

# Dynamic Memory Management Customization for Multi-Processor Systems-on-Chip\*

Sotirios Xydis, Iraklis Anagnostopoulos, Alexandros Bartzas, Dimitrios Soudris

School of Electrical and Computer Engineering – National Technical University of Athens, 15780 Zografou, Greece

## 1. Introduction and Motivation

Emerging computing devices integrate multiple services and heterogeneous applications such as multimedia, telecommunication protocols and wireless network communications. Such heterogeneity and computational complexity is benefited from recent advances in technology, which imposed a paradigm shift from uni-processor System-on-Chips (SoCs) to Multi-Processor SoCs (MPSoCs) and Network-on-Chip (NoC) architectures. The increased number of processing elements enables the exploitation of parallelism in coarser levels (thread-level) than the Instruction-Level-Parallelism found in uni-processor SoCs. Thus, multi-threaded applications are becoming increasingly prevalent for the next generation of computing systems. The process of porting multi-threaded applications to NoCs is a difficult task. Furthermore, multiple algorithms that need to run concurrently on such devices impose complex memory access patterns that may result in performance degradation and high energy consumption. For example, in multimedia applications like video games, the unpredicted behavior of the user causes varying inputs which significantly vary the dynamically allocated memory objects, leading to unexpected memory footprint variations unknown until run-time.

Dynamic memory requests are serviced by the dynamic memory manager (DMM). DMMs are responsible for organizing the dynamically allocated data in memory and also servicing the applications memory allocations and de-allocations at run-time. In case of a memory request for allocation of a new object the dynamic memory manager returns to the application the pointer which refers to memory position of allocated object. In C/C++ programming language dynamic memory management is performed through the `malloc/free` and `new/delete` operators. In case of a memory request for deallocation of an already dynamically allocated object, the dynamic memory manager returns to the application either a true or a false value in respect to success of the deallocation process. The DMM is a critical component in NoCs since the dynamic memory allocation often forms the main performance, and scalability bottleneck of multi-threaded applications [3]. Also, it greatly affects the energy and memory consumption of the overall system [1]. Extensive research has been conducted for general purpose dynamic memory allocators targeting either the single processor or the multiprocessor domain [3, 6].

Up until today, a lot of research has been performed in memory analysis and optimization techniques for computing systems to reduce their power dissipation and increase performance [5]. Traditional optimizations use compile-time, mani-

fest information and have focused on static allocation and how to synthesize memory hierarchies for SoCs [4]. For modern dynamic applications this is no longer possible, as the dynamicity of behavior due to the input dynamics cannot be captured by source code analysis alone. Thus, the application behavior and memory requirements significantly vary during run-time. This results that the static allocation of memory leads to an inefficient memory utilization [1]. An extensive survey of the dynamic memory allocation can be found in [6].

## 2. Exploration Methodology and Results

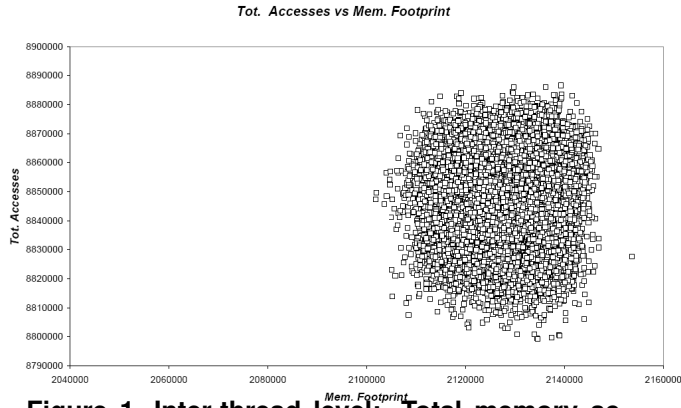
The end goal of exploring the multi-threaded dynamic memory manager (MTh-DMM) design space is to generate Pareto sets of customized dynamic memory managers, tailored to the designers constraints and the applications specific needs. Each possible MTh-DMM solution can be modeled as a different vector of decisions (composed of the leaves of decision trees). In our work, we categorize the existing MTh-DMMs according to their architectural features. Based on this categorization we define an extended design space for platform independent MTh-DMMs. Any known dynamic memory allocator (multi-threaded or not) found in relative literature can be modeled through the proposed design space. The design space is composed of inter- and intra-thread decision trees.

The inter-thread design space decisions are shared since they concern the management of the overall allocated heaps of the MTh-DMM manager in a global manner. On the other hand, intra-thread design space decisions concerns the management of each allocated heap individually. Thus, the inter-thread level includes decisions of globalized and shared policies, while intra-thread level defines heap local customization policies. Since the policy semantics are orthogonal for the inter- and intra-thread design spaces, we partitioned the problem of exploring and designing custom multi-threaded dynamic memory managers into two “constrained-orthogonal” problems.

