Improving Dynamic Memory Allocation on Many-Core Embedded Systems With Distributed Shared Memory

Ioannis Koutras, Iraklis Anagnostopoulos, Alexandros Bartzas, and Dimitrios Soudris, *Member, IEEE*

***Abstract*—Memory management on many-core architectures is a major challenge for improving the overall system performance. Memory resources are distributed over nodes for faster local accesses. Dynamic workloads heavily depend on memory requests and inefficient memory management leads to severe bottlenecks and performance degradation. In this paper, we focus on optimizing dynamic memory allocation on such platforms and present a scalable, microcode-accelerated distributed dynamic memory manager. The proposed manager exploits the presence of a hardware accelerator while offering a C application programming interface to application developers. Experimental results show performance gains on average 10% compared to allocators written purely in C and sufficient scalability as platform size increases.**

***Index Terms*—Dynamic memory management, microcode-accelerated, multiprocessor system-on-chip, network-on-chip (NoC).**

I. Introduction

MOORE'S law implies computer systems to keep becoming more complex by adding extra functionality on the same chip. This trend correlates strongly in the embedded domain, where consumer needs for portable devices of high-resolution cameras and displays drive the market. Hardware engineers put a considerable amount of effort in designing the architecture of such systems in order to maintain scalability. Interconnection networks have become quite complex, evolving from conventional buses to sophisticated network-on-chip (NoC). Although the implementation of the distributed shared memory (DSM) model comes with memory bottleneck issues, designers still select it for its easy-to-use programming model.

Dynamic memory managers (DMMs) help programs determine during run-time how and where dynamic data should be stored. Memory is allocated from a large pool of unused memory area called the *heap*. Existing approaches that handle dynamic data requests rely mostly on software solutions that tradeoff flexibility for processor cycles, resulting in performance degradation. Fast hardware solutions are already available, but programmers tend to follow a more centralized approach and do not exploit platform's characteristics. This centralized approach creates a central point of failure rendering the whole system unusable in case the central core fails. Moreover, a central core hinders scalability, because it is a bottleneck for processing and communication.

Many DMMs and related methodologies have been proposed targeting the embedded scope. Atienza *et al.* [1] provided a flow for designing memory managers which avoid wasting memory space and thus being optimized for embedded systems with scarce memory resources. The same concept is presented in [2], where power efficiency is also taken into account during the design of the DMM. As discussed in [3], since embedding devices are nowadays composed of multiple processing cores, it is worth expanding the design space of DMMs to multiheap organizations. Finally, Kim *et al.* [4] proposed a full-stack (in terms of hardware and software) memory management scheme targeting high-end Android devices with special memory management hardware unit. All these techniques apply on conventional single-processor systems and potentially on some multiprocessor ones, but do not take into account neither the NoC interface of novel hardware platforms, nor the complexity of dynamic applications.

Hardware-accelerated DMMs are also a hot topic, since they can achieve high performance. Marchal *et al.* [5] showed that using scratchpad memories along with direct memory access controllers has a positive impact on energy consumption for dynamic interactive applications. In the general-purpose domain, Intel has presented support for hardware transactional memory and a proposed memory allocator using it [6] shows more than double performance in memory-intensive benchmarks without compromising memory footprint. A hardware memory management unit responsible for dynamic memory allocation and de-allocation is presented in [7]. However, it is a centralized unit, able to allocate only complete global memory pages; the management of the data (de)allocation is left to the processors. A distributed, application-specific DMM using hardware-accelerated memory functions has been proposed in [8]. Although the latter solution seems tailored to embedded platforms, it lacks of a high-level interface and relies on priority tables making the DMM difficult to get configured and scaled for systems with more than four processing nodes.

In this paper, we propose a novel dynamic memory allocator for DSM embedded systems which is accelerated by hardware

Manuscript received June 7, 2016; accepted July 17, 2016. Date of publication July 29, 2016; date of current version August 26, 2016. This manuscript was recommended for publication by P. R. Panda.

I. Koutras and D. Soudris are with the School of Electrical and Computer Engineering, National Technical University of Athens, Athens 15780, Greece (e-mail: joko@microlab.ntua.gr).

