Custom Microcoded Dynamic Memory Management for Distributed On-Chip Memory Organizations

Iraklis Anagnostopoulos[1], Sotirios Xydis[1], Alexandros Bartzas[1], Zhonghai Lu[2], Dimitrios Soudris[1], Axel Jantsch[2]
[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Royal Institute of Technology-KTH, Dep. of Electronic, Communication and Software Systems, Sweden

***Abstract*—Multi-Processor System-on-Chip (MPSoCs) have attracted significant attention since they are recognized as a scalable paradigm to interconnect and organize a high number of cores. Current multi-core embedded systems exhibit increased levels of dynamic behavior, leading to unexpected memory footprint variations unknown at design time. Dynamic Memory Management (DMM) is a promising solution for such types of dynamic systems. Although some efficient dynamic memory managers have been proposed for conventional bus-based MPSoC platforms, there are no DMM solutions regarding the constraints and the opportunities delivered by the physical distribution of multiple memory nodes of the platform. In this work, we address the problem of providing *customized microcoded DMM* on MPSoC platforms with distributed memory organization. Customization is enabled at *application- and platform-level*. Results show that customized microcoded DMM can serve approximately 7× more allocation requests compared to pure distributed memory platforms and perform 25% faster than the corresponding high-level implementation in C language.**

***Index Terms*—Multiprocessor System-on-Chip, Dynamic Memory Management, Network-on-Chip**

I. Introduction and Related Work

High-performance single-chip computing devices evolve from single- to multi- and even many-core architectures [1]. The embedded memory content in System-on-Chips increased from 20% ten years ago to 85% of the chip area today and will continue to increase in the future. Memories are preferably distributed for medium and large scale system sizes, since centralized memory has already become the bottleneck of performance, power and cost [2]. Adopting the MPSoC architectural template, multi-threaded applications are becoming increasingly prevalent for next generation embedded systems. Traditional memory optimization uses compile-time information and focuses on static allocation in respect to memory hierarchy [2]. For modern dynamic applications this is no longer possible since there is a lot of memory unpredictability, which cannot be captured by source code analysis alone. However, the increased dynamism in data storage leads to unexpected memory footprint variations unknown at design time. Dynamic Memory (DM) managers are responsible for organizing the dynamically allocated data in memory and servicing the application memory requests at run-time. As illustrated in [3] simple Dynamic Memory Management (DMM) implementations often form a performance and scalability bottleneck in the case of multi-threaded applications, affecting the memory and energy consumption of the overall system.

Software-only DMM solutions have been extensively investigated and form the current practice, being flexible but consuming many processor cycles, limiting system performance [3], [4]. In the field of embedded computing, a set of systematic exploration methodologies for single- and multi-threaded customization on DMM have been proposed in [5], [6] respectively, aiming to move from general-purpose to application-specific solutions. Hardware acceleration regarding memory management has been proposed by many researchers [7], [8], [9], [10]. A hardware memory management unit (SoCDMMU), responsible for dynamic memory allocation and de-allocation is presented in [8]. However, this is a centralized unit and could be a potential bottleneck in MPSoCs. Furthermore, SoCDMMU is able to allocate only complete global memory pages and the management of the data (de)allocation of the local (or private) memories is left out to the processors. A hardware MMU for NoC architectures, is presented in [10], offering general-purpose DMM for shared memory with a granularity of complete memory pages. Dedicated hardware DMM solutions can achieve high performance, but any small change in functionality leads to re-design of the entire module. *Thus, we focus on the microcoded approach as a promising alternative to overcome the performance-flexibility dilemma, offering a programmable and flexible solution to accelerate a wide range of applications.*

In this work, we employ a hardware dual-microcoded controller (DMC) [9] offering full support for dynamic memory (de)allocation (`malloc()` and `free()` functions) in a Distributed Shared Memory (DSM) environment. We adopt the microcoded approach to address DMM issues on MPSoC platforms, aiming for hardware performance but maintaining flexibility of software implementations. *The main contribution of this work is the design and implementation of a microcoded, platform-dependent and memory distribution aware customized DMM for MPSoC platforms*. In order to guarantee high performance, the proposed DMM services are developed on top of DMC [9] that *a) is responsible for handling distributed memory requests and b) mitigates processor's workload.*

II. Proposed Methodology Flow

The proposed methodology framework for supporting custom DMM on MPSoC platforms with distributed memories is showed in Fig. 1. Given the source code of a multi-threaded dynamic application, we perform: (i) Application-dependent and (ii) Platform-dependent DMM customization. First, the application-dependent DMM customization is performed by generating a set of Pareto configurations. Having as inputs these configurations, the platform dependent DMM customization is initiated, where the selected DMM is translated to

