm	asm	-	-
mips	-	asm	asm	-	-	-	-	-
powerpc	-	asm	asm	-	C	-	-	-
s390	-	asm	-	-	-	asm	-	-
sh	-	asm	-	-	asm	asm	-	-
sparc32	-	asm	-	asm	asm	asm	-	-
sparc64	-	asm	-	asm	-	asm	-	-

on their implementations in the C standard library: `memset`, `memcmp` and `memcpy`[4].

In the rest of this paper, we will focus on the optimization of `memcpy` as the hottest of the three functions, but the same approach can be applied to the other two.

As a reminder, `memcpy` is defined in §7.24.2.1 of the *C standard* [21] like so: *The* `memcpy` *function copies n characters from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.*

2 Overview of Current Implementations

2.1 Pervasive Use of Assembly

Memory primitives have been part of the C standard library for at least three decades. They have seen persistent efforts to exploit the capabilities (and work around the quirks) of each architecture as it evolved over time. Developers, who were typically well-versed in the subtleties of their target micro-architecture, would choose assembly to obtain the maximal amount of control on the resulting generated code. As a result, out of the eight main publicly available *libc* implementations (see Table 1), all but one use assembly to implement `memcpy`. As shown in Table 2, *glibc*[5] alone has six x86-64 versions of `memcpy` to exploit different instruction set extensions such as SSE2, SSE3, AVX, and AVX512.

2.2 Dynamic Loading and Relocation

Standard C libraries are usually provided as *shared libraries*, which brings a number of advantages:

- it avoids code duplication—saving disk space and memory,
- it enables quick updates—this is especially important for maintenance and security reasons,
- to the developer, it accelerates the compilation process by reducing link time.

Shared libraries also come with costs:

- Symbols from shared libraries can't be resolved at compile time and they need extra run-time infrastructure. Modern Linux systems and x86-64 in particular implement shared libraries using *Position Independent Code* (PIC) [5–7]. PIC avoids costly relocation of function addresses at load time by using an extra indirect call through the *Procedure Linkage Table* (PLT). This indirect call hurts performance and increases *instruction Translation Lookaside Buffer* (iTLB) misses.
- Functions imported from shared libraries are not visible to the linker. This prevents optimizations such as inlining and *Feedback-Directed Optimization* (FDO). We discuss FDO in more detail in Section 2.4.

In Google data centers, applications are statically linked and their size is typically in the hundreds of megabytes. Statically linking the memory primitives only marginally increases the binary size and overcomes the aforementioned problems.

2.3 Run-Time Dispatch

As CPUs and *Instruction Set Architectures* (ISAs) evolve, new instructions become available and performance of older instructions may improve or degrade. Unless a *libc* is built for a specific CPU model, it must accommodate the older models but it should also provide an optimized versions for newer CPU models. To this end, Linux libraries use a run-time dispatching technique on top of PLT called *IFUNC* [26] that is commonly used by *glibc*. On the first call to an IFUNC, the indirect address—stored in the *Global Offset Table* (GOT)—points to a custom *resolver* function that determines the best implementation for the host machine. The resolver then overrides its own GOT entry so that subsequent calls access the selected function. After the setup phase, an IFUNC has the same indirect call costs and limitations as a function from a shared library, even if it is linked statically.

Table 2. Binary size of *glibc* 2.31 `memcpy` implementations for x86-64.

Name	Size (bytes)
`__memcpy_avx512_no_vzeroupper`	1855
`__memcpy_avx512_unaligned_erms`	1248
`__memcpy_avx_unaligned_erms`	984
`__memcpy_sse2_unaligned_erms`	765
`__memcpy_sse3`	10695
`__memcpy_sse3_back`	10966

[4]`memmove` is not considered in this paper as: (1) it requires additional logic and (2) in our experience, its use is anecdotal compared to `memcpy`, `memcmp` and `memset`.

[5]The GNU C library.

2.4 Feedback-Directed Optimization

The most significant optimizations in recent years came from the mainstream adoption of Feedback-Directed Optimization[6] (FDO) with global efficiency improvements typically over 10% [15]. FDO relies on a measured *execution profile* to make informed compilation decisions[7] based on how the application actually executes the code. Because memory primitives are so pervasive among all applications, they are especially well covered by sampling profilers[8] and could greatly benefit from FDO.

However, FDO can only be applied to the parts of the application for which source code is available to the compiler/linker. In particular, it can't be applied to dynamically linked functions and IFUNCs. Moreover, FDO does not understand and is not allowed to modify code written in assembly. This effectively means that the way memory functions are implemented on most systems prevents them from benefiting from one of the most efficient optimization techniques.

3 Our Approach

Based on our analysis of the shortcomings of current implementations, we design our implementation primitives around two main principles:

- **High-level description.** The implementation should be written using a high-level language—we chose C++ for its expressiveness. This makes the code shorter and more readable, and it allows the compiler to use advanced optimization techniques, including FDO. When needed, we examine the generated code to debug performance issues, but never write assembly directly.
- **Data-based optimization.** On top of fleet-wide execution profiles, we record run-time values of the `memcpy` `size` argument for several representative workloads. We use the measured `size` distribution to automatically select the best set of elementary strategies and design the decision logic driving the function.

3.1 Measuring `size` Distributions

There are several ways of collecting the run-time values for the `size` argument:

1. Manually instrument our `memcpy` implementations.
2. Rely on compiler's instrumentation and value profiler.
3. Rely on profiling infrastructure.

Solutions 1 and 2 require a special (instrumented) build of each application and add an extra run-time cost which is impractical for use in production. Instead, we make use of existing profiling infrastructure: the Linux `perf` tool [27]. This is an interface to the Linux performance monitoring subsystem that can attach to a running process and collect profiling data without a need to recompile or restart the application. Among other features, it allows sampling register values at certain points during the execution of the program. In our case, we set it up to sample the `RDX` register right after a `call` instruction. Under the System V x86-64 ABI [25], this register contains the `size` argument of `memcpy`. Since `memcpy` is statically linked, its address in the binary is known at compile time, so we can easily filter samples belonging to `memcpy` and gather them into a histogram.

3.2 Performance Definition

There are many dimensions to consider when designing a memory function implementation. Contrary to existing implementations that usually optimize for throughput,[9] we will focus on latency and code size as they are most important for our use cases. Code size is measured with the help of the `nm` linux command and latency is measured through a custom benchmarking framework. We define latency as the running time of the memory operation for a given hardware in a given context.

