An Efficient Memory Management Method for Embedded Vector Processors

Shengxuan Li[1], Haoqi Ren[1], Zhifeng Zhang[1], Bin Tan[2], and Jun Wu[3(✉)]

[1] Department of Computer Science, Tongji University, Shanghai, China
{lsx1998,renhaoqi,zhangzf}@tongji.edu.cn
[2] School of Electronic and Information Engineering, Jinggangshan University, Ji'an, China
tanbin@jgsu.edu.cn
[3] School of Computer Science, Fudan University, Shanghai, China
wujun@fudan.edu.cn

Abstract. For processors with vectorial computing units like DSP, it is very important to ensure vector load/store operations alignment of memory blocks, and minimize space wastage when making memory allocations. In this paper, we design and implement a memory management method, vector memory pool, suitable for embedded vector processors. By partitioning an entire block of memory space into many aligned vector objects and making efficiently use of vector processing units, the processing of memory manipulation library functions such as memset/memcpy is accelerated. The implementation and comparative verification of vector memory pool on RT-Thread Nano based on SWIFT DSP was completed, and the running efficiency reached a tens of times improvement compared to the original method.

Keywords: DSP · RT-Thread Nano · vector memory pool

1 Introduction

DSP is an embedded processor that is very good at realizing various digital signal processing operations (such as digital filtering, spectrum analysis, etc.) at high speed. The strengths of high-performance DSP are their ability to perform vector operations, pointer linear addressing, and other data processing with large amounts of operations. Generally, DSP chips are mostly used in embedded scenarios, but they also have the need to support multitasking. In order to realize multitasking scheduling and memory management functions on embedded devices while ensuring the real-time performance of embedded DSP chips, the porting and targeted optimization of real-time operating systems on embedded devices are especially important [1].

A real-time operating system is an operating system that can support the operation of real-time control systems. Compared to improving the efficiency of using computer systems, the primary task of real-time operating systems is to schedule all available resources to accomplish real-time control tasks as much as possible. Real-time operating

F. Gao et al. (Eds.): ChinaCom 2022, LNICST 500, pp. 247–258, 2023.
https://doi.org/10.1007/978-3-031-34790-0_19

system programs are small, task switching is fast, and interrupts are blocked for a very short time [2].

The resource environment of embedded systems is also various. Some systems are resource constrained with only tens of KB of memory available for allocation, while others have several MB of memory, and it becomes complicated to choose an efficient memory allocation algorithm for these different systems that is suitable for them.

In this paper, we design a new static memory management method, vector memory pool, for a real-time operating system, RT-Thread Nano, running on SWIFT DSP developed by the CIC lab of Tongji University, and verify its correctness and performance.

2 Relate Work

2.1 RT-Thread Nano

RT-Thread is a free open source embedded real-time operating system platform created by a group of open-source enthusiasts in China, mainly for small and medium-sized microcontrollers, using the GPLv2 license, which can be applied to commercial projects for free and does not require the project to be open source [3–5]. RT-Thread Nano is a minimalist version of the hard real-time kernel, developed in C language, using object-oriented programming The RT-Thread Nano is a cuttable, preemptive real-time multitasking RTOS developed in C with an object-oriented programming mindset.

The RT-Thread operating system provides different memory allocation management algorithms in a targeted manner, depending on the upper layer applications and system resources. In general, they can be divided into two categories: memory heap management and memory pool management. And memory heap management is further divided into three cases according to specific memory devices: the first is for allocation management of small memory blocks (small memory management algorithm); the second is for allocation management of large memory blocks (slab management algorithm); and the third is for allocation cases of multiple memory heaps (memheap management algorithm).

The memory heap manager can allocate memory blocks of arbitrary size, which is very flexible and convenient. However, it also has obvious drawbacks: first, it is not efficient in allocation, and free memory blocks have to be looked up at each allocation; second, it is easy to generate memory fragmentation. To improve the efficiency of memory allocation and avoid memory fragmentation, RT-Thread provides an alternative memory management method: Memory Pool [6–9].

Memory pool is a memory allocation method used to allocate a large number of small memory blocks of the same size, which can greatly speed up memory allocation and release and avoid memory fragmentation as much as possible. In addition, RT-Thread's memory pool supports thread hang function, when there is no free memory block in the memory pool, the application thread would be hung until there is a new available memory block in the memory pool, and then the hung application thread would be woken up.