This work is partially supported by the E.C. funded FP7-215244 MOSART Project, www.mosart-project.org

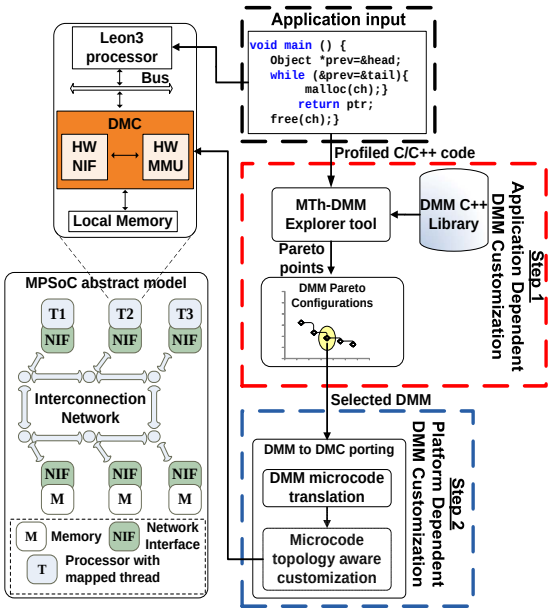

Fig. 1. Proposed methodology

microcode, enhanced with platform's aware information and uploaded to DMC's instruction memory.

A. Application-dependent DMM customization

In this step, we generate a set of Pareto customized Multi-Threaded Dynamic Memory Managers (MTh-DMM), tailored to the designer's constraints and the application's specific needs. MTh-DMM Explorer tool [6] was used to generate the application specific MTh-DMMs. This works on a platform-independent level searching among inter- and intra-thread DMM decision trees [6]. Exploration is performed based on a constraint-orthogonal partition methodology. The Pareto dimensions for extracting the application-specific memory allocators are: a) memory footprint and b) number of accesses. The MTh-DMM explorer tool generates application-specific C++ allocators, and these allocators are the Pareto configurations. Specifically, inter-thread exploration produces a first set of Pareto configurations which are propagated as constraints for further refinement during the intra-thread exploration phase. An automated code generator module produces the source code of the final DMM Pareto configurations, based on a C++ library containing modular and parameterizable MTh-DMM software implementations.

In this work, we focus on platform-dependent customization of DMM services rather than application-dependent ones. Thus, we use and extend work on multi-threaded dynamic memory management towards platform-specific customization.

B. Platform-dependent DMM customization

The main goal of this step is to move from platform-independent to platform-dependent DMM customization in order to increase performance and exploit platform's features. This is done through porting the application-specific DMM to DMC microcode. Two steps are required. At first, the high-level DMM configurations in C++ (platform-independent level) are translated to DMC's microcode. Second, the microcode is extended to take into consideration topology-aware features, such as memory distribution and communication cost.

DMM Microcode Translation: Having as input the C++ DMM Pareto configurations, our tool transforms the high-level code to equivalent microcode functions. We have built generic and fully configurable microcode templates based on the extracted C++ allocators and exploit platform's features. More specifically, according to platform characteristics the designer can generate a large number of Heap organizations by configuring different architecture-dependent DMM parameters, such as: (a) **Number, type and size of Fixed List Heaps**, (b) **Heap size** and (c) **Heap positioning** (local or global Heap). Also, the inter- and intra-thread allocation policies (i.e. First Fit, FIFO etc) from the DMM decision trees presented in [6] have been translated to equivalent microcoded functions.

Customization according to memory distribution: In order to achieve platform-dependent customization, the last step is to perform topology-aware refinement. As mentioned, the C++ DMM implementations work at a high abstraction level, thus leaving to the host operating system the decision of which (part of) physical memory is accessed during allocation requests. However, the management of accessing physical memory becomes dominant in MPSoC architectures due to memory distribution over the platform. For a given memory distribution, we further increase performance of the selected DMM by implementing microcoded functions instructing DMC: i) which (neighboring) Local Heap is more appropriate to ask for a (remote) allocation request and ii) which Global Heap is closer. At design time, based on *topology criteria*, we build priority tables $PT_{s,d}$, $(s, d \in N)$ for each node N of the MPSoC platform. $P \in N$ represents the processing nodes of the MPSoC and $M \in N$ represents the memory ones. $PT_{s,d}$ describes the priority weight of source s accessing destination d. $PT_{s,d}$ priorities are exploited at run-time guiding DMC to try allocation to (neighboring) nodes according to $PT_{s,d}$ table, starting from the node with the highest priority. The $PT_{s,d}$ value is defined in Equation 1.

