Integrated Dynamic Memory Manager for a RISC-V Processor

Chun-Jen Tsai[†], Chun Wei Chao, and Sheng-Di Hong
Department of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan
†cjtsai@cs.nycu.edu.tw

***Abstract*—In this paper, we present an open-source RISC-V processor with an integrated dynamic memory manager hardware module. Traditionally, the management of the main memory of a computing system is handled by a software library. However, the process involves searching and manipulation of the link lists of memory blocks, which can be expensive when the memory becomes fragmented. As a result, for embedded systems that have to be online for a long duration, a static data structure is often used to reduce the overhead of dynamic memory management at the cost of less software flexibility. Nevertheless, modern VLSI technology allows the efficient implementations of hardwired resource managers directly into the processor microarchitecture for better performance. As the experiments in this paper show, a hardware memory manager integrated within the processor core can be much more efficient than using a software library. Hardwired resource managers are particularly useful for IOT devices since the processors typically run at a lower clock rate. The proposed architecture is implemented and verified on a Xilinx FPGA development board and will be made open source.**

Keywords—Memory manager hardware; RISC-V processor design; application-specific ISA; processor architecture; FPGA

I. Introduction

A modern computing system often runs multiple computation units that share a common main memory area. For hardware-software co-designed systems, these computation units will be a mixture of hardware logics and software applications. Traditionally, the runtime management of the memory blocks for each application is usually handled by the system library software of the OS executed by the general-purpose processor cores. There are several issues of this approach. First of all, a hardware accelerator must request a dynamic memory block through the processor, which increase the overhead of a co-designed system. Secondly, when the main memory area becomes fragmented after a series of allocations and deallocations, the time it takes to search for a free memory block of certain size in the list of the free memory blocks would become expensive [1]. In addition, the merging of the consecutive free memory blocks into a larger block, or the collection of garbage blocks required by a dynamically typed object-oriented programming languages, such as Java or Python, can also be very expensive and cause an application to stall at runtime. Therefore, the idea of hardware-based memory management has been considered even in the early days of computer design [2].

One way to fix the problem of memory fragmentation is to use the MMU of a CPU to map noncontiguous physical memory blocks to a contiguous virtual address space [4]. However, this is not possible for processing cores (e.g., DSP, GPU, and accelerator IPs) that do not share the same MMU or for deeply embedded systems without MMUs. Also, it has extra overhead since the allocation unit has to match the page size.

In this paper, we present the possibility to use a hardwired Dynamic Memory Manager (DMM) unit to replace the traditional software memory manager for the processor cores. A hardware memory manager can expose interfaces for both the processor core and the hardware accelerator IPs to share a common dynamic memory manager for the main memory. However, in this paper, the proposed system focuses on the processor interface of the dynamic memory manager alone. For future work, we will integrate a flexible Direct Cache Access (DCA) unit similar to the system proposed in [3]. This way, the DMM unit can be shared by the processor and the hardware accelerators efficiently.

There are two expensive operations in a DMM unit: the search for a free memory block from the link lists maintained by the memory manager and the merging and purging of memory blocks in the lists after freeing a memory block. Most of the schemes for speeding up these tasks are designed for the software-based memory allocators. For example, to reduce the overhead of free-space searching, a binary buddy system is commonly used [5]. In the binary buddy systems, memory blocks are split and merged following a binary tree structure where each leaf represents a memory block. Two blocks from the split of a free block are buddies. If both buddies are free, they will be merged to form a block doubling their size. There are techniques that allow fast search of a best-fit free block in a buddy system. However, a buddy system requires more memory space to maintain the data structure.