The memory pool's thread pending feature is ideal for scenarios that require synchronization via memory resources, such as when playing music, the player thread decodes

the music file and sends it to the sound card driver, which drives the hardware to play the music.

The following is a description of the static memory pool implementation.

A memory pool is created by first requesting a large block of memory from the system, and then dividing it into multiple smaller blocks of the same size, which are directly connected by a link list (this link list is also called free block link list). At each allocation, the first memory block at the head of the chain is taken from the free link list and provided to the requestor. As you can see in the Fig. 1, there are multiple memory pools of different sizes allowed in physical memory, each of which in turn consists of multiple free memory blocks, which the kernel uses for memory management. When a memory pool object is created, the memory pool object is assigned to a memory pool control block whose parameters include the pool name, memory buffer, memory block size, number of blocks, and a queue of waiting threads [10–12].

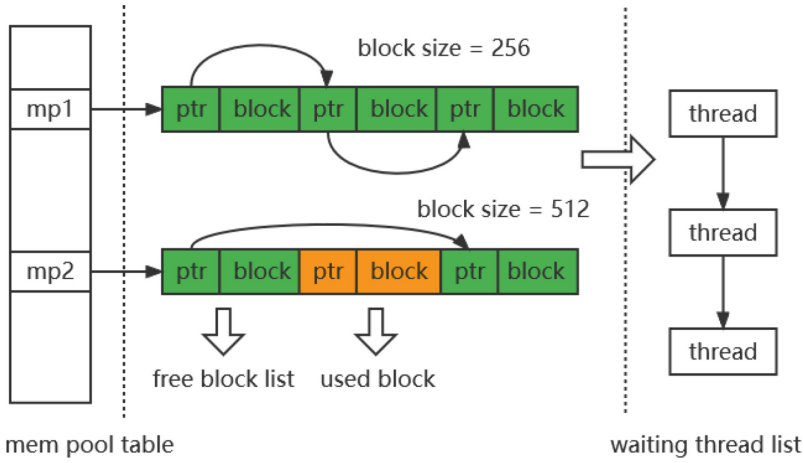

Fig. 1. The design of a typical memory pool in RT-Thread.

2.2 SWIFT DSP

Swift is a SIMD VLIW DSP chip, from hardware architecture to assembly instruction set are developed by CIC lab of Tongji University, is an 8-launch, 13-stage pipeline very long instruction word DSP processor, while the on-chip by a set of bit width of 2560bit vector registers, and for the vector registers designed a variety of memory load/store and calculation instructions. It also provides a complete tool chain, including LLVM-based compiler, assembler, linker, and functional/structural simulator, and supports GDB debugging and FPGA test verification [13–15].

The CIC lab of Tongji University has previously implemented RT-Thread support for SWIFT DSP and completed the porting and correctness testing of RT-Thread Nano on DSP. Now, we are considering the optimization of its memory management performance, mainly considering the efficient use of memory and vector optimization of some system library functions like memset by using the existing vector register resources on chip.

3 Design and Implementation

3.1 Vector Memory Pool

For embedded processors like DSPs, there is no memory mapping and all applications access the real physical address. Memory pools are suitable for this class of systems without MMU. 0 address corresponds to the real memory starting address (so it must also be accessible).

For processors with vector registers, the load/store instruction is usually designed separately for vector processors, and the access address alignment of such instructions is strongly related to the width of the vector, while the general-purpose memory pool design usually does not give special consideration to vector load/store, and the adoption of the previously described memory pool design leads to the following phenomenon.

Although many memory pool designs are designed with multiple block size structures of different bit widths, they are handled by linking a pointer and a memory block together and linking the memory blocks into a free block chain through the pointer, which leads to the fact that the addresses of the memory blocks are not aligned as required, and a large amount of space is often wasted for vector structure alignment in order to meet the needs of vector object read and write usage. As we can see in the Fig. 2.