$$PT_{s,d} = \frac{w_1 P_{s,d} + (1-w_1)(w_2 ML_d + (1-w_2)(w_3 MP_d + (1-w_3 D_{s,d})))}{\sum_{\forall i}^{(i \neq d)} \{w_1 Ps.i + (1-w_1)(w_2 ML_i + (1-w_2)(w_3 MP_i + (1-w_3 D_{s,i})))\}} \quad (1)$$

where $i \in M$, $P_{s,d}$ and $D_{s,d}$ are the power consumption and delay of the (s,d) link respectively. ML_d and MP_d are the memory latency and memory power consumption per access of the d memory respectively. Also, $\sum_{i=1}^{3} w_i = 1, w_i \geq 0$, are the weights for configuring the cost function. The microcode functions responsible for triggering remote Heaps (local or global) are totally independent and transparent to DMM's code. They are placed at the end of the microcode templates and they are automatically triggered when the local DMC asks for a remote (de)allocation request. DMC uses the message passing policy to propagate information to neighboring nodes. In that way, the execution of microcode to a different node is allowed even if the remote DMC has not received any signal from its own local core.

III. Experimental Results

We used the platform presented in [9] that offers base distributed shared memory (DSM) services such as virtual-to-physical (V2P) address translation, shared memory access,

synchronization, cache coherency and memory consistency. *DMM works transparently on top of these low-level base services.* All shared memories are globally visible to all nodes and organized as a single virtual addressing space using virtual addressing and V2P translation. The used memory architecture is NUMA (Non-Uniform Memory Architecture) and the dynamic data structures used by the application are allocated in SDRAM memories. The control store where the microcode resides during initialization and execution phases is a scratch-pad memory. This microcode is an instruction sequence that implements the allocator functions (`malloc()`/`free()`). The system is composed of Processor-Memory (PM) nodes interconnected via a packet-switched mesh network (Fig. 1). A node can also be a memory node without a processor, pure logic or an interface node to off-chip memory. Each PM node contains a LEON3 processor, hardware modules connected to the local bus, and a local memory. The DMC connects the processor, the local memory and the network, and serves requests from the local processor and the remote processors via the network concurrently [9].

A simplified network subsystem is used as the driver application [11]. The application consists of five kernels which are triggered by wireless streams. Each kernel corresponds to a thread and communicates with the other threads through asynchronous FIFO queues: the output of one thread is the input for another one. The application is organized into threads that perform: packet injection, packet formation, encryption (DES), TCP checksum, scheduling (DRR) and quality-of-service management. The application threads use dynamic data structures, and especially a) the list of nodes in the DRR algorithm and b) the queue of pending packets for each of the nodes. Through systematic application profiling we captured the allocation behavior of the application [11]. This information contains the block size distribution of the memory allocation requests. The application performs a big number of small-size continuous allocation requests (40 to 80 bytes) that correspond to network headers combined with bigger allocation requests (1200 - 1500 bytes) that correspond to actual network packets.

Based on the allocation behavior, the MTh-DMM Explorer tool generated the Pareto set of application specific DMM. Fig. 2 shows the two selected DMM Pareto configurations for the specific application. Column 2 depicts the per-thread Heap organization. The topology used for the evaluation of our approach is a 2×2 NoC. According to mapping decisions [12], nodes $(0,0),(0,1),(1,0)$ are processing nodes with their own Local Heap while node $(1,1)$ is the Global Heap. For local memories the Heap size is $4KB$ and Global Heap size (node $(1,1)$) is $32KB$. For the presented topology we implemented four different DMM configurations, depicted in Fig. 3, depending on memory distribution over the platform.

Configuration 1: Pure Distributed Memory. In this configuration (Fig. 3a), each node sends allocation requests for dynamic data to its Local Heap. There is no Global Heap. **Configuration 2:** Centralized single Heap. In this configuration (Fig. 3b), each node sends allocation requests for dynamic data only to Global Heap (node $(1,1)$). There are no Local Heaps. **Configuration 3:** Distributed multiple-Heap with Global Heap. In this configuration (Fig. 3c), each node first sends allocation requests to its Local Heap. If Local Heap is not able (due to lack of space) to serve any more allocation requests, the request then is sent to Global Heap. **Configuration 4:** Memory distribution-aware multiple-Heap with Global Heap. In this configuration (Fig. 3d), each node first sends allocation requests to its Local Heap. If Local Heap is not able (due to lack of space) to serve any more allocation requests, then, *according to priorities*, the Global Heap or the Local Heap of another node is selected in order to serve the allocation request.