I. Anagnostopoulos is with the Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, IL 62901 USA.

A. Bartzas was with the School of Electrical and Computer Engineering, National Technical University of Athens, Athens 15780, Greece.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2016.2596318

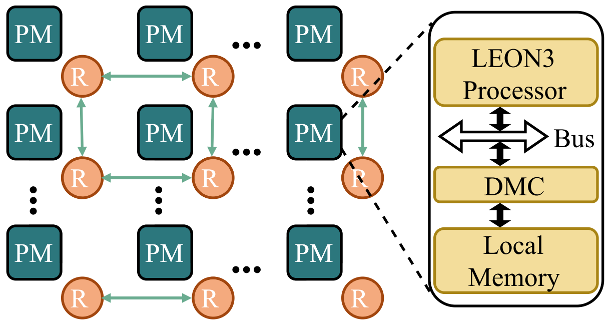

Fig. 1. DSM platform with programmable memory controllers as seen in [9].

and yet maintains scalability. The main contributions of the introduced allocator are: 1) It exposes a unified heap space accessible by every node; 2) It exploits the presence of programmable hardware memory controllers by using customized microcoded functions to accelerate dynamic data management functions; 3) It requires minimal initial configuration; and 4) It offers an application programming interface (API) to manage the heap, similar to the `C` standard library one [as in `malloc()/free()`].

II. Exploiting Hardware to Improve Dynamic Memory Management

Suppose that we have a DSM system on the one in Fig. 1: multiple nodes of processor-memory are interconnected via a packet-switched mesh network of routers. Each node contains additionally a dual microcoded controller (DMC), a programmable hardware accelerator to handle memory requests [9]. The utilized platform has been proved to accelerate memory operations in a DSM environment [8]–[10] by programming the DMC and providing services such as: virtual-to-physical (V2P) address translation, synchronization, cache coherency, memory consistency, and shared memory access [9].

The V2P service enables the DSM model: local memories are organized with private and shared parts and by using the V2P service, the processors are able to access via the local DMC controller any shared memory by using a higher address scheme than the local one (e.g., `0x40200000` versus `0x00000`).

Thanks to the V2P service, porting a DMM of a more generic platform is a straightforward procedure. We have ported for the platform of Fig. 1 the most power-efficient allocator of [2] with minimal changes in code. The developed allocator considers the heap as a single one and thus, writing applications accessing multiple memories from multiple nodes is easier. Unfortunately, we prove in a later section the performance to be weak, since there is no awareness of the memory locality: the allocator chooses memory regions to allocate regardless of which processor makes the allocation request.

The locality problem is solved by the allocator presented in [8], where heap space is fragmented into smaller heaps, one per node, so that specific DMC instructions (microcode from this point forward) can speed up the memory allocation processes. Albeit more performant, this approach comes with excessive development costs: application developers should know *a priori* which processors need to access which heaps and define accordingly a priority table per node.

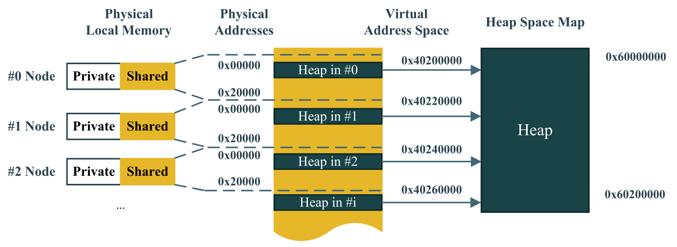

Fig. 2. HSM on top of V2P translation service.

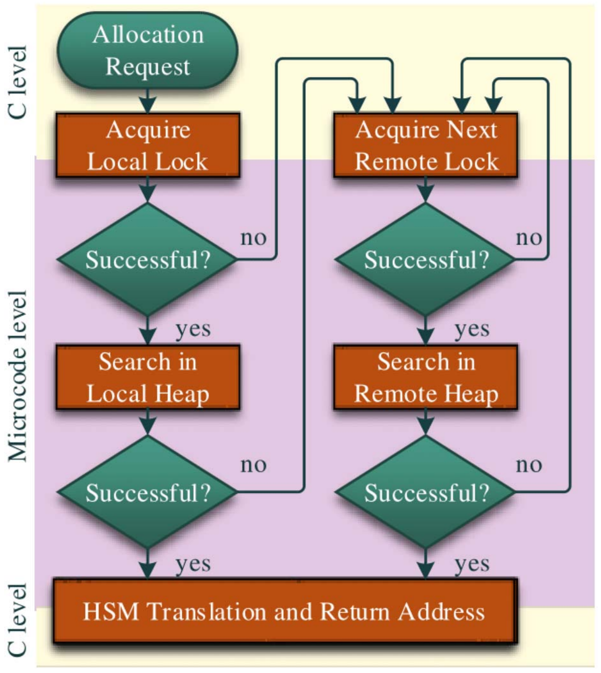

Fig. 3. Complete allocation scheme of the proposed allocator.

Ultimately, we identify this as a tradeoff between a truly single, easy-to-program heap, and hardware acceleration. In order to maintain a single distributed heap, while still offering microcode-accelerated dynamic memory management, we propose two new features for the target platform: 1) *heap space map (HSM)* and 2) *lazy heap selection*.