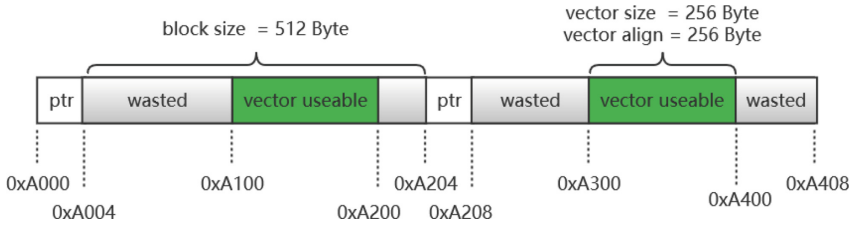

Fig. 2. The space wasted in memory pool for vector align.

For the special case of vector registers mentioned above, this paper designs a new vector memory pool that splits the memory block from the original chain structure so that each memory block is guaranteed to be address aligned, adds a pointer to the memory block in the original chain architecture, and records the pointer from the memory block to the original structure through an external hash table. The specific logic structure is as follows Fig. 3.

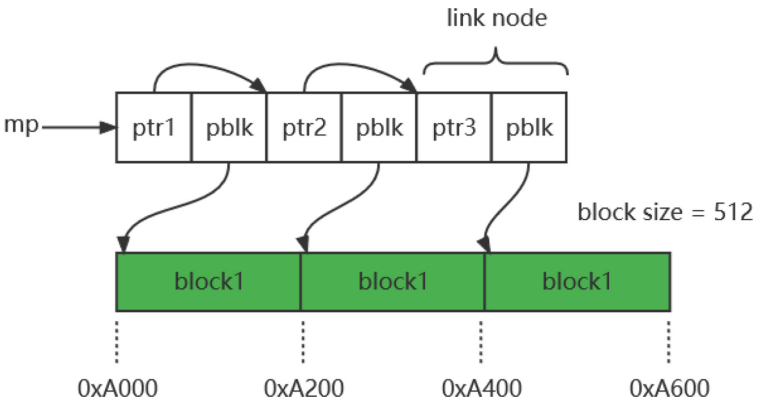

Fig. 3. The logic design of vector memory pool.

3.2 Design Details

Top-Level Design

At the top level, a mptable is created to record the mp structures according to different vector sizes, each mp structure points to the whole memory segment allocated to it, and the memory blocks are aligned according to the vector width, and each memory segment is divided into blocks according to its block size, and a free block linklist is created. The specific structure is shown in the Fig. 4.

The mp structure records the memory space and size allocated to this thread pool, as well as the size of the memory blocks divided, the total number of free blocks and the chain of free blocks, and the chain of threads currently requesting memory from this mp and the number.

Initialization

Get a whole block of memory and complete vectorization alignment, divide it into multiple block blocks according to the desired size, generally, the size of the block could be much larger than the number of blocks divided, so it is only necessary to simply put the first $\lceil 8*n/block\_size + 1 \rceil$ blocks used to save the linklist structure of the head area, give each block used for memory allocation number 0-n, create an array of linked node structures of size n in the head area, each pointer structure and memory block one by one, ptr points to the next node structure, pblk points to the memory block corresponding to the current structure, and also create a hash table for block addressing.

Finally, the free block linklist is saved in the vector memory pool structure to complete the initialization.

Allocation and Free

The process of allocating a vector memory pool is relatively simple. After finding a free block, it is plucked from the chain table and the pointer is pointed to the original mp structure, which is not much different from the general memory pool allocation.

