

Optimized *memcpy* for 32-bit ARM Automotive Embedded Systems

1st Yucan Yu
Infocomm Technology
Singapore Institute of Technology
Singapore, Singapore
2100888@sit.singaporetech.edu.sg

2nd Yongsheng Daniel Zhou
Software and Central Technologies
Continental Automotive
Singapore, Singapore
daniel.zhou@continental.com

3rd Ying Taat Soh
Software and Central Technologies
Continental Automotive
Singapore, Singapore
yingtaat.soh@continental.com

4th Muhamed Fauzi Bin Abbas
Infocomm Technology
Singapore Institute of Technology
Singapore, Singapore
fauzi.abbas@singaporetech.edu.sg

Abstract—Efficient memory operations are critical for real-time performance in automotive embedded systems. The *memcpy* function, widely used in communication and graphical processing tasks, often falls short of meeting the low latency requirements when using the standard C library.

This paper introduces an optimized *memcpy* algorithm for 32-bit ARM microcontrollers, leveraging assembly-level enhancements to maximize microcontroller register usage and integrate advanced techniques such as byte-shifting and cache prefetching. Context-aware optimizations tailored for automotive applications improve aligned and unaligned data transfers across various memory types, achieving up to an eightfold increase in bandwidth.

Experimental results highlight significant performance improvements, especially in unaligned memory access scenarios, positioning the optimized *memcpy* as a robust and efficient solution for automotive embedded systems.

Keywords—*memcpy*, optimization, ARM, cache, register, alignment

I. INTRODUCTION

In the automotive sector, handling data efficiently is crucial as advanced systems like telematics need strong data handling and transfer methods. The *memcpy* function is commonly used to transfer data for communication [1] and graphics tasks.

However, the standard *memcpy* might not meet the low latency needs of automotive applications that typically run on real-time operating systems [2]. It often lacks optimization [3] for unaligned memory accesses, which happen when data is accessed from an address not a multiple of its size. This issue is common in the automotive field for various reasons, including compression algorithms like LZ4 [4] that may involve random combinations of aligned and unaligned accesses, on top of other operational complexities.

Investigations into the standard *memcpy* reveal significant idle time due to cache misses, indicating opportunities for optimization to enhance performance and memory efficiency.

This paper presents an optimized *memcpy* for 32-bit ARM MCUs used on automotive systems with little-endian byte ordering, utilizing the Greenhills toolchain during the software development process.

II. EXISTING SOLUTION

This section focuses on the standard implementation of the *memcpy* algorithm, as the subsections examine its strengths,

potential limitations, as well as evaluate whether this existing implementation addresses current challenges effectively.

A. Standard *memcpy*

The standard *memcpy* implementation, written in ARM assembly [5], efficiently handles larger volumes of data in four-byte chunks when the source address, destination address, and length are all aligned to 4 bytes. Figure 1 illustrates how this optimization (is compiled to) leverage single-register load/store instructions, such as LDR and STR, to improve performance. However, if any of these parameters are unaligned, the standard algorithm falls back to a less efficient, byte-by-byte transfer approach.

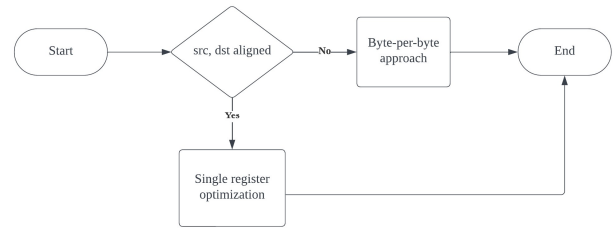


Fig. 1. Block diagram for standard *memcpy*



Fig. 2. Observing idle gaps on standard *memcpy*

Figure 2 shows further analysis on an oscilloscope, revealing a visible gap after every 32 bytes of data copied. This gap indicates substantial idle time caused by cache misses when the CPU retrieves data from the source and transfers it to the destination, a problem also present in

existing GNU LIBC implementations [6]. This observation highlights potential opportunities for optimizing the *memcpy* operation to minimize these idle periods and enhance the algorithm's performance and overall memory efficiency.