The HSM involves the creation of a second virtual address layer as seen in Fig. 2. The new addressing scheme should start higher than V2P (e.g., starting from `0x60000000`) and it unites the smaller heaps that are found in each node, so that a continuous memory space becomes available as a heap to the application. The heap-to-node translation is performed at two levels: first by the HSM service (in `C` level) and second by the corresponding memory controller (microcode V2P service).

Additionally to the HSM, we propose a lazily heap selection scheme to avoid priority tables. In [8], each node stores in its local memory the possible heaps to trigger in order to serve a request. If one heap is occupied while the allocator wants to just check it, the latter has to wait indefinitely for it. What we propose instead is to evaluate all the remote heaps with no exception, after verifying that the local heap is out of space. Moreover, if the allocator tries to acquire the lock of a remote heap and fails, it should move to the next remote heap.

Based on the virtual memory address space mentioned above and the heap selection algorithm, we propose an allocation scheme as seen in Fig. 3. Deallocation happens accordingly: after an HSM translation, a message is sent to the relevant DMC to free the memory block. The proposed allocator implements at microcode level all the fine-grained actions for performing dynamic memory allocation and deallocation. The application interfaces and the introductory mechanism of the lock mechanisms in the HSM are impletemented in C.

The main differentiators of this letter are hence the following.

1) The proposed allocator considers a global, continuous address space for memory requests like the C allocator [2] and unlike the microcoded one [8].
2) The proposed allocator leverages the use of DMC microcode like the microcoded one, while the C one does not.
3) The proposed allocator does not use priority tables as the microcoded one does.
4) The proposed allocator evaluates lazily the memory availability in remote heaps, while the microcoded allocator has to exhaustively evaluate a possible memory allocation in one remote heap before locking and evaluating another one.
5) The proposed allocator does not need any modification when using it for a different application.

III. Evaluation

We have evaluated our approach and compared it with other allocators on the platform architecture presented in [9]. No hardware change has been made. Each node is composed of a LEON3 processor, a DMC and memory which can be shared among the nodes. All the DMCs are interconnected in Nostrum [11], a packet-switched mesh network.

For the evaluation of our proposed allocator we have used the traces from four benchmarks presented in [12]. Those benchmarks have been proven there indicative of the performance of a memory allocator for embedded systems. More specifically: 1) features from an accelerated segment test (FAST), a computer-vision corner detection kernel; 2) a Gaussian kernel for blur effect; 3) integral, a kernel which calculates the integral of matrix elements; and 4) a matrix multiplication kernel. All benchmarks follow the master-slave code design. One node acts as the controller of the platform responsible for handling task management, while the rest of them are available for task execution. Dynamic memory management is important in such platforms. If only the master node handles the memory management, simulations show that the cycles spent regarding DMM for the used benchmarks compose on average the 18.8% (minimum 10.83% for the FAST application and maximum 23.12% for the Gaussian one) of the overall application cycles.

We first compared the performance of the proposed allocator against a hardware-based solution and a software-based one on a 2×2 system: 1) the memory distribution-aware microcoded DMM in [8] and 2) a pure private heap memory allocator selected from [2] and implemented in C. The C allocator was selected because it was on average 29% faster against other general-purpose solutions, exhibiting also reduced fragmentation, and thus is suitable for embedded systems. The performance of the compared allocators normalized to the performance of the microcoded one can be seen in Fig. 4. The microcoded allocator is the fastest of all since all

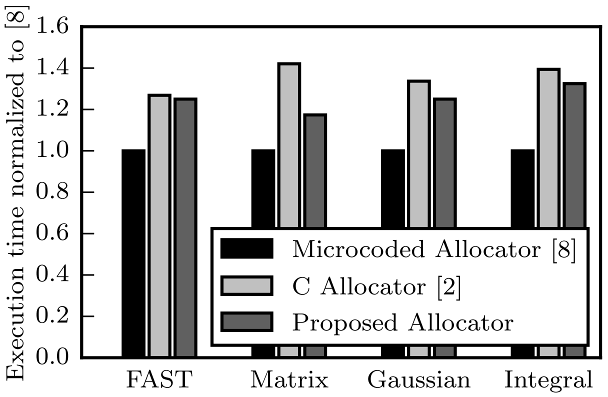

Fig. 4. Total execution time comparison of the proposed allocator with [2] and [8] under a pure private heap for a 2×2 platform. The proposed allocator makes a compromise between performance and ease of use.

operations and decisions are performed at low-level. However, this implementation lacks of a high-level API making the integration with C applications difficult. The proposed allocator is on average 25% slower than the microcoded allocator and 10% faster than the high-level one, since most of the (de)allocation operations are performed at microcode level and only the high-level heap address manipulation is performed in C. The presented allocator was designed for offering DMM without using a pure private heap structure. The goal is to let all nodes be aware of the heap status with a small performance penalty which in the case of pure private heaps does not exist. Thus, the proposed allocator is faster than a performance-aware allocator which is unaware of the broader platform.