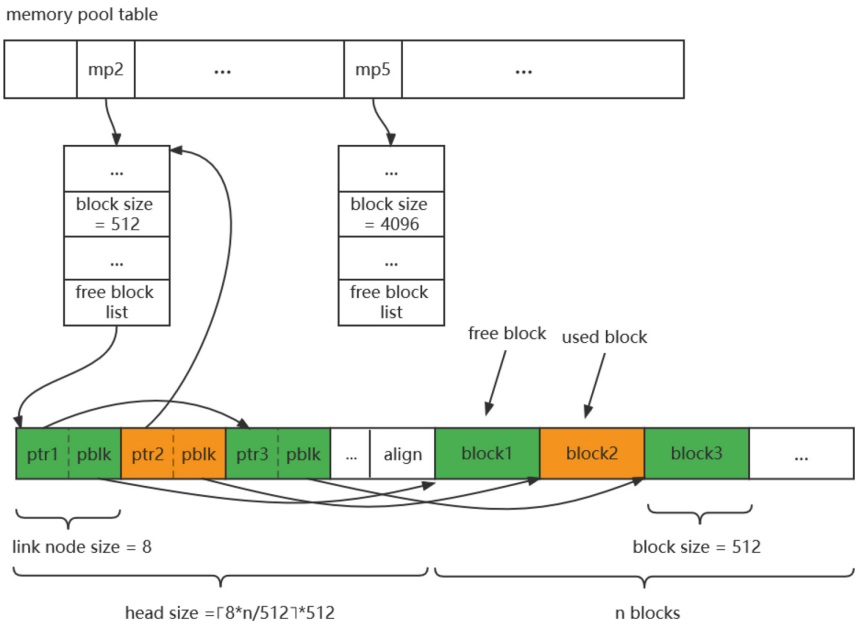

Fig. 4. The top-level design of vector memory pool.

The memory free process, because of the separation of memory blocks and pointers, cannot directly find the mp structure to which the current memory block belongs as in a regular memory pool implementation, so extra space and time are needed to complete the process by finding the corresponding structure of the memory block through an external hash table mapping and inserting the memory block back into the free list, using the header insertion method.

Figure 5 shows an example of a vector memory pool usage, where memory block 1 is first requested and then block 4 is freed.

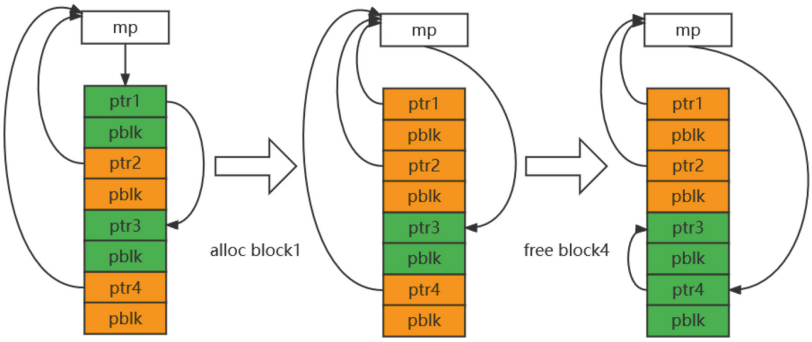

Fig. 5. An example of vector memory pool.

Memory Library Vectorization

Since the space arrangement of the vector memory pool is aligned according to the vector load/store format, we can use vector registers to accelerate some memory operations, typically such as memset/memcpy and other operations.

For the space allocated using vector memory pool, if we want to do batch memory read/write, by using vector registers, we can get a theoretical 64 times efficiency improvement, as the vector size is 2560 bits which can be considered as 64 cells of 40 bits. Here is a detailed example comparison of memcpy in Fig. 6, for larger memory copy, if we don't use vector registers, we need to loop n times, if we use vector registers + normal memory pool, we need to use additional scalar copy to handle memory space close to a vector size, if we use vector registers + vector memory pool method, we can maximize the efficiency of batch copy.

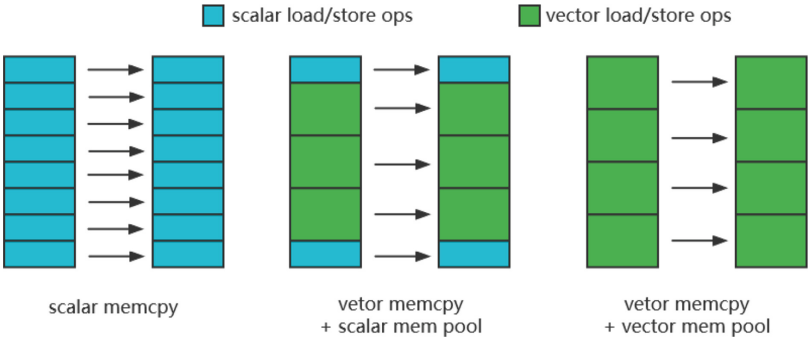

Fig. 6. An example comparison of memcpy.