As observed, the oscilloscope measurements reveal significant idle periods due to cache miss-induced latency, evidenced by gaps after every 32 bytes of data transfer. Optimization efforts are constrained by the utilization of a single register for bulk data movement despite the availability of additional unbanked registers that could enhance throughput. Moreover, the algorithm reverts to a byte-by-byte transfer mode when the source or destination addresses are unaligned, leading to suboptimal performance. These observations underscore the need for advanced optimization strategies to address these performance bottlenecks of current *memcpy* implementations on ARM-based systems.

III. MEMORY OPTIMIZATION TECHNIQUES

Based on the above findings, several memory optimization techniques have been identified as potential solutions for addressing the identified performance bottlenecks and inefficiencies.

A. Cache Prefetching

The ARM instruction set includes the PLD (Preload Data) instruction, which allows the processor to asynchronously prefetch data from the main memory into the data cache. This prefetching reduces latency by ensuring that data is readily available in the cache when needed, mitigating the delays associated with accessing slower main memory. Furthermore, the PLD instruction functions as an NOP (no operation) on non-cacheable memory without adverse effects on such systems [7, Chapter A3.8.4]. By leveraging the PLD instruction, it is possible to reduce idle time and prepare data in advance, thereby addressing the observable gaps in the standard *memcpy* implementation as detected via oscilloscope analysis.

B. Multiple Register Access Per Instruction

A key feature of the ARM architecture is its capability to perform multiple register accesses within a single instruction, exemplified by the Load Multiple (LDM) and Store Multiple (STM) instructions [7, Chapter A4.7]. This functionality is particularly advantageous in scenarios requiring the rapid transfer of large data blocks between memory locations, as it significantly reduces the overhead associated with repetitive load/store operations. Leveraging LDM and STM instructions can enhance the efficiency of *memcpy* to optimize its overall data transfer performance.

C. Data Alignment

Memory alignment is critical for system performance and reliability, as unaligned access generally incur cycle penalties [8, Chapter 3.3.2] and can lead to hardware faults. While some processors, such as the Cortex-M7, can handle more efficient accesses to specific unaligned memory locations using instructions like LDR and STR [9, Chapter 3.3.5], others enforce stricter alignment requirements. For example, the Cortex-M0+ does not support unaligned access, resulting in a HardFault if attempted [10, Chapter 3.3.4]. Even within the Cortex-M7 architecture, not all memory regions support unaligned accesses [9, Chapter 3.3.5]. Since memory addresses tend to become aligned after a few bytes of copying, maximizing such aligned accesses through innovative

manipulations can significantly enhance the performance of *memcpy* operations.

IV. PROPOSED SOLUTION

The proposed algorithm design for the optimized *memcpy* aims to boost performance in data copying operations by leveraging identified memory optimization techniques. These techniques enhance the efficiency of both aligned and unaligned memory copies, thereby improving overall system performance.

There are three primary use cases for executing *memcpy* in a 32-bit MCU architecture, excluding the alignment of the copy length as an optimization criterion. The processor instead focuses on the alignment of the source and destination addresses as shown in Figure 3.

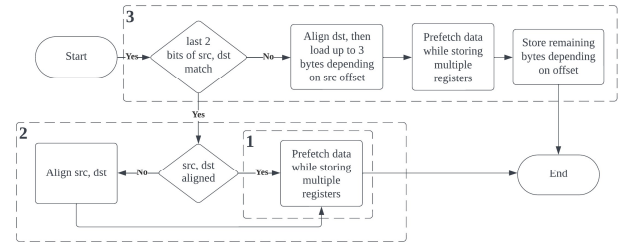


Fig. 3. Block diagram for optimized *memcpy*

A. Copy between aligned source and destination

With data alignment achieved for source and destination addresses, advanced techniques such as cache prefetching and multiple register access per instruction can be utilized without incurring additional cycle penalties. Data from the source address can be prefetched asynchronously into the data cache several cache lines ahead while the processor concurrently writes any current bytes to main memory via the store buffer when STM instructions have been executed. By preloading data into the cache, the processor can load large chunks of anticipated data into registers from the faster cache memory rather than accessing the slower main memory when load multiple instructions are invoked. This is illustrated in Figure 3, within box 1.