We have then proceeded to validate the distributed behavior and scalability of the presented allocator by comparing: 1) the average cycles per (de)allocation event and 2) the number of microcode instructions needed for node intercommunication to the distributed microcoded allocator [8] for various platform sizes. The number of nodes ranged from 4 (2×2) to 64 (8×8) and they are all interconnected in a 2-D mesh network. To the best of our knowledge, this is one of the most popular network topologies and one can hardly expect more than 64 nodes on an embedded platform.

As shown in Fig. 5, the microcoded DMM [8] needs on average less cycles than the proposed allocator in order to server an event when the platform is smaller than 5×5. However, *when the platform size increases beyond 25 nodes the presented manager needs less cycles.* This happens because the distributed microcoded allocator [8] is based on priority tables. By using the technique of priority tables, each node stores in its local memory, as a single linked list [8], the possible nodes to trigger in order to serve a DMM request. For a 2×2 NoC the table contains four records while for an 8×8 NoC the records grow up to 64 per node and a lot of nodes have the same nodes as targets while performing a (de)allocation request. As shown in other experimental results [8], nearly 80% of the time for dynamic memory management was "wasted" for lock acquiring. So, as the platform size increases the handling of these tables requires more cycles while the proposed allocator uses a lighter and more generic communication scheme between nodes, supporting arbitrary sizes of platforms and scaling well as the platform