Since vector registers also support the special instruction called vmovg2v, it is possible to rewrite individual scalar values to each group of the vector, allowing efficient optimization for operations such as memset. By reusing hardware loop and step read/write instructions, the efficiency of mem operations can be further improved.

3.3 Implementation

RT-thread internally encapsulates the memory pool call interface, so we can keep the interface consistent by adding our designed vector memory pool to the original base memory allocation method of RT-Thread, which could simplify our work on other unrelated aspects, allowing us to focus on the vector memory pool implementation, and enabling us to compare the vector memory pool with the existing methods, which is beneficial for performance analysis.

The SWIFT DSP is a high-performance vector processor with a 256 bytes wide data bus and up to 2560 bits vector arithmetic units. CIC lab has equipped it with a complete and efficient tool chain and completed the port of RT-Thread Nano, so the

implementation and verification of our designed vector memory pool can be done using RT-Thread Nano running on the SWIFT DSP.

Initialization

To create a vector memory pool, refer to the standard memory pool interface provided by RT-Thread, we design the following static vector memory pool initialization interface. rt_err_t rt_vmp_init(rt_mp_t mp, void *start, rt_size_t size, rt_size_t block_size);

Initialize the vector memory pool, first pass in the relevant data via parameters, including the target address used to allocate space, the address needs to be externally aligned to the actual vector situation before passing in, the size of the space, and the size of the memory block, after which the actual number of available memory blocks is calculated based on the two sizes, while the relevant counters are configured to set all blocks to idle and the number of waiting threads to 0.

After that, split the head and blocks according to the calculated number of blocks, map the head area into an array of link nodes, initialize the link nodes and free memory blocks, establish a one-to-one correspondence, and create a hash table to save the mapping of memory blocks to nodes, after that, return the established vmp structure to complete the initialization.

Allocation and Free

Similarly, refer to the original interface of RT-Thread and implement void *rt_vmp_alloc (rt_mp_t mp, rt_int32_t time); for requesting memory, where the meaning of the time parameter is the timeout for requesting the allocation of a memory block. If there are free memory blocks in the memory pool, the first memory block is taken from the free block chain table of the memory pool, the number of free blocks is reduced and this memory block is returned; if there are no more free memory blocks in the memory pool, the timeout time setting is judged: if the timeout time is set to zero, the empty memory block is returned immediately; if the waiting time is greater than zero, the current thread is hung on this memory pool object until there are free memory block available in the memory pool, or until the wait time is reached.

For memory freeing, implement void rt_vmp_free (void *block); interface. When using this function interface, first find out the connection node corresponding to the memory block that needs to be freed through the hash table, calculate the memory pool object that the memory block is in (or belongs to), then increase the number of available memory blocks in the memory pool object, and add the freed memory block to the free memory block chain table. Then determine if there is a pending thread on that memory pool object, and if so, wake up the first thread on the pending thread chain table.

Memory Library Vectorization

For mem-related operations, according to the original interface, implement void *rt_vmemset(void *src, int c, rt_ubase_t n); and void *rt_vmemcpy(void *dest, const void *src, rt_ubase_t n); this two mem library functions, mixed in c and assembly language, call special vector operation instructions to complete the vector mem operation. The core operators of these two memory operations are vmovrg2v10 and vstore10.

4 Evaluation

By comparing with RT-Thread Nano's existing memory pool and mem operation library, we can analyze the operational efficiency of the static vector memory pool we designed.

We use the RT-Thread Nano running on the SWIFT DSP structure simulator to perform the test. This structure simulator is able to get the exact running time of the program on the DSP, which can be used for our experimental results verification and comparative analysis. And the simulator could record and output the number of cycles the program runs when it stops at a breakpoint, so that we can calculate the running time of each library function by adding breakpoints before and after the target code segment to be tested. The test case is a simple multi-threaded code, configuring the vector bit width used by the DSP to 2048 bits, designing two threads with the same priority, and the entire execution of each thread is much smaller than the size of a time slice allocated by the operating system, so there is no problem with the impact of task scheduling on the clock cycle. One thread use the Rt-Thread memory pool and mem library functions, and the other use vector memory pool and vector mem library, and each operation is looped 10 times to calculate the average number of running cycles.