3.3 Benchmarking

Measuring the performance of memory functions is both hard and critical, so we devote a whole section to our benchmarking methodology. In the context of Google, we can rely on fleet-wide profiling to measure the efficiency of an implementation. However, linking statically means that we have to wait for all applications to be released before seeing the results, which is not practical.

The aim here is to provide an accurate and reproducible measurement of latency. To do so we heed the following design principles:

1. **Measuring instrument**. We make use of the *Time Stamp Counter* (TSC). It is the most precise clock available: a hardware register located on the CPU itself, counting the number of CPU cycles elapsed since boot. There are two versions of the TSC on x86-64 processors. One is in-core and ticks at a rate depending on a number of factors (power saving mode, defined maximum and minimum frequency, thermal regulation); the other one is Uncore,[10] which ticks at a fixed frequency usually referred to as *ref cycles*. Most high end CPU architectures provide similar functionality (SPARC, ARM, IBM POWER). The Uncore counter's frequency is about the same as that of the processor in non-turbo mode. We use the Uncore counter to account for uniformly elapsing time.
2. **Frequency Scaling**. The Operating System usually provides a way for the user to control the CPU core frequencies (i.e., Scaling Governors under Linux), but

[6]In some contexts, FDO is also called *Profile-Guided Optimization*.

[7]Examples of what can be achieved are: determining which functions to inline, or how to lay out the code to improve locality and reduce branching.

[8]Modern datacenters continuously run sampling profilers on their workloads as a part of standard operations [22].

[9]Number of bytes processed per unit of time.

[10]https://en.wikipedia.org/wiki/Uncore

the CPU may reduce its frequency automatically under some workloads [19, 24]. By instructing the CPU to run at the maximum allowed frequency and using the uncore TSC, we make sure to account for potential frequency reductions.

3. **Operating System**. To lower the impact of interruption and process migration, we pin the benchmarking application to a reserved core.[11]
4. **Compiler perturbations**. Clever dead-code and dependency analyses performed automatically by the compiler may cancel or distort the measurement. To this end we use techniques such as the `escape` inline assembly described by Carruth [8].
5. **Memory Subsystem**. On modern non-embedded systems, RAM bus frequency is only a fraction of the CPU core frequency. This well-known bottleneck is balanced out by several levels of caching. The number, size, and speed of each of these layers makes it hard to attribute the result of the measurement to the algorithm itself. For this reason, our framework takes extra care to fit all the needed data into the L1 cache. The results are slightly idealized but more reproducible and comparable between CPUs of different cache sizes.
6. **CPU**. At the CPU level, out-of-order execution, superscalar architecture, *Macro-Operation Fusion* (MOP Fusion), Micro-Operation Fusion, and speculative execution make it harder to correlate TSC with individual instructions. To mitigate these imprecisions and increase the *Signal to Noise Ratio* (SNR) we run enough iterations of each function to get noise down to ±1%. Note that the repeated execution will bias the results as the CPU will learn the branching pattern perfectly. Consequently, we randomize the size of the operation as well as the source and destination pointers. Randomization is pre-computed and stored in an array so it is not accounted for in the measurement. The size of the array is large enough to prevent quantization effects but small enough to keep all data in L1.

Additionally, it is worth noting that a number of effects do play a role in production and are not directly measurable through microbenchmarks. We end this section with the following known limitations:

1. The code size of the benchmark is smaller than the hot code of real applications and it doesn't exhibit instruction cache and iTLB pressure as much.
2. For the reasons stated in item 5 above, the current version of the benchmark does not handle large operations spanning multiple cache levels.

[11]If the processor supports *Simultaneous MultiThreading* (SMT) all the logical CPUs sharing the same core should be reserved as well, or SMT should be disabled altogether.

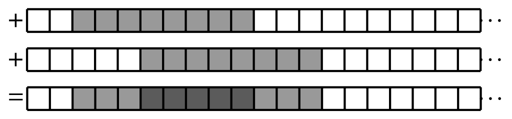


Figure 1. Example Overlapping operation on 11 bytes. Bytes from the source location are copied using two 8-byte block operations that partially overlap. The five bytes in the middle are copied twice.

3.4 Generic Access Strategies

At a high level, a `memcpy` implementation is assembled from a set of simpler copying strategies (*elementary strategies*) along with its associated decision logic that selects the best copying strategy for a given `size`.

By analyzing the code of available implementations and through experimentation, we identified the following patterns for processing memory. In this section, we list these strategies and describe their specifics.

3.4.1 Block Operation

A *Block Operation* consists of accessing a fixed amount of memory of size *BlockSize*. To be efficient, *BlockSize* should correspond to sizes natively supported by *load* and *store* instructions of the CPU architecture. It is the simplest access pattern and is implemented using the simple *load* and *store* instructions provided by the ISA.

3.4.2 Overlap Operation

This access pattern is the composition of two Block Operations that may partially overlap, and it is a central component of efficient implementations. This approach has several advantages. First, a single access pattern involving two operations of size N can handle a range of sizes $\in [N; 2 \times N]$. Second, modern processors offer memory addressing modes that render this pattern efficient in practice.

Figure 1 shows an example of Overlap Operation for 11 bytes. As depicted by the darker cells, one or more bytes may be accessed several times. While this strategy is valid for implementing `memcpy`, such a technique is not suitable when `volatile`[12] semantics apply.

3.4.3 Loop Operation

The Loop Operation involves a repetition of Block Operations followed by a trailing Overlap Operation. Note that up to $BlockSize - 1$ bytes may be accessed twice. Figure 2 shows an example of a Loop Operation.

3.4.4 Aligned Loop Operation

The Aligned Loop Operation starts with a Block Operation to ensure that reads are aligned. It is then followed by a

[12]Note that the `volatile` keyword in C presents a number of issues and is considered for deprecation [4].

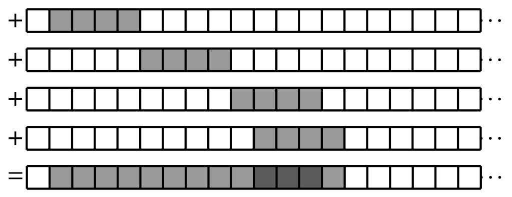


Figure 2. Example Loop Operation on 13 bytes. The first 12 bytes are copied using three non-overlapping 4-byte Block Operations. The remaining one byte is copied using a partially overlapping 4-byte Block Operation. Three bytes are accessed twice.