B. Copy between unaligned source and/or destination

In a 32-bit MCU, alignment of both source and destination addresses can be simultaneously achieved if their final 2 bits are identical after copying a certain number of bytes. For example, when the source and destination addresses are initially located at 0x9 and 0x49, respectively, after copying 3 bytes of data, both addresses will become aligned at 0xC and 0x4C. Once alignment is attained, the optimized algorithm can transition to the code path designated for copying between aligned source and destination addresses, thereby leveraging the existing optimizations detailed in the previous subsection. This is illustrated in Figure 3, within box 2.

In contrary, the source and destination addresses cannot be aligned if their final 2 bits are different, regardless of the amount of data copied. Therefore, alignment techniques for the source and destination addresses must be considered and performed separately. A byte-by-byte approach is utilized for the destination address until it becomes aligned to 4 bytes. Once this alignment is achieved, the process narrows down to three possible scenarios, where the offset required to align the source address is either 1, 2, or 3 bytes. This offset can be

loaded and stored in a spare register to achieve alignment for the source address. At this point, the source and destination addresses will be aligned, enabling optimization techniques such as multiple register access per instruction and cache prefetching without incurring cycle penalties.

However, advanced byte-shifting techniques are necessary to ensure accurate data copying from the source to the destination, as shown in Figure 4. For example, with an offset of 1 byte to align the source address in a little-endian system, after the first offset byte is loaded into a spare register, the next 4 bytes loaded into another register must be shifted left by 8 bits. This shift allows the merging of only 3 bytes with the first byte, ensuring the correct data is retrieved. This process is repeated for all subsequent data until a store multiple instruction is executed.

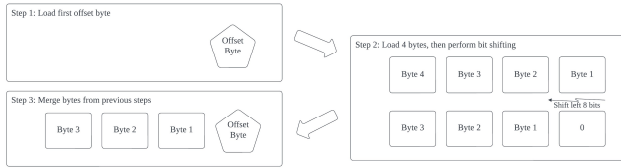


Fig. 4. Example of byte shifting

At the end of this process, any remaining offset bytes must be accurately stored back into memory, which involves ensuring that these bytes are correctly aligned and placed into the appropriate memory locations to maintain data integrity and consistency. This is illustrated in Figure 3, within box 3.

V. EXPERIMENTATION AND FINDINGS

The experiments described in this section were designed to assess its performance enhancements across various memory configurations, and were conducted at an active clock frequency of 100 MHz across three memory types: NC (non-cacheable memory), M1 (cacheable memory), and M2 (smaller cacheable memory with different protocol than M1), ranked by their processing speed. Test sizes ranged from 2KB to 20KB as the testing program encompassed five distinct types of tests: aligned, unaligned with the same final 2 bits, and offsets of 3, 2, and 1 byte (considered as unaligned with different final 2 bits). Each test type was executed ten times to ensure the statistical significance and reliability of the results.

A. Validation Steps

The performance test procedure involved initialising a source buffer with data, then flushing and invalidating the data cache associated with the test buffers to ensure results were unaffected by any residing data in the cache. Interrupts were suspended to prevent higher-priority tasks from affecting performance measurements during the memcpy operations. System timers recorded the execution time of each memcpy instance. Bandwidth results were calculated based on test size and execution time, before the results were compared between the optimized and *standard* memcpy implementations. Following the optimizations, a second round of oscilloscope analysis was also conducted to verify the effectiveness of the implementation, particularly in reducing the gaps previously observed after every 32 bytes of data copying.

B. Results and Analysis

This section presents and analyzes the outcomes of the tests conducted to evaluate the performance of the proposed optimized *memcpy* implementation, which has previously

been verified to maintain accurate data copying by checking for an expected return value of 0 using a *memcpy* after copying data from a source to its destination.

Tables I, II and III provide the percentage improvement of the optimized *memcpy* over the standard implementation for various memory transfer configurations and data sizes (2k, 4k, 8k, 16k, 20k) for aligned and unaligned (same and different final 2 bits) memory transfers.