We analyze the operational efficiency by comparing the number of the program running clock cycles, including the initialization /allocation and free of the memory pool, and focusing on the efficiency of mem-related operations. As can be seen in Figs. 7 and 8, the vector memory pool does not have a particularly large additional overhead in establishing allocation and recovery compared to the standard implementation, except for the free operation, which requires an additional time overhead of reverse lookup.

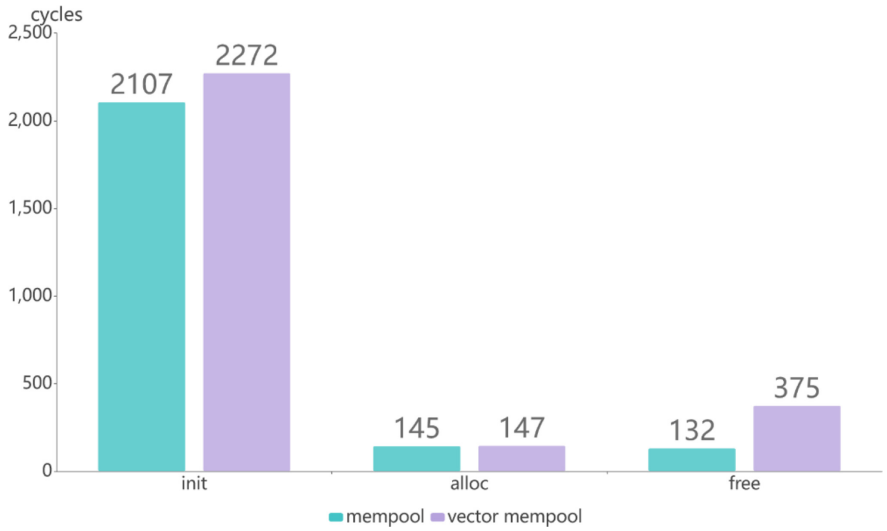

Fig. 7. The evaluation results of vector memory pool

We can see that by using vector arithmetic units, both memset and memcpy operations have a large performance improvement, and the magnitude of the improvement varies

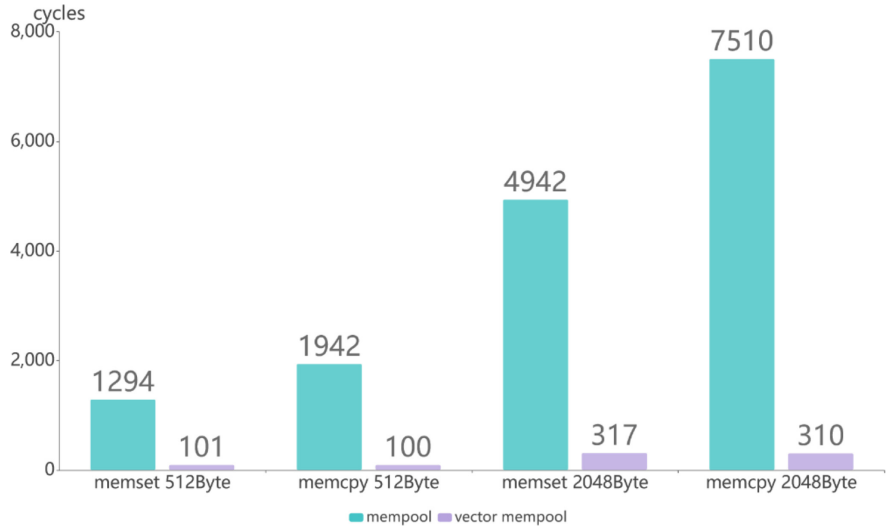

Fig. 8. The evaluation results of vector memory library

according to the bit width size. The performance improvement is 12.8 times, and at 2048 bits, the performance improvement is 15.6 times, the efficiency improvement of memcpy also reaches 19.42 times and 24.22 times.