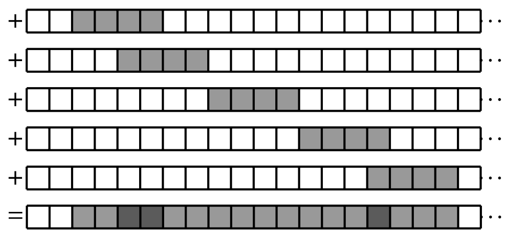


Figure 3. Example Aligned Loop Operation on 17 bytes. Two partially overlapping operations are used at the beginning and at the end. All remaining bytes are handled using aligned non-overlapping Block Operations.

Loop Operation. Note that in a worst case scenario, up to $BlockSize - 2$ bytes may be accessed three times. Figure 3 shows an example of an Aligned Operation.

3.4.5 Special Instructions

Some ISAs provide primitives to accelerate string manipulation. For example, `x86-64` offers specialized instructions for *copy* (`rep mov`), *initialization* (`rep sto`) and *comparison* (`repe cmps`). The performance of these instructions varies greatly from microarchitecture to microarchitecture [28] but it is important to consider them as their footprint is so small they could be easily inlined everywhere. In the rest of this paper we refer to these special instructions as `accelerators`.

3.5 Run-Time Size Coverage

The Generic Access Strategies we presented in the previous section can handle one or more sizes, Table 3 lists their run-time size coverage in terms of *BlockSize*.

These strategies can be composed in different ways to build a full implementation covering the whole `size_t` range [0, `SIZE_MAX`]. By way of illustration, a possible coverage is given in Table 4.

To find the best `memcpy` implementation, we use a constraint solver to enumerate all valid `memcpy` implementations matching our specification. We consider only memory functions that subdivide the range [0, `SIZE_MAX`] into smaller

Table 3. Size coverage for a given strategy

Operation	Size range
Block	$[BlockSize, BlockSize]$
Overlap	$[BlockSize, 2 \times BlockSize]$
Loop	$[BlockSize, +\infty]$
Aligned Loop	$[BlockSize, +\infty]$
Accelerator	$[0, +\infty]$

Table 4. An example of coverage for a memory operation

Size range	Operation	BlockSize
[0, 0]	-	-
[1, 1]	Block	1
[2, 4]	Overlap	2
[5, `SIZE_MAX`]	Loop	4

regions and cover each of them using one of the strategies described in Section 3.4 using the following schema:

```
A                 B        C      D            E
| individual sizes | overlap | loop | accelerator |
```

where `A`, `B`, `C`, `D` and `E` are bounds separating the regions.

If the run-time *copy* `size` is in [`A`, `B`) the copy will be handled with the `individual sizes` strategy, in [`B`, `C`) with the `overlap` strategy, and so on. `A` and `E` are anchored to `0` and `SIZE_MAX` respectively to make sure the whole range is covered (eq. 1), and we make sure that the bounds are ordered (eq. 2).

$$(A = 0) \land (E = \texttt{SIZE\_MAX}) \tag{1}$$

$$(A \le B) \land (B \le C) \land (C \le D) \land (D \le E) \tag{2}$$

When the two bounds of a region are equal, e.g., when $A = B$, the region is empty and the corresponding strategy is not used in the generated function.

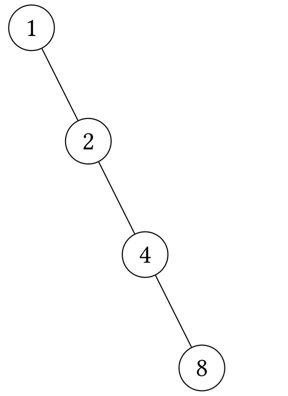
The **`individual sizes`** region handles each run-time size individually with a Block Operation. In this study we restrict `B` to small values as larger values substantially increase code size.

$$(A = B) \lor (B \le 8) \lor (B = 16) \tag{3}$$

The **`overlap`** region handles run-time sizes from `B` to `C` by using one or more Overlap Operations. In this study we explore all `overlap` regions for $size \in [4; 256]$ as well as the empty region.

$$(B = C) \lor \begin{cases} B < C \\ B \in \{4, 8, 16, 32, 64, 128\} \\ C \in \{8, 16, 32, 64, 128, 256\} \end{cases} \tag{4}$$

The **`loop`** region employs the aligned loop strategy with a run-time $BlockSize \in \{8, 16, 32, 64\}$. If $C = D$, we also force `loop_block` = 0 to remove redundant implementations.


(a) Simple tree processing small sizes first

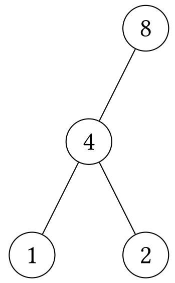

(b) An optimized tree where most values are above 8

Figure 4. A depiction of two branching strategies

$$\left.\begin{array}{r}\text{loop\_block} = 0\\ C = D\end{array}\right\} \vee \left\{\begin{array}{l}\text{loop\_block} \in \{8, 16, 32, 64\}\\ C > 2 \times \text{loop\_block}\\ D - C > \text{loop\_block}\end{array}\right. \quad (5)$$

The **`accelerator`** region makes use of special instructions where available. We allow `accelerator` to start for small sizes or amongst a set of discrete values.

$$(D \leq 16) \vee (D \in \{32, 64, 128, 256, 512, 1024\}) \vee (D = E) \quad (6)$$

3.6 Branching Pattern

Once a coverage has been determined, it is necessary to decide which branch should be favored to lower the whole operation latency. A simple approach is to make sure that small sizes are processed first so that the cost of branching is proportional to the amount of bytes to process (Figure 4a). Another possibility is to prioritize sizes that occur the most. To this end, we make use of Knuth's *Optimum Binary Search Trees* [23] that constructs a binary search tree producing code with the shortest expected control height for a given set of probabilities (Figure 4b).

The implementation can also explore a more elaborate model to represent the cost of compares, branch instructions, and taken branches. A hybrid expansion approach can also be used. For instance when a sequence of individual sizes gets dispatched to Block Operations, all sizes can be lumped together and share one jump table. Note that this approach will be the most effective when it is used for inlining—using per-callsite profile data.