Allocators	Description	Code size (# instructions)			
		Conf. 1	Conf. 2	Conf. 3	Conf. 4
DMM1	Fixed-List 0 [40B] Fixed-List 1 [1460B] Fixed-List 2 [1500B] GlobalHeap	1,407	485	1,736	1,859
DMM2	Fixed-List 0 [0– 40B] Fixed-List 1 [1280– 1460B] Fixed-List 2 [1460– 1500B] Fixed-List 3 [92B] Global Heap	1,467	485	1,796	1,919

Fig. 2. Description of the selected DMM configurations

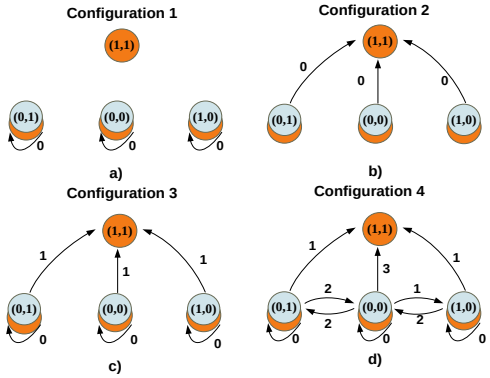

Fig. 3. a) Configuration 1 b) Configuration 2 c) Configuration 3 d) Configuration 4. Directed edges present the priority of choosing the destination node (0=highest priority, 3=lowest priority).

For the two selected DMMs and for each of the four aforementioned configurations, Fig. 4 shows: i) the cycles performed until a Heap memory overflow event appears, ii) the DMM event distribution and iii) the microcode performance compared to the equivalent `C` implementation on the LEON3 processor. Above each bar the actual count of served DMM events ***(Local Heap/ Global Heap)*** is presented until Heap memory overflow appears and this number of served DMM events is the same whether we follow the C or the microcode implementation for the same configuration (of the same DMM allocator). Heap memory overflow is the time (counted in cycles) when Heap was unable, due to lack of space, to serve any more allocation requests. According to Fig. 4, when DMM is aware of the memory distribution, the time in which Heap overflow appears, increases. Specifically, configuration 4 performs $7\times$ more cycles for DMM 2 compared to configuration 1. Also, when DMM is aware of the memory distribution, the number of served DMM requests increases. For example, configuration 4 achieved to serve approximately $7\times$ more DMM events for both DMM 1 and DMM 2 compared to configuration 1 verifying also the first result of Heap's lifetime increase. Additionally, according to Fig. 4, DMC serves the same number of DMM events in fewer cycles, performing faster than its corresponding `C` implementation (for the same

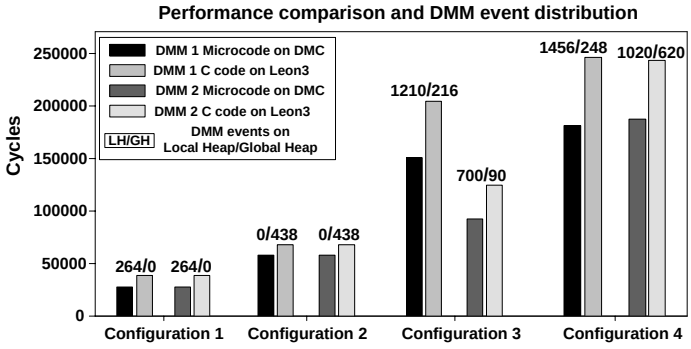

Fig. 4. Performance comparison and DMM event distribution.

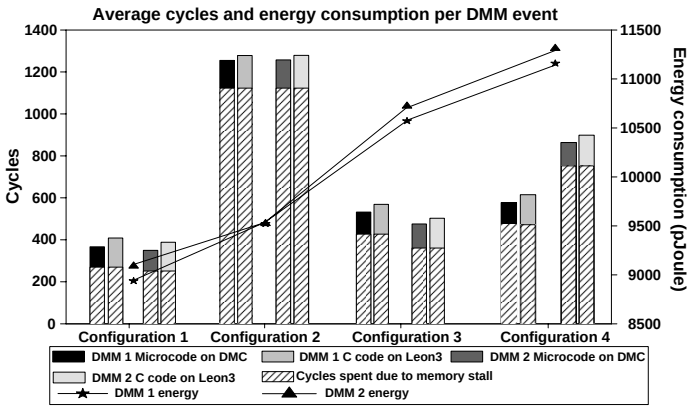

Fig. 5. Average cycles and energy consumption per DMM event

configuration). Specifically, DMM events performed by DMC (microcode) are on average 25% faster than LEON3 (C code). This happens because DMC is responsible for handling distributed memory requests and so every time LEON3 wants to access the memory, DMC is responsible for establishing the communication.