There are many advantages for a hardware memory manager [6]. For example, for the allocation operations, the search for a free memory block can be done in parallel among all memory lists. Secondly, the memory defragmentation or the garbage

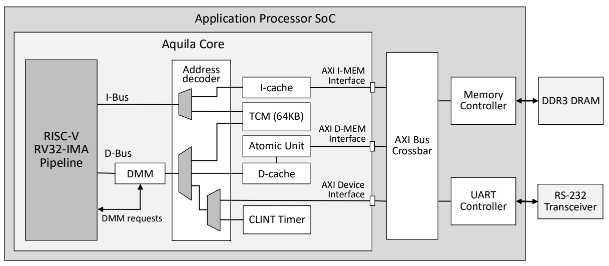

Fig. 1. The architecture of the RISC-V application processor.

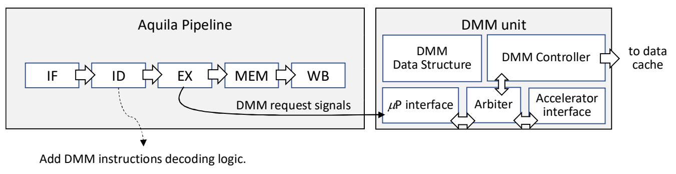

Fig. 2. The proposed interface between the processor pipeline and the DMM unit.

collection tasks can run in the background without interference to the processing cores. This is particularly true for systems with high performance data caches. Furthermore, simultaneous allocation requests can be initiated directly from different heterogeneous computing cores to the common memory manager. There is no need for the heterogeneous computing cores to wait passively for the general-purpose processing cores to handle memory requests. A hardware memory allocator for the heterogeneous multi-core systems has been proposed in [7]. The design is based on the buddy system that is commonly used for software DMMs. Although buddy systems can speed up the search for a free memory block for software-based memory allocators, it requires more complex data structures that may not be necessary for a hardware memory allocator since a hardware logic can perform parallel operations easily. In [8], a hardware free-list cache has been proposed to facilitate the fast accesses of memory management data structure by the processor. However, the performance evaluations are based on simulations instead of synthesizable models.

The contribution of this paper is as follows. First, we have proposed to integrate a dynamic memory management module into the processor architecture. Secondly, we have designed the VLSI architecture of a DMM module based on an efficient and well-adopted memory management algorithm for embedded systems. Finally, the proposed architecture has been implemented based on an open-source RISC-V processor, and verified on an FPGA platform. The RTL model of the proposed DMM design and the integrated application processor SoC is available at https://github.com/eisl-nctu/dmm for others to verify our results. The organization of this paper is as follows. In section II, we present an overview to the proposed application processor. Section III presents the design and the implementation of the DMM module. Section IV shows the experimental results. Finally, some conclusions and future work are given in section V.

II. Overview of The RISC-V Application Processor

The proposed application processor architecture is illustrated in Fig. 1. The SoC contains an open-source RISC-V RM32-IMA processor core, Aquila [9]. The Aquila core is a reusable AXI4 IP core that integrates a classical single-issue in-order five-stage RISC-V pipeline, an on-chip dual-port tightly-coupled memory (TCM) block, level-one instruction and data caches, an atomic-instruction monitor, and a Core-Local Interrupt (CLINT) module with a system timer. It supports RISC-V CSR instructions and is capable of executing a multi-threading real-time OS such as FreeRTOS. The RTL model of Aquila is written in Verilog, when synthesized using a Xilinx FPGA clocked at 100MHz, the performance of Aquila is 1.0 DMIPS.

In this paper, the Aquila core is extended with a DMM module that handles the dynamic memory requests from the processor core. As shown in Fig. 1, the DMM module is not an external IP of the AXI bus infrastructure. Instead, the DMM module is part of the processor core architecture. The slightly modified pipeline architecture of Aquila to accommodate the DMM module is shown in Fig. 2. The Instruction Decode (ID) stage is modified to supoort the decoding of DMM instructions (to be discussed later) and the Execute (EX) stage sends the DMM request signals to the microprocessor interface of the DMM. The accelerator interface of the DMM can be used to accept dynamic memory allocation requests from an accelerator logic. However, it is for future extension and not used in this research.