3.7 Optimized Memory Functions

The usage of memory primitives varies widely depending on the use case. One extreme case is the Linux kernel that spends a very significant proportion of its time clearing or copying memory pages, which are always 4096 bytes in length. Just like *libc*, the Linux kernel provides several implementations, with a dynamic selection mechanism. In contrast, for typical applications running on Google fleet, we've found that sizes are very biased towards zero (see Section 4.1). The best algorithm for the Linux kernel is not necessarily optimal for other applications. In this section we provide a principled approach to design efficient memory functions tailored to a given environment:

1. We first model the function in terms of its coverage strategies as described in Section 3.5, and use the Z3 Solver [16] to enumerate all valid coverages. Each of them is described in terms of its parameters `A`, `B`, `C`, `D`, `E`, and `loop_block`, and whether we use an optimized branching pattern or not (Section 3.6).
2. We generate the source code for all of them and encode the values of the parameters into the function name—this allows us to generate all functions in a single file and quickly identify and understand the implementation.
3. The code is then compiled and benchmarked on all relevant microarchitectures using synthetic distributions observed in production.
4. The best performing implementation is chosen, in the case of a fleet of heterogeneous microarchitectures we pick the implementation that minimizes the overall latency.

Not only does this approach allow the functions to be optimized for a particular application (or set of applications) but it also adapts to special requirements like special compilation flags, new microarchitectures or new CPUs. The model described here generates 290 different functions which are benchmarked in one or two hours. It is straightforward to extend the exploration of the implementation space.

The techniques mentioned here are related to the domain of auto-tuning which has been an important field of research in *High Performance Computing* (HPC) [3] and Signal Processing libraries [17, 18, 30]. To our knowledge, the only related work in this domain is the one from Ying et al. [31], focusing on optimal generation of assembly for *Reduced Instruction Set Computer* (RISC).

4 Results

4.1 Memfunctions Size Distributions

The instrumentation procedure described in Section 3.1 introduces a small overhead (1–3%) on the application itself during the time that the profiler is active, due to the overhead introduced by `perf_events` profiling. A 30-second profiling session typically gathers enough data for analysis, depending on the frequency with which the application calls the memory operations. Figures 5, 6, and 7 shows the cumulative probability distribution for sizes of the biggest users of `memcpy`, `memcmp` and `memset` at Google. The data is publicly available [12]. Table 5 summarizes these figures and shows

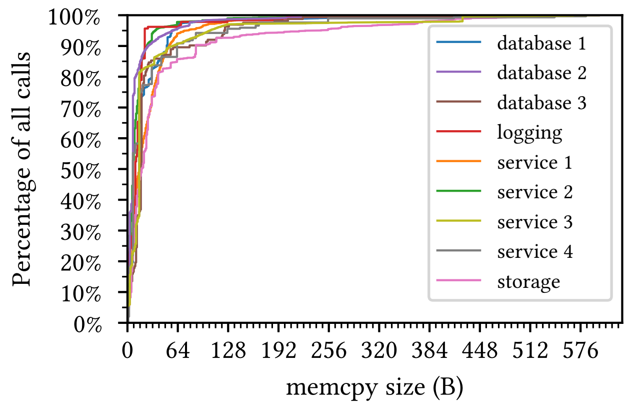


Figure 5. Cumulative histogram of `memcpy` `size` for various Google workloads.

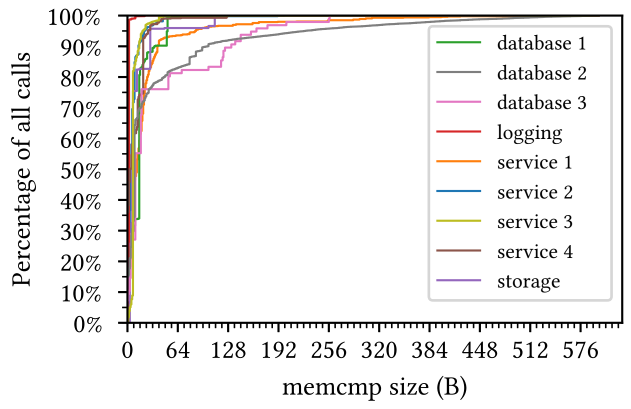


Figure 6. Cumulative histogram of `memcmp` `size` for various Google workloads.

Table 5. Percent of calls below certain size values

Function	% of calls ≤ 128	% of calls ≤ 1024
`memcpy`	96%	99%
`memcmp`	99.5%	100%
`memset`	91%	99.9%

that the majority of calls to memory functions act on a small number of bytes.

The fact that the size distribution is strongly skewed towards small sizes advocates optimizing for *latency* instead of *throughput*. We have seen previously that relocation and run-time dispatch introduce an indirect call which delays processing. For example, a trivial zero `size` memory operation would still have to go through address resolution and perform the indirect call before returning to the calling code. To remove this latency entirely we link the memory primitives

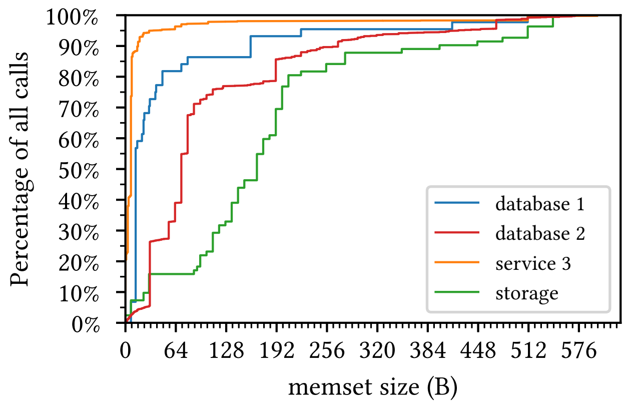


Figure 7. Cumulative histogram of `memset` `size` for various Google workloads.

statically inside the binary and refrain from using IFUNC. This comes with the disadvantage that we can no longer pick the best implementation for the CPU at run-time; this choice has to be made during compilation. Arguably, for distributions skewed towards small sizes, the use of wider load and store instructions available on newer microarchitectures doesn't matter as much. In situations where it is important, this can be mitigated by providing several binaries covering the diversity of the fleet.

