Efficient Memory Allocations on a Many-Core Accelerator

Ioannis Koutras, Alexandros Bartzas, Dimitrios Soudris

School of Electrical and Computer Engineering
National Technical University of Athens
9, Iroon Polytechniou str.
157 80, Athens, Greece
{joko, alexis, dsoudris}@microlab.ntua.gr

Abstract: Memory management is one of the key challenges in the design of embedded systems where memory is a scarce resource. The problem scales disproportionally as new embedded systems incorporate many-core architectures where the cores have to struggle accessing an even more limited amount of resources. In this paper we present a way of creating custom memory allocators for many-core accelerators. We evaluated our approach in the P2012 platform, a many-core accelerator from ST. It is shown that a custom memory allocator created by our framework could save on average 62% of the total cycles spent on memory resource management when compared with the platform’s current memory allocator without increasing the allocator’s overhead.

1 Introduction

The current design trend in System-on-Chips (SoCs) utilizes heavily multiple processors and is therefore shifted towards the Multi-Processor SoC (MPSoC) design paradigm. As technology allows the integration of an aggressively increasing number of transistors, the concept of many-core computing steadily emerges, suggesting tens of processor cores integrated in one chip as a computing fabric [Bor07]. Without any question, memory management on such architectures could contribute notably in improving performance and mitigating the scalability bottleneck, as the processors struggle constantly to access data from shared memory locations. This bottleneck could have an even greater impact on many-core architectures designed for the embedded system context due to the small size of memories used in such hardware.

Applications developed for such systems are slowly adapting to this model while trying to exploit every possible resource by using data- and task-level parallelism. This leads to applications with highly dynamic behaviour and parallel execution of their tasks. This increased dynamism leads to unexpected memory footprint and fragmentation variations, which are difficult to be identified adequately at the design time. Developing dynamic multi-threaded applications using worst-case estimates for managing memory in a static

*This work is partially supported by the E.C funded FP7-ICT-2009-4-248716 2PARMA Project. Official Website: http://www.2parma.eu.
manner would impose severe overheads in memory footprint and power consumption. In order to avoid such type of costly over-estimations, developers are motivated to efficiently utilize dynamic memory [WJNB95].

The dynamic memory manager (DMM) handles memory requests that happen during the application’s execution. Dynamic memory managers are responsible for organizing the dynamically allocated data in memory and also servicing the applications memory requests (allocation/de-allocation) at run-time [WJNB95, BMBW00]. In case of a memory request for allocation of a new object, the dynamic memory manager returns a pointer to the application, which points to the memory position of the allocated object. In C / C++ programming language, dynamic memory allocation is performed through `malloc()` / `new()` function calls. In case of a memory request for de-allocation of an already dynamically allocated object, the DMM returns to the application either a true or a false value in respect to success of the de-allocation process.

In this paper we study memory allocations on a many-core accelerator, namely Platform 2012, and propose a custom based memory management system to improve the performance of these operations.

The rest of the paper is organized as follows. In Section 2 we present previous work for memory management. Section 3 explains the P2012 hardware architecture, as well as the available programming models and execution patterns. Section 4.1 is dedicated to designing memory allocators for this platform and gives a brief overview of the current memory allocator. Experimental results are shown in Section 5. Finally, the conclusions and suggestions for future work are made in Section 6.

## 2 Related Work

Extensive research has been conducted for general-purpose dynamic memory management targeting both the single processor and the multi-processor domain. In [Kri99] a brief overview of the heap management is given and [WJNB95] discusses what policies and search algorithms are optimal for general-purpose use in single-processor systems. In [Iye93] authors are experimenting with maintaining multiple lists, a method which is more suited on multi-processor systems. In [VH99] a simple parallel allocator is implemented, which promises good scalability due to its simplicity, while [Mic04] proposes a way to reduce the synchronization overhead of dynamic memory allocation by using lock-free data structures. Authors in [LK99] benchmark various general-purposed memory allocators on typical server application workloads. Finally, the Hoard multi-threading memory allocator is presented in [BMBW00] aiming for low average fragmentation and little scaling overhead. Essentially, the inherent generality of existing DMMs eliminates the potential for hardware- and application-specific optimizations.