Fig. 5 shows: i) the average accelerator cycles, ii) the cycles spent due to memory stall and iii) the average energy consumption (pJoule) consumed per DMM event for DMM 1 and 2. Configuration 1 appears to be the fastest one, however it is the one that first exhibits Heap memory overflow. As expected, configuration 2 is the slowest among all. It needs more cycles since all processing nodes access the same global Heap for each (de)allocation and they are stalled due to memory synchronization (safe-lock) mechanisms. Configuration 3 offers good performance and additionally being more resilient in comparison to configurations 1 and 2. Configuration 4 requires a little more cycles than configuration 3 but it is a small penalty compared to the fact that it is the best solution regarding Heap memory overflow and served DMM events. We accounted energy consumed from the execution of the DMM microcode (based on post synthesis estimations at 0.09 um^2 of the DMC [9]) and the memory accessing pattern to the local and global heaps (based on Cacti [13] estimations). Configuration 2 consumes 6% more energy compared to configuration 1, since all its DMM events occur on the global Heap and the local controllers use their message passing instructions to guide the global Heap. Configuration 3, consumes approximately 18% and 19%, for DMM 1 and DMM 2 respectively, more energy in comparison to Configuration 1. This is caused by the fact that Configuration 3 consists of more microcode instructions (Fig. 2) and thus energy consumption is increased. Configuration 4, consumes approximately 25% more energy in comparison to Configuration 1. Also it consumes the highest energy amount due to the augmented code size and the often communication for detecting the most available Heap to use.

Experimental results show that the proposed approach for designing customized microcoded memory distribution-aware DMM (configuration 4): a) can serve more DMM events by using all available Heaps of the platform, b) increases Heap lifetime, c) is fully configurable and easy to use (offering microcoded templates), d) achieves better performance exploiting the presence of the DMC for handling distributed memory requests, thus mitigating processor's workload and e) has a negligible penalty regarding energy consumption.

IV. Conclusions

This paper presented a methodology for enabling custom DMM services for distributed shared memory MPSoC platforms. Experimental results showed that in the proposed memory distribution-aware DMMs the Heap overflow chance is reduced, while the served allocation requests increase with a small penalty in average cycles and energy per DMM event. Specifically for the presented application, the gain was approximately 7× for served allocation requests with a small increase of approximately 14% to average energy consumption per allocation compared to the Pure Distributed Memory organization. Also, the microcode approach is on average 25% faster than the C implementation enhancing the reason for choosing a hardware controller for handling distributed memory requests.

References

[1] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. of DAC*, 2007, pp. 746–749.

[2] F. Catthoor *et al.*, *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.

[3] E. D. Berger *et al.*, "Hoard: a scalable memory allocator for multithreaded applications," in *Proc. of ASPLOS*, vol. 35, no. 11. ACM, 2000, pp. 117–128.

[4] P. R. Wilson *et al.*, "Dynamic storage allocation: A survey and critical review," in *Proc. of IWMM*. Springer-Verlag, 1995, pp. 1–116.

[5] D. Atienza *et al.*, "Systematic dynamic memory management design methodology for reduced memory footprint," *ACM TODAES*, vol. 11, no. 2, pp. 465–489, 2006.

[6] S. Xydis *et al.*, "Custom mutli-threaded dynamic memory management for multiprocessor system-on-chip platforms," in *Proc. of ICSAMOS*, jul. 2010, pp. 102–109.

[7] J. M. Chang and E. F. Gehringer, "A high-performance memory allocator for object-oriented systems," *IEEE Trans. Comput.*, vol. 45, no. 3, pp. 357–366, 1996.

[8] M. Shalan and V. J. Mooney, "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," in *Proc. of CODES*. ACM, 2002, pp. 79–84.

[9] X. Chen *et al.*, "Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller," in *Proc. of DATE*, 2010, pp. 39–44.

[10] M. Monchiero *et al.*, "Exploration of distributed shared memory architectures for NoC-based multiprocessors," *JSA*, vol. 53, no. 10, pp. 719–732, 2007.

[11] A. Bartzas *et al.*, "Software metadata: Systematic characterization of the memory behaviour of dynamic applications," *JSS*, vol. 83, no. 6, pp. 1051 – 1075, 2010, Software Architecture and Mobility.

[12] S. Murali and G. D. Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," *Proc. of DATE*, vol. 2, p. 20896, 2004.

[13] S. Thoziyoor and N. Muralimanohar, "Cacti 5.0, technical report hpl-2007-167, hp labs," 2007.