Analyzing the above results, first, during the initialization phase of the memory pool, the vector memory pool needs to build a block reverse lookup table at the same time, and this operation incurs some overhead, but since the init operation itself requires a large number of clock cycles, the slight increase does not affect the usage. For the memory allocation operation, both memory pools require an operation to get the block address, and the number of cycles spent here is almost the same, so it can be considered to have no effect. As for the memory free operation, since the vector memory pool requires a reverse lookup, the time overhead in this area is relatively large, and this is an area that can be optimized in the future. By introducing methods like bitmap, the time complexity of the lookup can be adjusted to O(1), and the performance loss of the free operation can be accepted due to the huge performance improvement in the mem operation, for example, a common scenario where a block of memory is acquired and initialized using malloc and freed after use, where this three memory library functions, alloc + memset + free, are used, and the huge performance gain from memset can completely cover the loss of free. In fact, the performance of the vector memory pool in such a scenario is 6.22 times higher than before.

For vector mem functions, the performance improvement of the vector memory pool is more significant and, as the block bit width size increases, the performance improvement becomes more pronounced. This is because, as the memory block size increases, the percentage of call overhead generated by the system gradually decreases and the efficiency of vector utilization becomes more obvious.

5 Conclusion

In this paper, we design and implement a memory management method, vector memory pool, suitable for use in embedded vector processors. By dividing the contiguous address space according to the vector bit-width alignment, it solves the space wastage problem in memory allocation for processors like DSP that need to use a lot of vector load/store, and by using vector units, the processing of memory manipulation library functions such as memset/memcpy is accelerated. The implementation of vector memory pooling and comparison verification was done on rt-thread nano running on SWIFT DSP. Compared with the original method, there is only a slight loss of efficiency in the creation of memory pools for allocation and free, but it is able to ensure that the allocated space has been aligned according to the vector load/store requirements, and to achieve a tens of times improvement in the operational efficiency of mem operations.

Acknowledgement. The authors thank the editors and the anonymous reviewers for their invaluable comments to help to improve the quality of this paper. This work was supported by National Key R&D Program of China under Grant 2020YFA0711400, National Natural Science Foundation of China under Grants 61831018 and U21A20452, the Outstanding youth project of Natural Science Foundation of Jiangxi Province 20212ACB212001, and the Jiangxi Double Thousand Plan under Grant jxsq2019201125.

References

1. Shang, Q., Liu, W.: Multi-function DSP experimental system based on TMS320VC5509. In: Proceedings of 2016 2nd International Conference on Social, Education and Management Engineering (SEME 2016), pp. 107–111. DEStech Publications (2016)
2. Tarasiuk, T., Szweda, M.: DSP instrument for transient monitoring. Comput. Stand. Interfaces **33**(2) (2010)
3. Shen, J.Q., Wu, J., Zhang, Z.F., et al.: Design and implementation of binaryutilities generator. Appl. Mech. Mater. **644**, 3260–3265 (2014). Trans Tech Publications Ltd.
4. Fridman, J., Greenfield, Z.: The TigerSHARC DSP architecture. IEEE Micro **20**, 66–76 (January 2000)
5. Zhou, Y., He, F., Hou, N., Qiu, Y.: Parallel ant colony optimization on multi-core SIMD CPUs. Future Gener. Comput. Syst. **79** (2018)
6. Maiyuran, S., Garg, V., Abdallah, M.A., et al.: Memory access latency hiding with hint buffer: U.S. Patent 6,718,440, 6 April 2004
7. Adachi, Y., Kumano, T., Ogino, K.: Intermediate representation for stiff virtual objects. In: Proceedings Virtual Reality Annual International Symposium 1995, pp. 203–210. IEEE (1995)
8. Vanholder, H.: Efficient Inference with TensorRT (2016)
9. Chadha, P., Siddagangaiah, T.: Performance analysis of accelerated linear algebra compiler for TensorFlow
10. Sivalingam, K., Mujkanovic, N.: Graph compilers for AI training and inference
11. Griewank, A., Walther, A.: Evaluating derivatives: principles and techniques of algorithmic differentiation. SIAM (2008)
12. Paszke, A., Gross, S., Massa, F., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, pp. 8024–8035 (2019)

13. Moore, R.C., Lewis, W.: Intelligent selection of language model training data. In: Proceedings of the ACL 2010 Conference Short Papers, pp. 220–224. Association for Computational Linguistics (2010)
14. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 265–283 (2016)
15. Yang, Y., Wu, R., Zhang, L., Zhou, D.: An asynchronous adaptive priority round-robin arbiter based on four-phase dual-rail protocol. Chin. J. Elec. **24**(01), 1–7 (2015)