4.2 C++ Library

In Listing 1, we provide Block, Overlap and Aligned Loop C++ implementations of the *copy* operation. The individual building blocks are unit tested for correctness, buffer overflow and maximum number of accesses per byte. Note that the use of `__builtin_memcpy_inline` is a compiler intrinsic function that is *semantically* equivalent to a *for* loop copying bytes from *source* to *destination*—i.e., has the semantic of `memcpy`. Its use is necessary to prevent the compiler from recognizing the `memcpy` pattern which leads to a reentrancy problem [9].

4.3 Fleet-Wide Manual Implementation

Now that we have defined the primitive C++ copy functions, we can assemble them to handle all run-time sizes. Listing 2 shows a generic yet efficient implementation of `memcpy` that has been tuned by trying out different building block combinations and parameters. Full source code is available in [14]. Note that this version handles small sizes first. In Section 4.4, we will take a closer look at size distributions and explore optimized branching patterns.

A note on the use of AVX instructions. Although a large portion of Google fleet supports the AVX instruction set extension, their use is restricted to prevent important slowdown due to frequency reduction [19, 24]. To that end

```
template <size_t kBlockSize>
void CopyBlock(char *__restrict dst,
               const char *__restrict src) {
  __builtin_memcpy_inline(dst, src, kBlockSize);
}
template <size_t kBlockSize>
void CopyLastBlock(char *__restrict dst,
                   const char *__restrict src,
                   size_t count) {
  const size_t offset = count - kBlockSize;
  CopyBlock<kBlockSize>(dst + offset, src + offset);
}

template <size_t kBlockSize>
void CopyBlockOverlap(char *__restrict dst,
                      const char *__restrict src,
                      size_t count) {
  CopyBlock<kBlockSize>(dst, src);
  CopyLastBlock<kBlockSize>(dst, src, count);
}
template <size_t kBlockSize, size_t kAlignment = kBlockSize>
void CopyAlignedBlocks(char *__restrict dst,
                       const char *__restrict src,
                       size_t count) {
  CopyBlock<kAlignment>(dst, src);

  const size_t ofla =
      offset_from_last_aligned<kAlignment>(src);
  const size_t limit = count + ofla - kBlockSize;
  for (size_t offset = kAlignment; offset < limit;
       offset += kBlockSize)
    CopyBlock<kBlockSize>(
        dst - ofla + offset,
        assume_aligned<kAlignment>(src - ofla + offset));

  CopyLastBlock<kBlockSize>(dst, src, count);
}
```

Listing 1. Real C++ code for various *copy* operation strategies

we make use of LLVM's `-mprefer-vector-width=128` option that limits the width of generated vector instructions. For this reason, the x86-64 generated code (clang 12, options `-O3 -mcpu=haswell -mprefer-vector-width=128`) will not use AVX. It is still quite compact (377 bytes) and fits in six cache lines.

Performance. It is important to point out that although this implementation does not use all the techniques applied by *glibc* (e.g., non temporal move or `rep mov`) [29], it performs better on Google workloads. As suggested in [2], the smaller memory footprint compared to *glibc* (377B vs 765B, see Table 2) allows more application code to stay in the instruction cache. Indeed `memcpy`, `memcmp`, and `memset` are called on a regular basis and can contribute to instruction cache pressure, evicting previous application code from L1.

```
void memcpy(char *__restrict dst,
            const char *__restrict src,
            size_t count) {
  if (count == 0)
    return;
  if (count == 1)
    return CopyBlock<1>(dst, src);
  if (count == 2)
    return CopyBlock<2>(dst, src);
  if (count == 3)
    return CopyBlock<3>(dst, src);
  if (count == 4)
    return CopyBlock<4>(dst, src);
  if (count < 8)
    return CopyBlockOverlap<4>(dst, src, count);
  if (count < 16)
    return CopyBlockOverlap<8>(dst, src, count);
  if (count < 32)
    return CopyBlockOverlap<16>(dst, src, count);
  if (count < 64)
    return CopyBlockOverlap<32>(dst, src, count);
  if (count < 128)
    return CopyBlockOverlap<64>(dst, src, count);
  return CopyAlignedBlocks<32>(dst, src, count);
}
```

Listing 2. an *efficient* yet *readable* implementation of `memcpy`

Using this version of `memcpy` and associated `memcmp` and `memset` improved the throughput of one of our main services by +0.65% ± 0.1% *Requests Per Second* (RPS) compared to the default *shared glibc*. Overall we estimate that this work improves the performance of the fleet by 1%.

4.4 Fleet-Wide Autogenerated Implementations

As illustrated in Figure 5, most of the sizes for `memcpy` at Google are heavily skewed toward small values. The most occurring ones being spread between 0 and 32 with spikes depending on specific workloads. For the optimized branching pattern (Section 3.6) we use the sum of the measured size probability distribution; giving an equal weight to each of them. The code is compiled and benchmarked on a dedicated pool of machines with isolated cores running the *performance* frequency governor. We benchmark the following x86-64 microarchitectures: Intel Sandybridge, Intel Ivybridge, Intel Broadwell, Intel Haswell, Intel Skylake Server (SKX) and AMD Rome. The benchmarks make use of the `rep mov` instruction and a total of 290 `memcpy` configurations. We measure the average latency for each of the 9 size distributions of Figure 5 and an additional synthetic distribution for larger sizes (uniformly distributed sizes between 384 and 4096). We produce 1000 samples for each of them to get a sense of the statistical distribution. We show a sample of the distributions in Figure 8.

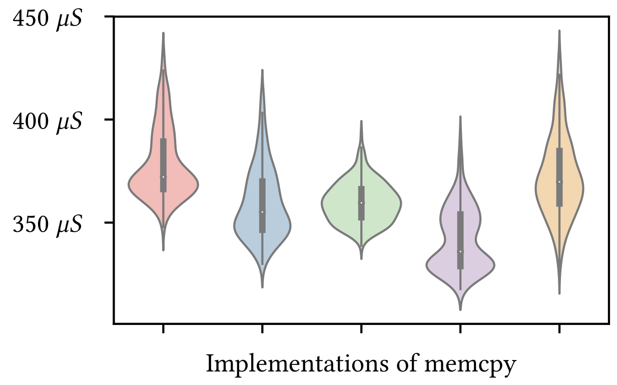


Figure 8. Distributions of latency measurement for several `memcpy` implementations as measured on SKX for *service 1*.