There are several reasons why the DMM module shall not be implemented as an AXI-IP external to the processor core:

(1) DMM must organize the main memory as a link list of the free/used memory blocks. It should access this data structure through some caching system or the performance would be degraded significantly due to access latency of DDRx main memory. If the DMM module maintain its own caching subsystem outside the processor core, there would be a cache coherency issue between the CPU cache and the DMM cache. It would be more efficient if the DMM is located between the load-store unit and the cache controller of the processor core.

(2) If DMM is an external AXI IP, the processor core must use the memory-mapped I/O interface to issue memory management requests and receive memory pointers. However, memory-mapped I/O (MMIO) interface may have higher overhead than the co-processor interface of modern processors. Therefore, modern application-specific processors often choose to use the coprocessor interface for function extension [10]. In the proposed architecture in Fig. 1, user-defined instructions are used to trigger the memory allocation and garbage collection requests. As a result, no AXI bus traffics will be generated for memory operations unless the requests require updating of the data cache blocks.

(3) The operations of the data cache controller, the virtual memory-management unit (MMU), and the DMM unit are highly correlated. For most modern processors the first two modules are part of the processor architecture. It would be less efficient if the DMM is implemented as a standalone software library. Ideally, a unified memory manager should be used to manage caching, address-mapping, and dynamic allocation and garbage collection of the memory subsystem in an optimized manner. The proposed architecture in this paper is a first step towards this long-term microarchitecture innovation for future processors.

III. Architecture Design of the DMM Hardware

A. The Data Structure Maintained by the DMM Circuit

The DMM module presented in this paper is a circuit implementation of the memory allocator in the popular Newlib [11] C library. In this design, the heap space in the main memory is segmented into memory chunks of used/free memory blocks. The data structure of each chunk is shown in Fig. 3. Note that there is a duplication of the chunk size field for the free memory

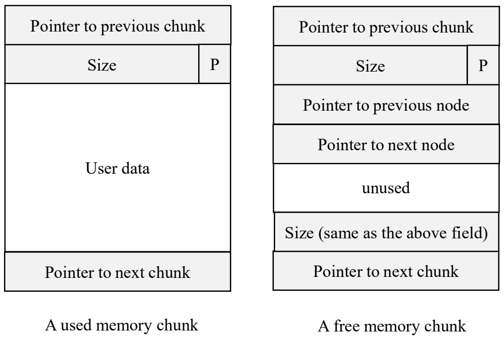

Fig. 3. Memory chunks in the heap space.

chunk because this makes the merging of consecutive free memory blocks easier. When the system is first initialized, the entire heap space is composed of only one big chunk of memory block, called the heap-top. Each memory allocation operation before any free operation has been issued will split the heap-top chunk into two chunks, an allocated chunk and the remaining heap-top chunk. If some of the allocated chunks are freed, the freed chunks will be recording using an array of link lists as shown in Fig. 4.

Each link list contains a list of memory blocks of certain sizes. Link list #1 points to the heap-top chunk. Link lists #2 ~ #64 contain chunks of the exact size from 16-byte to 512-byte. Link lists #65 ~ #127 contains blocks within a size range, sorted in descending order. For example, link list #65 is a list of blocks of sizes within the range (512, 576]. Note that Fig. 4 is a conceptual diagram. The 126 head pointers of the link lists are stored in the registers of the DMM module. The memory block nodes from different lists are stored in the main memory using the chunk structure.

The main controller of the DMM module is shown in Fig. 5. When an "malloc(size)" instruction is executed by the processor, the controller enters the "malloc analysis" state to determine which sub-controller will be invoked. The DMM module maintains a 126-bit non-empty bitmap register to indicate which free-chunk link lists are non-empty. First, if the size parameter rounded up to multiples of 8-byte matches one of the fixed size bins, and the bitmap register shows that there is a link list of that size, the exact-bin allocation controller will be invoked. If not, and one of the sorted-bins has a link list that can accommodate the requested memory size, the sorted-bin allocation controller will be invoked. If both attempts fails, the heap-top allocation controller will be invoked. Note that, the "malloc analysis" state only take one clock cycle to invoke the correct controller, which is much faster than a software library that executes searching of a fit chunk sequentially.