TABLE I. IMPROVEMENTS FOR ALIGNED MEMORY TRANSFERS

Source to Destination	Data size				
	2k	4k	8k	16k	20k
M1 to M1	66%	81%	91%	81%	72%
M1 to M2	62%	69%	52%	-24%	-27%
M2 to M1	63%	66%	68%	72%	72%
M2 to M2	46%	65%	37%	4%	-3%
NC to NC	-2%	0%	-1%	-1%	1%
NC to M1	-2%	-2%	-1%	0%	-1%
NC to M2	1%	0%	0%	0%	0%
M2 to NC	178%	183%	185%	186%	186%
M1 to NC	175%	197%	212%	220%	222%

TABLE II. IMPROVEMENTS FOR UNALIGNED MEMORY TRANSFERS (SAME FINAL 2 BITS)

Source to Destination	Data size				
	2k	4k	8k	16k	20k
M1 to M1	192%	237%	257%	248%	244%
M1 to M2	189%	224%	177%	25%	30%
M2 to M1	60%	65%	67%	66%	66%
M2 to M2	58%	59%	122%	27%	14%
NC to NC	248%	340%	416%	466%	479%
NC to M1	202%	280%	337%	380%	390%
NC to M2	54%	52%	52%	47%	45%
M2 to NC	620%	639%	648%	653%	654%
M1 to NC	633%	697%	735%	757%	762%

TABLE III. IMPROVEMENTS FOR UNALIGNED MEMORY TRANSFERS (DIFFERENT FINAL 2 BITS)

Source to Destination	Data size				
	2k	4k	8k	16k	20k
M1 to M1	110%	125%	132%	129%	128%
M1 to M2	107%	123%	94%	24%	25%
M2 to M1	67%	71%	72%	66%	65%
M2 to M2	67%	68%	41%	5%	3%
NC to NC	101%	116%	126%	127%	126%
NC to M1	70%	81%	90%	95%	96%
NC to M2	52%	50%	50%	44%	43%
M2 to NC	666%	675%	697%	699%	703%
M1 to NC	516%	549%	579%	590%	594%

Generally, bandwidth increases are observed for scenarios involving cacheable source memories until the cache is exhausted for certain memory protocols, as evidenced by test sizes exceeding 8KB for M2 destinations, notably with even negative increases of -24% and -27%, observed at 16k and 20k respectively during M1 to M2, as shown in Table I.

In contrast, the most significant improvements are observed in M2 to NC and M1 to NC transfers, with gains of up to 703% and 762%, as shown in Tables III and II respectively, highlighting the efficiency of the optimized *memcpy* for cacheable to non-cacheable memory operations. While there may be negligible impact on bandwidth for non-cacheable source memories during aligned memory transfers, substantial performance improvements of up to 479%, as shown in Table II, are still observed across these non-cacheable scenarios during unaligned memory transfers.

Overall, the optimized *memcpy* offers significant enhancements in data transfer performance, particularly for unaligned memory transfers with up to eight times improvements. These gains can be achieved with a modest increase of less than half a kilobyte in code size, making the optimized implementation a viable and efficient solution for automotive embedded systems requiring high-performance data transfers.

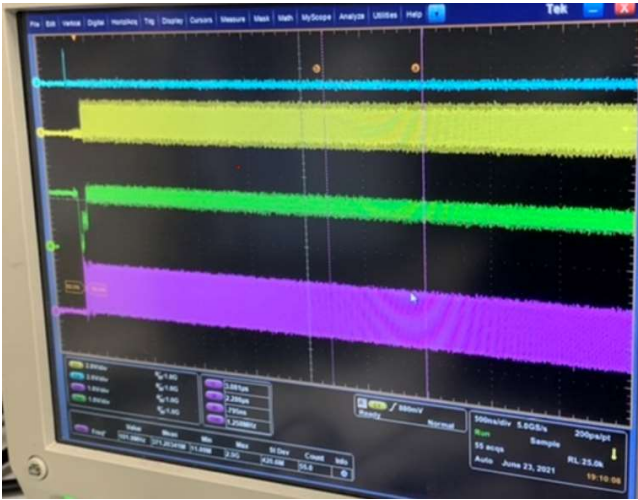


Fig. 5. Pipeline activity for optimized *memcpy* on the oscilloscope

As illustrated in Figure 5, the pipeline activity analysis validates the concurrent process of data prefetching from the memory device. At the same time, the processor accesses the previous 32 bytes from the cache. This asynchronous prefetching strategy significantly enhances overall memory performance by effectively overlapping data retrieval with processor execution, minimizing latency, and maximizing data throughput.