In the context of this paper, we reduce the thousand samples to a single median value[13] as this provides a total order over the set of functions. For each distribution we derive a speedup ratio by dividing each median latency by the one of the *glibc*[14]—which is our baseline implementation. We use the geometric mean (geomean) of these speedups to represent the performance of a particular implementation over the baseline and across the selected distributions. In the rest of the paper, we refer to this metric as the *score* of the function.

This systematic exploration exhibits implementations that perform about +10% faster than our manual implementation for four out of six microarchitectures in our particular environment (combination of workloads and special compilation options). For Haswell and Broadwell, the manual implementation is already close to the optimum. In Figure 9, we plot the score for the 290 autogenerated x86-64 `memcpy` implementations as measured on Skylake Server.

Key Takeaways. Overall, the best performing functions for x86-64 rely on a small `individual sizes` region ($B = 4$ or 8) followed by a set of overlapping copies up to 256 bytes while an aligned copy loop using blocks of 64 bytes helps accommodate the bigger sizes. The `rep mov` instruction may be useful for even bigger copies depending on the microarchitecture but it's usually on par with implementation relying exclusively on aligned copy loop. It is to be noted that the worst performing functions for our workloads are the ones that rely almost exclusively on the specialized `rep mov` instruction.

Timings. The whole process takes a few hours. The enumeration and code generation for all valid implementations takes a few seconds. The duration of the benchmarking process ranges from tens of minutes to a few hours, depending on a number of parameters such as: the number of samples, the number and shapes of the distributions, the number and efficiency of the generated functions, and the number and the performance of the considered microarchitectures. In this study, we used 1000 samples, 10 distributions, 290 functions and 6 microarchitectures. The single threaded benchmark is launched on the 6 microarchitectures in parallel and the results are available in under two hours.

[13]Considering that some distributions of latency measurement are multi-modal we acknowledge that a better metric is desirable.
[14]GNU C Library 2.19 with local modifications.

Table 6. Score for worst, manual implementation and best `memcpy` functions taking the system *glibc* as a baseline.

	worst	manual	best
Rome	0.42	1.35	1.43
Skylake Server	0.54	1.38	1.53
Haswell	0.52	1.39	1.40
Broadwell	0.52	1.42	1.43
Ivybridge	0.57	1.22	1.34
Sandybridge	0.26	1.23	1.35
Neoverse-N1	0.82	1.21	1.36

Open Source. One of our goals is to make our improvements available publicly to people who do not have access to Google fleet. The community should be able to measure performance and contribute improvements, so we released the methodology and the source code of our benchmark as open source [11, 13].

Results. The worst, manual implementation and best performing implementations are summarized in Table 6. It is worth mentioning that the optimized branching pattern does not yield a systematic performance improvement and seems to be quite dependent on the processor microarchitecture. Further investigation is required to determine the usefulness of this optimization.

Case Study. To test the robustness of our method on a different architecture, we conducted the same experiment on an ARM Neoverse-N1 machine. The ARM ISA has no instructions similar to `rep mov`, so the number of generated functions drops to 60. We complement this set of functions with a particular implementation from ARM Optimized Routines [1] to check how close we get to optimized assembly—this implementation is statically linked. In our context, our top performing function has an overall speedup of 1.36 over the system's *glibc*[15] which is about +8% faster than the handwritten assembly scoring 1.26. We note that each of them has different strengths and weaknesses depending on a particular size distribution.

4.5 Tailored Implementations

In the two previous sections, we have shown how we can write a generic implementation of `memcpy` that works efficiently across the fleet. However, we do not claim that all workloads are similar to those that we run at Google. In this

[15]GNU C Library 2.27 with local modifications.

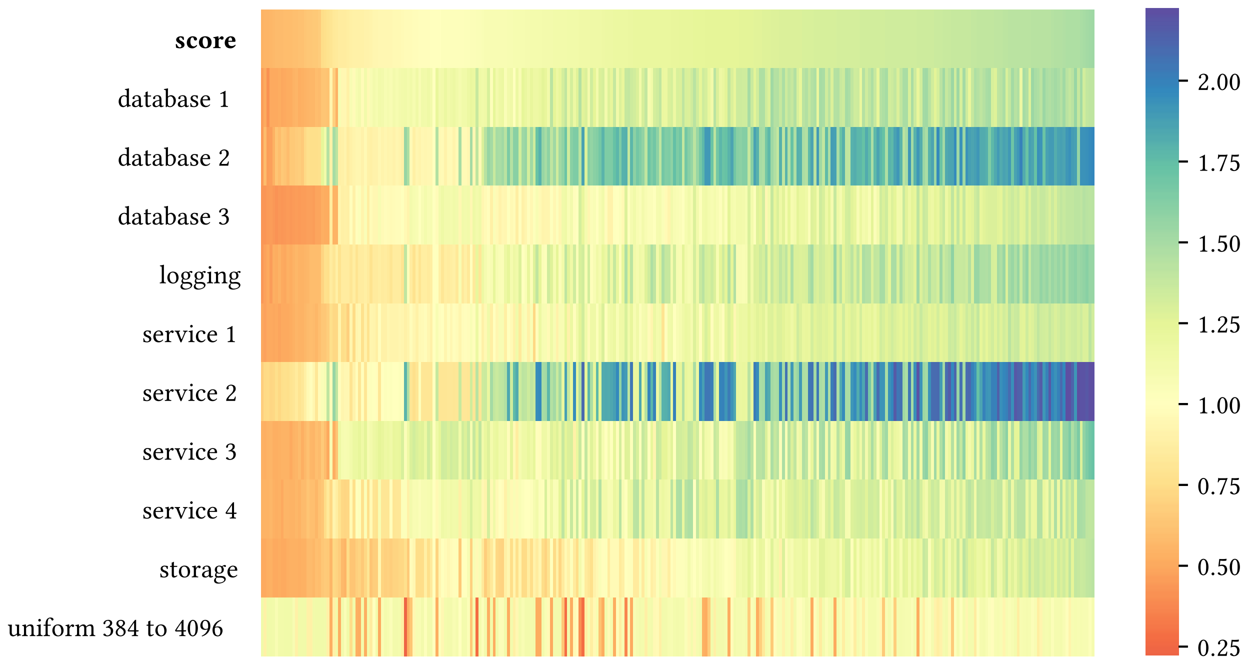


Figure 9. The speedups of the 290 generated memcpy functions for individual applications on SKX over *glibc* (*score* $\in [0.53, 1.53]$).