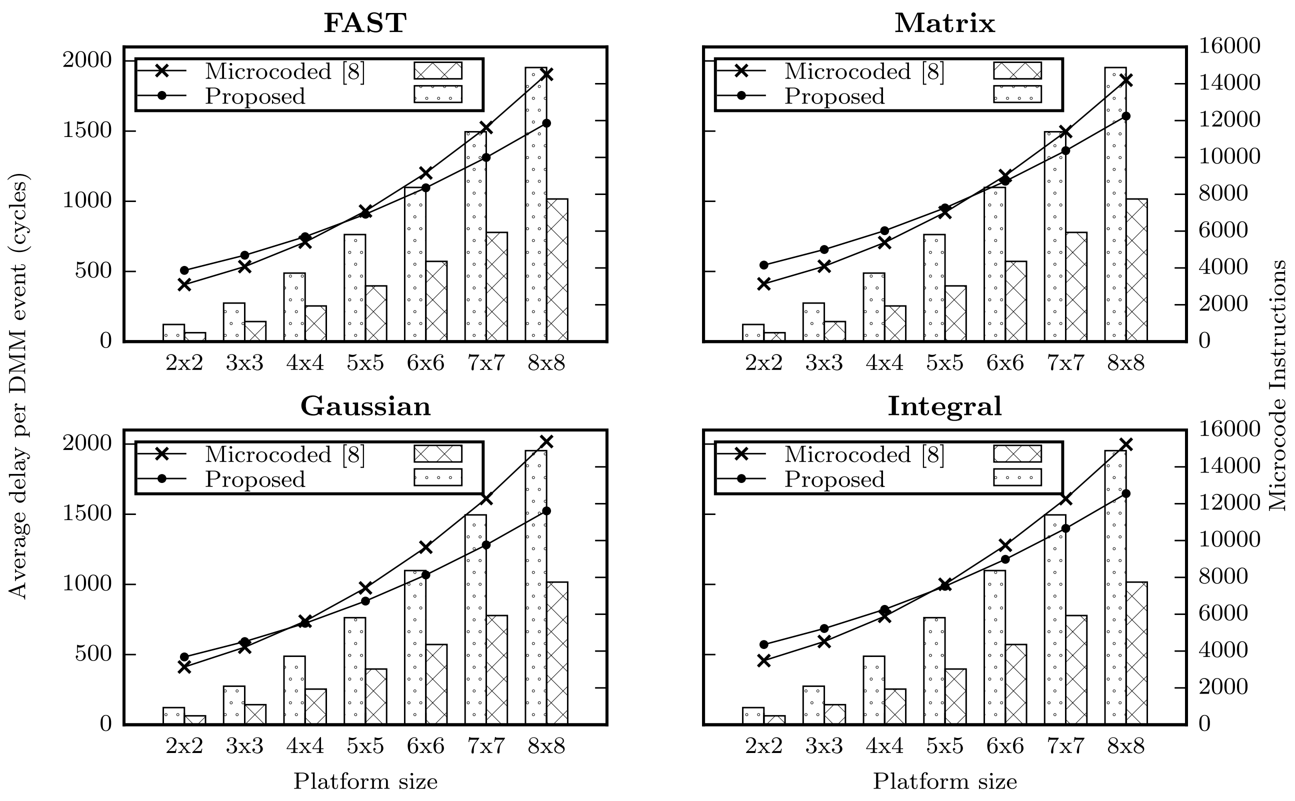

Fig. 5. Average cycles per DMM event and number of microcode instructions needed for intercommunication for various platform sizes and applications.

size increases. Fig. 5 shows that the purely microcoded allocator needs on average 29% more cycles to serve an event each time the platform increases, whereas, the proposed one has a smaller increase (approximately 20% more cycles). Although the purely microcoded one executes 1.9× on average less microcode instructions for node intercommunication, as shown with bars and the right axes of Fig. 5, the proposed solution proves to be much faster since it requests memory from more suitable nodes.

IV. Conclusion

This paper presented an efficient, hardware-accelerated, scalable dynamic memory allocator for NoC-based, many-core embedded systems in which the low-level DMM operations are implemented in microcode. The allocator provides consistent address space for the application dynamic data and the search of free space on remote memories is not halting. The allocator is generic to the platform and does not need recompilation for different applications. It requires, though, to be initialized with the memory size per platform and node. Experimental results proved that the proposed allocator 1) is scalable enough to surpass even low-level allocators on bigger platform sizes; 2) provides distributed functionality offering different parts of shared memory as a continuous heap space; and 3) serves requests 10% faster on average compared to high-level allocators without compromising ease of use.

References

[1] D. Atienza, J. M. M. Mendias, S. Mamagkakis, D. Soudris, and F. Catthoor, "Systematic dynamic memory management design methodology for reduced memory footprint," *ACM Trans. Design Autom. Electron. Syst.*, vol. 11, no. 2, pp. 465–489, 2006.

[2] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, and D. Soudris, "Energy-efficient dynamic memory allocators at the middleware level of embedded systems," in *Proc. 6th ACM IEEE Int. Conf. Embedded Softw.*, Seoul, South Korea, 2006, pp. 215–222.

[3] S. Xydis, A. Bartzas, I. Anagnostopoulos, D. Soudris, and K. Pekmestzi, "Custom multi-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *Proc. Int. Conf. Embedded Comput. Syst. (SAMOS)*, Jul. 2010, pp. 102–109.

[4] M. Kim, J. Koo, H. Lee, and J. R. Geraci, "Memory management scheme to improve utilization efficiency and provide fast contiguous allocation without a statically reserved area," *ACM Trans. Design Autom. Electron. Syst.*, vol. 21, no. 1, pp. 1–23, Dec. 2015.

[5] P. Marchal *et al.*, "Integrated task scheduling and data assignment for SDRAMs in dynamic applications," *IEEE Design Test Comput.*, vol. 21, no. 5, pp. 378–387, Sep. 2004.

[6] B. C. Kuszmaul, "SuperMalloc: A super fast multithreaded malloc for 64-bit machines," in *Proc. Int. Symp. Memory Manag. (ISMM)*, Portland, OR, USA, 2015, pp. 41–55.

[7] M. Shalan and V. J. Mooney, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *Proc. 10th Int. Symp. Hardw./Softw. Codesign (CODES)*, Estes Park, CO, USA, 2002, pp. 79–84.

[8] I. Anagnostopoulos *et al.*, "Custom microcoded dynamic memory management for distributed on-chip memory organizations," *IEEE Embedded Syst. Lett.*, vol. 3, no. 2, pp. 66–69, Jun. 2011.

[9] X. Chen, Z. Lu, A. Jantsch, and S. Chen, "Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller," in *Proc. DATE*, Dresden, Germany, 2010, pp. 39–44.

[10] X. Chen *et al.*, "Reducing virtual-to-physical address translation overhead in distributed shared memory based multi-core network-on-chips according to data property," *Comput. Elect. Eng.*, vol. 39, no. 2, pp. 596–612, 2013.

[11] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The Nostrum backbone-a communication protocol stack for networks on chip," in *Proc. 17th Int. Conf. VLSI Design*, Mumbai, India, 2004, pp. 693–696.

[12] I. Anagnostopoulos *et al.*, "Power-aware dynamic memory management on many-core platforms utilizing DVFS," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 1s, 2013, Art. no. 40.