The first problem refers to the exploration of inter-thread decisions, while the second one refers to exploration decisions available in intra-thread design space. The inter-thread MTh-DMM exploration problem generates a Pareto optimal set of solutions (we consider multi-objective optimization for the MTh-DMMs). These Pareto configurations are propagated as constraints to the intra-thread level exploration. Thus, local heap customization decisions are explored over global Pareto decisions. In this way, we manage to handle the complexity of customizing at the intra-thread level each possible inter-thread configuration. The major steps/flow of the proposed exploration methodology are summarized as follows:

- 1) Given an application, its native source code is annotated with proper profiling constructs that capture the dynamic mem-

\*Partially supported by E.C. funded MOSART IST-215244 Project, [www.mosart-project.org](http://www.mosart-project.org)



**Figure 1. Inter-thread level: Total memory accesses versus memory footprint**

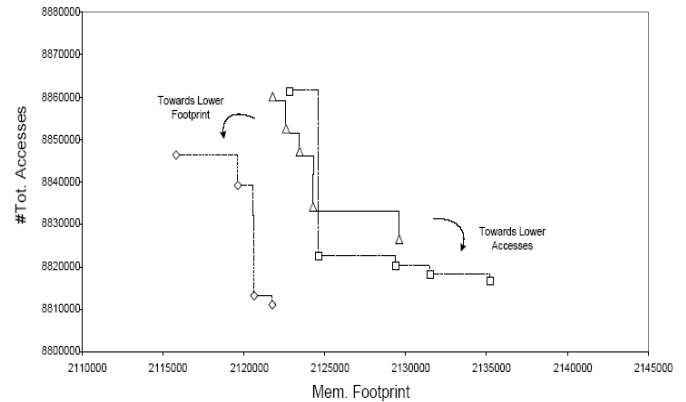
ory behaviour of the application. The software designer performs this task manually.

2) Inter-thread level exploration is then performed in order to automatically generate the source code of valid MTh-DMM solutions. The various MTh-DMMs are linked with the dynamic applications source code and each solution is compiled and evaluated based on various metrics. A Pareto analyzer is invoked in order to extract the Pareto optimal inter-thread level MTh-DMM solutions. These Pareto optimal solutions form the “inter- to intra-thread” level constraints and they are propagated to the intra-thread level exploration.

3) Intra-thread level exploration is invoked customizing each heap individually present in the propagated inter-thread Pareto solutions. Automatic code generation produces valid intra-thread customized MTh-DMM solutions for each Pareto point. The intra-thread level customized MTh-DMMs are linked again with the applications source code and each solution is re-compiled and re-evaluated. The Pareto analyzer is invoked again and the combined intra- and inter-thread level Pareto optimal MTh-DMM solutions are extracted and returned to the designer as the final customized MTh-DMM solutions. Having the Pareto optimal solutions the designer selects the MTh-DMM solution, which satisfies their design constraints.

We experimentally evaluate the proposed exploration methodology and the corresponding tool flow based on a real-life case study of a multi-threaded wireless application [2]. At first, inter-thread level exploration has been performed. A solution space of 67,655 valid and semantically disjoint MTh-DMM configurations has been generated. Figure 1 displays a 2D diagram of the solution space characterized through the number total memory accesses versus the maximum memory footprint needed for each MTh-DMM configuration. Each point corresponds to a unique MTh-DMM solution applied to the network application. From the diagram of Figure 2, only the points delivering the best #Accesses vs. Footprint trade-offs are needed to be further explored. The best trade-off values correspond to the Pareto optimal solution points that are extracted in order to be propagated and guide the intra-thread level exploration. We have developed two heuristic-based intra-thread explorations in order to avoid the evaluation of the enormous design space. Access-oriented exploration heuristic targets to

*Inter-Thread Pareto Curve Shifting Through Intra-Thread Customization*



**Figure 2. Overlapped Inter- and Intra-Thread Level Pareto Curves. Impact of intra-thread level customization**

minimize the memory accesses performed through heap customization (intra-thread level) of inter-thread Pareto solutions. On the other hand, footprint-oriented exploration heuristic targets to minimize the required memory footprint, by properly customizing each heap found into the inter-thread Pareto solutions. Figure 2 depicts the three Pareto curves generated after the implication of each exploration. The customization of inter-thread Pareto solutions is graphically depicted through the inter-thread Pareto curve shifting towards solutions with either less memory accesses (in case of access-oriented intra-thread heuristic) or lower required memory footprint (in case of footprint oriented intra-thread heuristic). For a real multi-threaded network application, the automated tool-flow implementing the proposed methodology offers design time reductions up to 22 days comparing with a hard-coded exploration flow implement from an expert MTh-DMM designer.

## References

- [1] D. Atienza and et al. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Trans. Des. Autom. Electron. Syst.*, 11(2):465–489, 2006.
- [2] A. Bartzas and et al. Enabling run-time memory data transfer optimizations at the system level with automated extraction of embedded software metadata information. In *Proc. of ASP-DAC*, pages 434–439, 2008.
- [3] E. D. Berger and et al. Hoard: a scalable memory allocator for multithreaded applications. In *Proc. of ASPLOS*, volume 35, pages 117–128. ACM, 2000.
- [4] F. Catthoor and et al. *Data access and storage management for embedded programmable processors*. Kluwer Academic Publishers, 2002.
- [5] P. R. Panda and et al. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 6(2):149–206, 2001.
- [6] P. R. Wilson and et al. Dynamic storage allocation: A survey and critical review. In *Proc. of IWMM*, pages 1–116. Springer-Verlag, 1995.