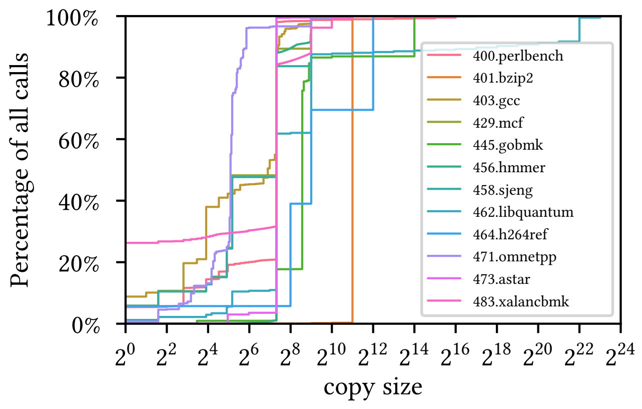


Figure 10. Cumulative distribution of memcpy sizes for each SPEC int benchmark (*ref* set). Note the log scale.

Table 7. Percentage of the time spent in memcpy for each SPEC int benchmark (*ref* dataset) using the *glibc* implementation on Haswell

benchmark	% time in memcpy
400.perlbench	1.1%
401.bzip2	0.2%
403.gcc	0.3%
429.mcf	0.0%
445.gobmk	0.3%
456.hmmer	0.0%
458.sjeng	0.0%
462.libquantum	0.0%
464.h264ref	24.3%
471.omnetpp	0.0%
473.astar	0.0%
483.xalancbmk	3.3%

section, we show how our approach can be used to derive efficient implementations for a specific workload.

The SPEC benchmark suite [20] is a widely used set of benchmarks that covers a diverse set of compute-intensive workloads from real-world applications. Figure 10 shows the distribution of memcpy sizes for each benchmark of the *ref* data set.

One important thing to notice is that even though most workloads make small copies, some of them (e.g., h264ref, libquantum) perform a significant number of large copies. Additionally, Table 7 shows that for some of them, memcpy represents a very significant portion of the total time for the benchmark. Among all these examples, 464.h264ref is an outlier as it has both a non-standard distribution of sizes, and spends a large amount of time in memcpy (about 24% of total cycles).

We focus on 464.h264ref to demonstrate the power of this techniques on an application with very specific needs.

```
if (count == 0) {
  return;  // 0
} else if (count >= 1) {
  if (count == 1) {
    return CopyBlock<1>(dst, src);  // 1
  } else if (count >= 2) {
    return CopyRepMovsb(dst, src, count);  // [    2, +inf]
  }
}
```

Listing 3. The auto-generated implementation selected by `automemcpy` for `464.h264ref`.

Table 8. Performance of the *glibc*, fleet-wide and specialized memcpy for the `464.h264ref` SPEC benchmark, on HSW. Intervals are given with 95% confidence, and the speedup is defined as the ratio between the time for *glibc* and that for the implementation (larger is better)

`memcpy` implementation	% time in `memcpy`	`memcpy` speedup	total speedup
glibc	24.35% ± 0.13	1.00	1.00
fleet-wide	25.14% ± 0.10	0.96	0.99
specialized	15.05% ± 0.15	1.80	1.11

To generate an optimized `memcpy` implementation, we collected a size profile when running the benchmark, and ran `automemcpy` on the result. The framework selected the implementation shown in Listing 3. Intuitively the framework detects, based on data, that `memcpy` sizes are either 0, 1, or very large, and so it optimizes for these two cases. The resulting code is extremely simple and small.

We ran the `464.h264ref` benchmark with the original *glibc* implementation, our fleet-wide implementation, and the specialized implementation generated by `automemcpy`. All experiments were performed on a 6-core Xeon E5 running at 3.50GHz with 16MB L3 cache. The results are summarized in Table 8. Unsurprisingly, our fleet-wide implementation is worse than the original *glibc*, because the implementation optimizes for a different size distribution than that of `464.h264ref`. However, the specialized implementation outperforms the one from *glibc* by a factor of 1.8, resulting in the benchmark's running more than 10% faster overall.

Keep in mind that the auto-generated `memcpy` implementation is only going to be efficient as long as the size distribution of the data does not significantly differ from the one on which is was trained. Therefore, such specializations must be regenerated periodically. Ideally, they should be part of the release process of the target application.

5 Limitations and Further Work

5.1 Collected Data

Since the resulting implementation depends on the collected data, it is critical that the measured `size` distribution be as representative as possible of the real run-time behavior. In the context of this paper, we used nine archetypal workloads and—as we have seen in Section 4.1—these distributions are all very similar for `memcpy`. However, it is possible that other workloads use a different `size` distribution. We plan on working on a more systematic approach to collect `size` distribution data for all workloads—weighted by their prevalence in the fleet.

5.2 Heterogeneous `size` Distributions

As seen in Section 4.5, building tailored implementations for a binary can greatly improve its global performance. Some particular workloads may use an entirely different `size` distribution, some may also have a different `size` distribution per call site. We want to explore letting the compiler automatically generate the proper implementation by itself, based on sampled run-time data. One can even envision generating multiple specialized implementations for one application, based on the call site context. They could be inlined or shared across multiple contexts via cloning.

5.3 Limitations of Benchmarking

Overall, we pick the function that maximizes the score, but a number of effects do play a role in production that are not directly measurable through benchmarks. For instance, the code size of the benchmark is smaller than the hot code of real applications and doesn't exhibit instruction cache pressure as much. When a candidate function is selected, it is crucial to also test it in production and verify that the performance gains are realized.

5.4 Framework Extensions

The framework presented in this paper can be extended in a number of ways:

1. By design, the benchmarking methodology imposes a restriction on the number of bytes to process: all data movement should stay within L1 to prevent accounting for the specifics of the memory subsystem. If bigger sizes are needed, different benchmarking methods must be used as we can no longer rely on repeating the measurement to increase SNR.
2. More Generic Access Strategies (Section 3.4) can be added if necessary. One can think of new processor capabilities like *SVE* for recent ARM processors or special instructions hinting the memory subsystem like `PREFETCHT0` or `PREFETCHNTA`.
3. The Constraint System developed in Section 3.5 can be adapted to explore other parts of the implementation space (e.g., use more discrete values, use of unaligned

loops, use of aligned loops with different values for *BlockSize* and *Alignment*, alignment on *destination* rather than *source* for aligned loop).