On the other hand, there is plenty of research work regarding memory allocation for specific hardware platforms, which can not be easily applied in other systems. A memory allocator which favours cache locality on specific SMP systems is proposed in [SAN06]. Authors in [BS10] study heap management in the Cell processor, a relevant hardware ar-
chitecture, but they do not handle shared memory; instead of this, the processing units have to handle their own, dedicated memory and they communicate with the system through explicit DMA calls, a limitation posed by the individual hardware platform.

Customized (i.e. application-specific) DMMs have also been proposed as an effective way either to improve performance [BZM01] or to reduce memory footprint [Ati06]. However, DMM customization has been studied only for single-threaded applications running on single-processor platforms. Recently, a systematic exploration strategy was proposed [XBA10], which is performed at design-time in order to customize DMM services to the specific needs of a multi-threaded application. However, the evaluation was performed on a general-purpose machine using a general-purpose POSIX-compliant operating system, which provided the necessary synchronization primitives. In contrast, in this work we are targeting memory management on many-core accelerators with low-level synchronization primitives.

Overall, memory management specialized for minimal memory footprint on a massive number of processing elements is not fully explored in the current bibliography.

3 Architecture Case: ST Platform 2012

Platform 2012 (P2012) is a many-core accelerator developed by ST Microelectronics and CEA. The ultimate goal of P2012 is to fill the area and power efficiency gap between general-purpose embedded CPUs and fully hardwired application accelerators [SC11]. This goal makes this platform fully compliant with the scope of many-core accelerators in which we want to study the memory resource management.

Figure 1: P2012 Cluster as depicted in [SC11]
Processing elements in P2012 are organized into clusters. The organization of a cluster is shown in Figure 1: Clusters are composed by several general-purpose processing elements of the STxP70-v4 architecture, called ENCore processors. There is an option to include specialized hardware IP cores, but this hardware feature, as well as others shown in the Figure 1, are out of the context of memory resource management. Each cluster is managed by a processor unit similar to the ENCore processors in terms of architecture, called cluster controller, and the whole accelerator (also called computing fabric) is managed by another similar processing element, called fabric controller. A first implementation of the platform contains 4 clusters with 16 processing elements, but it is expected to be even more scalable.

Regarding the memory hierarchy, P2012 adopted a multi-level one: There are three levels of memory, namely Level-1, 2 and 3. All the ENCore processors use the same memory map and can access every level. L1 is shared by all the processing elements of each cluster. It is a very fast, yet small (256 kB) memory. It is designed to be accessed in a uniform way, meaning that it is guaranteed to be accessed by ENCore processors in a fixed, low number of cycles regardless of the traffic. L2 is the memory which is shared among clusters in a NUMA way. It is unclear how many cycles it will take for a processing element to access it, as this depends on the traffic of the Network-on-Chip (NoC) which connects the clusters. The size of L2 is currently set to 1 MB, but there are configurations where L2 is completely omitted. Lastly, a portion of the host’s memory is accessible as L3 memory with typical size of 256 MB. The penalty in execution is certainly big whenever a processing element tries to access this memory, so memory accesses from the fabric side to L3 should always be kept minimal.

3.1 P2012 Software Stack

P2012 acts as an accelerator, so the existence of a host system is necessary. Figure 2 shows
the software stack required for deploying an application to an accelerator from a host. The application description and the programming models are the same between the host and accelerator. However, each one of them has a different run-time environment. From the host side is the Operating System (in P2012 Linux or Android are supported) and the accelerator device drivers, and from the accelerator side is the run-time environment of the accelerator. P2012 Software Development Kit (SDK) contains all the required tools of developing applications for P2012, as well as the source code of platform services up to some extent. A user community actively supports the SDK in [Min11].

The application execution is organized as follows: A Linux driver manages the communication between the host and the fabric, i.e. loading the application code from the host to the fabric and managing the P2012 resources from the host side. Runtime services are available by every cluster controller. They are responsible for the application deployment in each ENCore processor and they provide additionally the scheduling and the whole resource management of the cluster in terms of processor and memory resource allocation and de-allocation, as well as of power management. Runtime code is resident to P2012 cluster controllers and its API is used in order to develop programming models.