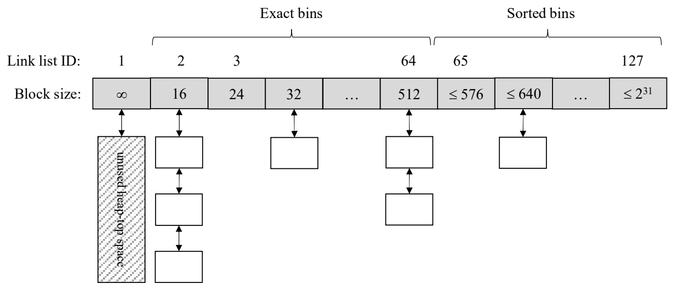

Fig. 4. Link lists of free memory blocks maintained by the DMM module.

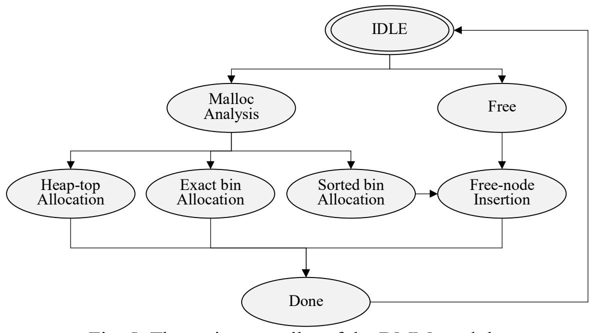

Fig. 5. The main controller of the DMM module.

If the sorted-bin allocation controller is invoked to satisfy the request, the first free-chunk in the target link list that is larger than or equal to the requested size will be selected and split (if necessary) to satisfy the request. The remaining free-space of the split will be inserted into the proper link-list as a free chunk during the "Free-node insertion" state in the main controller. This insertion state will also be invoked when a "free()" operation is issued by the processor so that the freed space will be inserted into one of the link lists. For a software library, the free operation can be expensive when merging and purging are desired for garbage collection. For hardware implementation, since all the operations related to memory free can be executed in background without imposing computational costs of the processors. In addition, the DMM module can wait until there is no external memory accesses from the processor to update the chunk data structure in the main memory. Therefore, for efficient operations of the DMM module, it has to be designed alongside the data cache, as will be presented in the next subsection.

B. Integration of the DMM Module with the Data Cache.

One may argue that the DMM module can be designed as a stand-alone AXI IP, and invoked via the MMIO interface. This way, there is no need to modify the processor core microarchitecture and multiple cores can share a common MMU module easily. However, if DMM sits outside the processor core, its performance will be hindered significantly since it has to access the main memory directly for the maintenance of the link list structure of the memory chunks and send cache coherence requests to all processor cores. Comparisons of the proposed design against this stand-alone IP design will be presented in the experimental section. Even if the DMM implements its own caching subsystem to speed up the main memory accesses, it still has to resolve the coherence problem with the processor cache. In short, it is more appropriate to have an integrated design of the DMM module within the processor microarchitecture. In the proposed architecture in Fig. 1, the DMM is located between the processor core and the level-one data cache controller. All processor memory requests will pass through the DMM module and send to the data cache. However, if the processor issues a DMM requests from the Execute stage through the DMM interface, succeeding memory requests to the data cache will be stalled until the DMM module finishes its update of the link list structure of the memory chunks through the data cache.