4. If the number of possible implementations becomes so great that a systematic exploration is no more practical, the problem can be viewed as an online optimization problem.

6 Conclusion

In this paper, we have devised a method to quickly iterate and explore the design space of memory primitives. It relies on the composition of elementary strategies activated by a decision logic written in C++. This is a departure from the "hand-crafted assembly" approach where decision logic, elementary strategies and instruction selection are all intertwined. We have demonstrated that our implementation matches the performance of hand-written assembly and ultimately provides better performance, because it is much easier to optimize and tune for a given workload. The high level implementation is also easier to test and maintain, as logical units can be tested separately.

Furthermore, we present a method to automatically generate a set of valid C++ implementations of `memcpy` and evaluate them through a specific benchmark reproducing the `size` distributions measured in production. It allows finding and deploying the functions that perform best in a particular environment. This process requires minimal human effort, which makes it easy to repeat it whenever the usage patterns of the functions change. This method is suitable for optimizing both fleet-wide deployment (Section 4.4) and particular applications (Section 4.5).

Finally, we have demonstrated that we can improve the performance of applications by taking into account the specific run-time `size` distributions of their memory primitives. In specific benchmarks (SPEC `464.h264ref`) we were able to achieve up to ten percent speedup. In particular, by relying exclusively on the manual implementation, we have improved the performance of one of the most important—and highly optimized—services at Google by $+0.65\% \pm 0.1\%$ *Requests Per Second* (RPS). We have witnessed similar gains throughout the fleet which lead to an overall 1% performance increase.

References

[1] ARM. 2020. memcpy-advsimd.S. https://github.com/ARM-software/optimized-routines/blob/master/string/aarch64/memcpy-advsimd.S

[2] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *International Symposium on Computer Architecture (ISCA)*.

[3] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris, and Richard Vuduc. 2018. Autotuning in High-Performance Computing Applications. *Proc. IEEE* 106, 11 (2018), 2068–2083. https://doi.org/10.1109/JPROC.2018.2841200

[4] JF Bastien. 2018. Deprecating volatile. http://wg21.link/P1152R0

[5] Eli Bendersky. 2011. Load-time relocation of shared libraries. https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries

[6] Eli Bendersky. 2011. Position Independent Code (PIC) in shared libraries. https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries

[7] Eli Bendersky. 2011. Position Independent Code (PIC) in shared libraries on x64. https://eli.thegreenplace.net/2011/11/11/position-independent-code-pic-in-shared-libraries-on-x64

[8] Chandler Carruth. 2015. CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!". https://youtu.be/nXaxk27zwlk?t=2441

[9] Guillaume Chatelet. 2019. [llvm-dev] [RFC][clang/llvm] Allow efficient implementation of libc's memory functions in C/C++. https://lists.llvm.org/pipermail/llvm-dev/2019-April/131973.html

[10] Guillaume Chatelet. 2020. automemcpy paper supplementary materials. http://github.com/google-research/automemcpy

[11] Guillaume Chatelet. 2020. Benchmarking llvm-libc's memory functions. https://github.com/llvm/llvm-project/blob/main/libc/benchmarks/RATIONALE.md

[12] Guillaume Chatelet. 2020. [libc-dev] Q&A and the round table highlights from the virtual dev meeting. https://lists.llvm.org/pipermail/libc-dev/2020-October/000211.html

[13] Guillaume Chatelet. 2020. Libc mem* benchmarking framework. https://github.com/llvm/llvm-project/tree/main/libc/benchmarks

[14] Guillaume Chatelet and "other contributors". 2020. Libc string functions. https://github.com/llvm/llvm-project/tree/main/libc/src/string

[15] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications. In *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*. New York, NY, USA, 12–23.

[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[17] Franz Franchetti, Yevgen Voronenko, and Markus Püschel. 2005. Formal Loop Merging for Signal Transforms. *SIGPLAN Not.* 40, 6 (June 2005), 315–326. https://doi.org/10.1145/1064978.1065048

[18] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. https://doi.org/10.1109/JPROC.2004.840301

[19] Mathias Gottschlag and Frank Bellosa. 2019. Mechanism to Mitigate AVX-Induced Frequency Reduction. *CoRR* abs/1901.04982 (2019). arXiv:1901.04982 http://arxiv.org/abs/1901.04982

[20] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[21] ISO/IEC. 2020. ISO International Standard ISO/IEC 9899:202x (E) – Programming Language C [Working draft]. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2478.pdf

[22] Svilen Kanev, Juan Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2014. Profiling a warehouse-scale computer. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[23] Donald E. Knuth. 1971. Optimum binary search trees. *Acta informatica* 1, 1 (1971), 14–25.

[24] Vlad Krasnov. 2017. On the dangers of Intel's frequency scaling. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/

[25] Michael Matz and Janothers. 2014. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0. https://gitlab.com/x86-psABIs/x86-64-ABI

[26] Carlos O'Donell. 2017. What is an indirect function (IFUNC)? https://sourceware.org/glibc/wiki/GNU_IFUNC

[27] Linus Torvalds. 2021. Linux Perf Tool. https://github.com/torvalds/linux/tree/master/tools/perf

[28] Travis Downs aka BeeOnRope and Arnaud. 2019. Enhanced REP MOVSB for memcpy. https://stackoverflow.com/a/43574756

[29] various contributors. 2021. Glibc memcpy Implementation Strategies. https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S;hb=6e02b3e9327b7dbb063958d2b124b64fcb4bbe3f#l20

[30] R. Whaley and Antoine Petitet. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience* 35 (02 2005), 101–121. https://doi.org/10.1002/spe.626

[31] Huan Ying, Hao Zhu, Donghui Wang, and Chaohuan Hou. 2013. A novel scheme to generate optimal memcpy assembly code. In *2013 IEEE Third International Conference on Information Science and Technology (ICIST)*. 594–597. https://doi.org/10.1109/ICIST.2013.6747619