Applications may be developed currently in two programming models which are available and supported in P2012: Open Compute Language (OpenCL) and Native Programming Model (NPM). The first one is quickly adopted as industry’s standard, as most of the giant hardware companies are active members to Khronos Compute Working Group which is responsible for the development of the official OpenCL specification. NPM is a more device-specific approach to program the platform. It handles concurrency as an Actor model. Other programming models can be implemented on top of NPM; P2012’s OpenCL runtime on the P2012 platform is also built on top of it [SC11].

![Figure 3: Executing an application on P2012](image)

Both programming models utilize P2012 runtime in order to access resources and the low-level task scheduler. In fact, they both use the execution model as shown in Figure 3.
First, the host initializes the fabric and sends the necessary code to be executed. Resource allocation, such as task mapping to specific ENCore processors and memory allocations for processor stacks and data buffers, is the next step in the execution scheme. In fact, resource allocation should be completely finished before the task execution phase begins. During task execution there is no new memory allocation: even the new task instances use the memory space which was previously allocated to the initial identical task. After the end of every task on the ENCore processors, resource de-allocation takes place and P2012 is set off.

4 Creating a custom memory allocator for P2012

4.1 dmmlib Framework

dmmlib is a highly portable dynamic memory management library written in C. It allows developers to generate custom memory allocators by choosing the desired features and policies.

The framework provides custom implementations for dynamic memory allocation, re-allocation and de-allocation which could replace classic system calls, i.e. malloc(), realloc() and free(). The generated functions could be completely standalone if the developer knows a priori the total memory to be managed, or they could use OS calls to access more memory space such as sbrk() or mmap(). Multi-thread dynamic memory management can be supported by using POSIX mutexes or platform-specific synchronization primitives. The memory can be organized in a multiple number of heaps, each of which may contain or not lists of fixed-sized free blocks. There are several implementations of block organizations (singly, doubly linked lists etc.) and a broad variety of search block algorithms is supported (exact-, first-, best-, next-, good-fit etc.) Additionally, there is support for coalescing and splitting of memory blocks to prevent excessive fragmentation. Respective thresholds are provided at design time or runtime for both of the previous mechanisms. Finally, the library supports a rich set of statistics, such as the number of memory accesses and requested allocation sizes, which are fully available at run-time at the expense of some overhead on the metadata structures.

Some of the aforementioned features can only change on design time, so the developer will have to select them before generating the library’s binary code. In a similar manner, other features can be changed during runtime without any need to recompile the library. Most of them are highly parameterizable either statically in design time, or dynamically through knobs at runtime. The perspective of tuning those knobs at runtime in order to create more adaptive allocators is explored more thoroughly in [XSBS11]. In any way, it should be noted that there is a clear trade-off between code size and customizability as the developer chooses to implement features and policies statically or not.

In the context of this work the dmmlib framework was integrated in P2012. The P2012 runtime is targeted, as this is considered the lowest level for resource management inside a P2012 cluster. As a result of the deep integration inside the runtime, every custom
memory allocator generated by dmmlib, supports both NPM and OpenCL. Another benefit of this integration that the usage of the optimized allocator is transparent to the application developers, i.e. not requiring any modifications in the application code.

### 4.2 Original memory allocator

The memory allocator used on P2012 is a variation of Doug Lea’s family of dynamic memory allocators (dlmalloc) [Dou00]. The Lea allocator is considered one of the best overall memory allocators competing even with custom allocators [BZM02].

For every allocation call a memory block is given to the application, containing the requested space bounded by the block’s size information. This helps separating each block and improves operations on the data layout such as coalescing blocks.

Free memory blocks are maintained on circular doubly linked lists: each free block contains pointers to its previous and next in the list block inside the space destined for application data. There are 128 of free lists containing blocks according to their size, e.g. there are lists for blocks of 16, 32 bytes and so on. The maximum number of the blocks which can be included in these lists is getting lowered logarithmically as the size of blocks is being raised.

Blocks are sorted and searched through as in LRU or FIFO allocation with support to coalesce and split blocks.