Theoretically, the DMM module and the level-one data cache controller can be merged into a single unit for more efficient operations. In particular, for the multi-core systems, it would be easier to have a unified DMM/level-one cache

	31 20	19 15	14 12	11 7	6 0
malloc()	imm[11:0] (x)	rs1	funct3(000)	rd	1101011
free()	imm[11:0] (x)	rs1	funct3(010)	rd (x)	1101011

Fig. 6. User-defined instructions for malloc() and free(). In the figure, (x) stands for unused field.

controller for each processor core. However, we will leave such design for future work.

C. Interface to the memory allocator.

Modern ISA designs typically allows for user-defined instruction extension. Compared with function extension using the MMIO interface and a custom IP, a user-defined instruction does require system bus traffic for the processor core to communicate with the custom IP. In addition, the parameter source and the returned value can be passed in-and-out directly using the processor registers. For functions, such as malloc(), that only pass an integer parameter and expect the return of a pointer, user-defined instruction provides an ideal interface.

In the proposed design, we have defined two instructions in the user instruction space of the RISC-V ISA, as shown in Fig. 6. For malloc(), the size parameter is passed in using the 'rs1' register. The Execute stage will send its content to the corresponding DMM interface port and wait for the returned pointer. The DMM returns the pointer to the free memory block as soon as the correct memory chunk is selected. The pointer will be written back to the 'rd' register at the Writeback stage. Note that the physical modifications of the link lists in DMM and the chunk structure in the main memory will be executed in background so that they will not stall the processor pipeline.

IV. Experimental Results

The synthesizable RTL model of the proposed DMM architecture is written in Verilog. It has been verified on a Xilinx KC-705 FPGA platform using Vivado 2020.1. The target frequency for the Aquila SoC is 100MHz. The resource utilization of the logic on a Xilinx Kintex-7 XC7K325T FPGA is shown in TABLE I. The dual-port TCM is set to 64KB.

TABLE I. Resource usage on Xilinx Kintex-7 XC7K325T.

	LUT	Flip-Flop	BRAMs	DSPs
RISC-V Pipeline	4765	3937	0	4
D-Cache (16KB)	4927	1932	0	18
I-Cache (16 KB)	4827	11087	0	16
DMM	4149	975	6KB	0
Entire Aquila SoC†	38341	40197	210KB	4
FPGA %	19%	10%	12%	1%

† The entire SoC includes the memory controller and all the infrastructure IPs from Xilinx.

To evaluate the performance gain of the proposed DMM, we have generated five workload traces of memory allocation/free operations on a Linux system. The workload traces are collected using a modified version of the standard C dynamic linking library that log the malloc()/free() calls from the applications into a workload trace file. Note that the workload traces are not used to evaluate the end-to-end full application performance of batch tasks with or without the hardware DMM. Therefore, the timestamps of the malloc()/free() operations are not used in these traces. The key point of DMM performance is for interactive and real-time tasks where DMM operations may delay UI and real-time responses. For interactive applications, especially when they are written using dynamically-typed object-oriented languages, DMM operations can easily cause GUI stalls that hinders user experiences or violation of real-time constraints due to a large number of object creations in a short time period. This is especially true when the applications create a lot of new objects to instantiate a new session or when the background garbage-collection mechanism kicks in.

The trace file contains the type of dynamic memory operations, *malloc(size)* and *free()*, in sequential order. The number of memory operations for the five traces are 632, 5446, 5940, 35036, and 209980, respectively. Some histograms of the malloc() workload executing Python (trace 3) and Java (trace 4) programs are shown in Fig. 7. Note that for the Java workload, there is a single peak of 13,757 allocations for memory blocks of size 16-byte. This is typical for object-oriented applications where a lot of small objects will be created/deleted during runtime.

To evaluate the performance of the proposed hardware DMM against the memory management functions of the Newlib software library [11], we have extracted the malloc() and free() functions from the Newlib library and compiled them with the gcc –O2 optimization flag. A test program is designed to issue the same malloc() and free() calls as specified in the workload traces of each applications to either the software library or the hardware DMM. The CSR 64-bit cycle-counter registers (mcycle, mcycleh) of the RISC-V processor are used to measure the total number of clock cycles for all the malloc() and free() operations. The results are shown in TABLE II and TABLE III.