While further optimization may be needed for specific scenarios involving these heterogeneous memory protocols, especially for scenarios involving M2 destinations during larger data transfers, the optimized *memcpy* has demonstrated substantial performance enhancements across various data transfer use cases in automotive embedded systems with minimal downsides that effectively balances improved efficiency and resource utilization, making it a robust solution for automotive embedded systems.

VI. CONCLUSION

The optimized *memcpy* algorithm presented in this paper significantly enhances the performance of data transfer operations in 32-bit ARM microcontrollers, addressing the inefficiencies of existing *memcpy* implementations. By employing advanced memory optimization techniques such as cache prefetching, multiple register access per instruction, and precise data alignment, the proposed solution effectively

reduces idle time and improves data throughput. Experimental validation across various memory configurations and data sizes demonstrates substantial performance gains, particularly for unaligned memory transfers, with improvements reaching up to 762%. These enhancements can be achieved with a minimal increase in code size, ensuring the solution's viability in resource-constrained automotive embedded systems while meeting the stringent performance requirements of real-time automotive applications.

VII. FUTURE WORK

Future work could focus on further optimizing the algorithm for specific memory configurations and larger data transfers, especially for aligned memory transfers. Additionally, exploring the potential for hardware-level support to further enhance memory transfer efficiency could yield significant benefits. Extending these optimization techniques to other commonly used functions in automotive embedded systems could also provide comprehensive performance improvements across the board as the concepts and techniques utilized in this study are expected to be reusable across multiple CPU architectures, such as the TriCore, with only minor adaptations required for the source code.

ACKNOWLEDGMENT

We would like to express our gratitude to Continental Automotive for providing us with the opportunity to explore and work on this optimized *memcpy*. We acknowledge that the source code and memory types used in this paper are proprietary and will remain confidential, and as such, shall not be published.

REFERENCES

- [1] G. Schirmer and R. Dömer, "ABSTRACT COMMUNICATION MODELING: A Case Study Using the CAN Automotive Bus," Jan. 2005. [Online]. Available: https://www.cecs.uci.edu/~doemer/publications/TESS_05_125.pdf
- [2] L. Harvie, "RTOS Essentials: A Quickstart Guide for Embedded Engineers," Mar. 2024. [Online]. Available: <https://runtimeec.com/wp-content/uploads/2024/03/rtos-essentials-ebook.pdf>
- [3] A. Fog, "Instructions for asmlib: A multi-platform library of highly optimized functions for C and C++," May. 2022. [Online]. Available: <https://agner.org/optimize/asmlib-instructions.pdf>
- [4] Various contributors, lz4.c, 2024, GitHub Repository. [Online]. Available: <https://github.com/lz4/lz4/blob/dev/lib/lz4.c>.
- [5] G. Chatelet, C. Kennelly, S. (L.) Xi, O. Sykora, C. Courbet, X. D. Li, and B. D. Backer, "automemcpy: A Framework for Automatic Generation of Fundamental Memory Operations," in Proc. 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), Virtual, Canada, June 22, 2021. doi: <https://doi.org/10.1145/3459898.3463904>.
- [6] Various contributors, "25131 – memcpy performance problem with ARM 32 A9be due to high cache-misses," 2019. [Online]. Available: https://sourceware.org/bugzilla/show_bug.cgi?id=25131
- [7] ARM, ARMv7-M Architecture Reference Manual (Revision r1p0), 2005.
- [8] ARM, ARM Cortex-M4 Technical Reference Manual (Revision r0p0), 2010.
- [9] ARM, ARM Cortex-M7 Devices Generic User Guide (Revision r1p2), 2015.
- [10] ARM, ARM Cortex-M0+ Devices Generic User Guide (Revision r0p1), 2012.