From TABLE II, it is evident that the hardware DMM outperforms the software DMM on an average of over three times for malloc() operations. For free() operations shown in TABLE III, the performance of the hardware DMM is well-over four times against that of the software DMM. Note that in TABLE II and TABLE III, the time measurements are the "foreground" time. Because the function calls for the software DMM are blocking calls, all the time ticks required for the memory management operations are counted as the foreground time. On the other hand, for the hardware DMM, both malloc() and free() are non-blocking calls. Therefore, the foreground time of each call is only part of the time it takes to process the memory management operation. After the DMM locates a memory block, it can return the pointer to the application program immediately, and then execute the maintenance of the link list in the background (by the hardware) while the application is using the memory block in the foreground.

In order to get insight into the percentage of background time for DMM, we have designed two internal counters in the DMM logic to measure the cycles it takes to execute the hardware malloc() and free() operations of the applications. TABLE IV shows the result. As one can see, one major advantage of the hardware DMM module is that it can hide a large portion of the

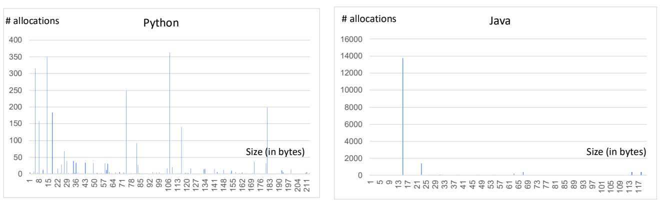

Fig. 7. The histograms of the malloc() sizes of traces 3 (Python) and 4 (Java).

memory operations in the background in parallel while the processor is busy running the applications. The entries labeled with BG% stand for the percentage of time it spent in the background in the overall processing time. For the hardware free() operation, roughly 80% of the processing time is hidden in the background.

There are two peculiar entries in TABLE IV that are worth discussion. First of all, the BG% entry of Trace 1 shows that the percentage of the background operations of malloc() is 46%. The reason is that Trace 1 has a fairly small number of memory operations. Thus, the application ends soon after the DMM data structure has been properly cached. As a result, the overhead of data cache initialization becomes evident in the background portion of the malloc() operations. Similarly, for Trace 4, the BG% number is –13%. That is, the function call time is actually slightly longer than the processing time. This is due to the final cache-related operations stalling the execution of the Aquila processor after the hardware DMM has returned to the IDLE state. Therefore, the malloc() calls did not return immediately.

Finally, TABLE V shows the case when the hardware DMM is used as a stand-alone AXI bus IP instead of integrated into the Aquila microarchitecture. The speed ratio in this table is computed against the time ticks in TABLE IV. In this case, the DMM logic has no access to the data cache of the Aquila core and has to access the main memory directly for memory management operations, which significantly slows down the performance. Although one can implement more sophisticated cache-coherent bus protocols, such as the AXI ACP, to fix this problem, it does require higher cost on the bus infrastructure. In summary, the proposed architecture is cost-effective for deeply embedded applications.

TABLE II. The malloc() function call time in clock cycles.

	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Newlib SW	60,041	544,875	623,394	3,262,795	19,833,382
DMM HW	10,566	188,900	195,464	1,058,993	6,380,338
Speed ratio	5.68	2.88	3.19	3.08	3.11

TABLE III. The free() function call time in clock cycles.

	Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
Newlib SW	34,141	392,779	389,305	1,863,239	16,782,290
DMM HW	4,979	85,472	79,922	415,953	2,983,677
Speed ratio	6.86	4.60	4.87	4.48	5.62

TABLE IV. The total processing time for the hardware DMM functions calls.

		Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
malloc	overall	19,416	205,991	239,443	933,280	6,812,579
	BG %	46%	8%	18%	-13%	6%
free	overall	32,366	403,343	414,590	1,942,614	13,976,990
	BG %	85%	79%	81%	79%	79%

TABLE V. The performance of DMM as a standalone AXI bus IP. This table shows the benefit of integrating the DMM unit into the processor core.

		Trace 1	Trace 2	Trace 3	Trace 4	Trace 5
malloc	time	420,047	320,418	442,970	1,999,291	18,174,206
	ratio	0.05	0.64	0.54	0.47	0.37
free	time	193,932	1,742,239	1,849,297	9,944,019	64,061,657
	ratio	0.17	0.23	0.22	0.20	0.22

V. Conclusions and Future Work

In this paper, we present the design of a hardware DMM module and its integration into the microarchitecture of an open-source RISC-V application processor. The proposed DMM module can perform the dynamic memory management operations in the background and is more efficient than a software DMM library. Currently, the design does not support aggressive memory defragmentation. The complete FPGA-synthesizable source code of the proposed system will available on github. For future work, we will add a more aggressive background memory defragmenter and extend the function of the memory manager to support virtual memory management. In addition, the integration of the DMM module with the data cache controller can be further optimized.

Another improvement that will be done in the future is to combine the DCA module [3] and the DMM unit to provide more efficient, cache coherent interface across multiple processor cores and accelerator IPs. For example, for a multi-core processor, in order to support cache coherence, there would be a coherence cache controller where the DMM can be integrated with. When a specific processor core issues a DMM instruction, the DMM module will forward the request to the coherent cache controller and make sure the cache blocks across all caches are consistent. In this case, an arbiter is required to schedule concurrent DMM requests from all cores. We will leave the integration of the proposed DMM unit into a multicore system for future work.

Acknowledgment

This work was partly funded by the Taiwan Semiconductor Research Institute (TSRI), NARL, Taiwan, ROC.

References

[1] A. Coleman and J. Zalewski, “A study of real-time memory management - evaluating operating system's performance,” *Automatyka/Automatics*, 17. 29. 10.7494/automat.2013.17.1.29.

[2] Mayer, Alastair JW. "The architecture of the Burroughs B5000: 20 years later and still ahead of the times?." *ACM SIGARCH Computer Architecture News 10.4* (1982): 3-10.

[3] C.-J. Tsai and Yi-De Lee, “Embedded TCP/IP Controller for a RISC-V SoC,” Proc. of the 30th IEEE Int. Conf. on Very Large Scale Integration (VLSI-SoC 2022), Patras, Greece, October 3-5, 2022.

[4] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, “Dynamic Storage Allocation: A Survey and Critical Review,” *Proc. IWMM*, pp.117–128, 1995.

[5] J. L. Peterson and T. A. Norman, “Buddy systems,” *Communications of the ACM*, 20(6):421–431, 1977.

[6] W. Li, S. Mohanty, and K. Kavi, “A Page-based Hybrid (Software-Hardware) Dynamic Memory Allocator,” *IEEE Computer Architecture Letters*, Vol. 5, Issue 2, 2006.

[7] Z. Xue and D. B. Thomas, “SysAlloc: A Hardware Manager for Dynamic Memory Allocation in Heterogeneous Systems,” *Proc. of Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2015.

[8] S. Kanev, S. Xi, G.-Y. Wei, and D. Brooks, “Malloc: Accelerating Memory Allocation,” *ACM SIGPLAN Notices*, Vol 52, Issue 4, Apr. 2017, pp.33-45.

[9] The Aquila SoC project, available on-line at https://github.com/eisl-nctu/aquila.

[10] Y. Zhou, X. Jin, T. Xiang, and D. Zha, “Enhancing Energy Efficiency of RISC-V Processor-based Embedded Graphics Systems through Frame Buffer Compression,” *Microprocessors and Microsystems*, 77 (2020), 103140.

[11] Newlib - C library for embedded systems, available on-line at https://github.com/ourairquality/newlib.