### 4.3 Choosing options for a dmmlib memory allocator on P2012

dlmalloc is a very efficient general-purpose memory allocator, but it is not optimized for the current application execution scheme on P2012. In order to investigate further the integration of dmmlib in P2012, a custom memory allocator was generated with features and options we considered optimal for this platform.

The main reasoning for most of the choices was to achieve simplicity as the target platform is an embedded one. More specifically, we have chosen block structure identical to the dlmalloc’s one in order to prevent metadata overhead on this low-memory environment. We have tried to achieve simplicity in the lists structure. Thus, we are using circular singly linked lists instead of using circular doubly linked lists, achieving a reduction in memory footprint compared to the original allocator. Furthermore, instead of using a great number of lists for free blocks, we choose to use only one such list. This simplifies a lot the procedure of finding an appropriate free memory block big enough to accommodate the allocation request. Finally, we have disabled coalescing and splitting, further simplifying the `malloc()` and `free()` calls, thus resulting in performance gains.

The resulted memory allocator is an extremely simple one, which fits in the current execution flow of P2012. It would cause fragmentation issues on applications which require executing various tasks on different times, but the current implementations of program-
ming models do not create such tasks or handle the resources this way.

5 Experimental Results

5.1 Experimental Setup

In order to compare the proposed custom memory allocator with the current one, it is necessary to run cycle-accurate simulations for multiple applications running on P2012. RTL simulation was chosen since NoC traffic and memory access timings are the most accurate ones in this type of simulation.

RTL simulation is one of the most accurate ones, but it is also one of the slowest. All of the tested applications use one cluster, so simulating just one cluster is adequate for the benchmarking needs. Even then, simulating one cluster in RTL requires a tremendous amount of time even when running simple applications. In order to improve the simulation time, the applications were simplified: Since all of the memory allocations and de-allocations take place in the cluster controller, we do not need to enable the ENCore processors. The memory allocation and de-allocation traces of applications were extracted and simple test-benches were created, where only these types of operations were used.

We will show results from the following applications, all of which are available through the official P2012 SDK [Min11]:

Integral Calculates the integral of an input matrix.

Gaussian Applies a Gaussian blur effect on an input picture.

FAST Features from Accelerated Segment Test, a corner detection algorithm implementation used on computer vision.

Matrix Performs matrix multiplication of an input matrix with a constant matrix.

5.2 Allocators’ Evaluation

In terms of code size the simplicity of the generated allocator makes a big difference compared to Doug Lea’s: 60 kB versus 88 kB, a 32% improvement on the code size.

Metadata (i.e. internal heap organization and usage information and statistics) is essentially the same as the block structure is kept intact: the block size information appears in the memory twice, before and after the space for application data. This is equivalent to 64 bits per block, as the size information in dmmlib takes up the size of a word and P2012’s ENCore processors use 32-bit words.

The speed-up of using the custom memory allocator over Doug Lea’s is also substantial. Figure 4 shows an average improvement of 60% percent in cycles for executing all the
memory allocations required by each application. In Figure 5 the memory de-allocations have been included in the total time, but they have very little affect on the final outcome. Tables 1 and 2 present a clear view on the average and the max time each allocator spends for one event and the respective differences.

The trend we could extract from these timings is that applications which use many small blocks (e.g. the Matrix application) are more benefited by the customized memory allocator of dmmlib.
5.3 Estimating memory allocations on more dynamic task execution cases

It is quite certain that it will be possible to have more complex task execution flows as P2012 matures. Resources would then have to be de-allocated and re-allocated in a more fine-grained way, i.e. between task executions and not just after the last task execution of the application. In respect with this, we need to test the memory allocators in use cases where data is allocated and de-allocated in a more dynamic way.

There are still not any available applications matching this behavior. What we propose in order to estimate the performance of memory allocators in such cases is to use the traces of the previous applications and to randomize the spots of data de-allocations. The allocation dependencies will still be respected (e.g. we can not de-allocate memory which is not allocated before) and the memory requests will be realistic enough.

For this part we have used the FAST and Matrix applications since the tasks of these applications is more near to real application kernels. In Figure 6 there is an overview of the total cycles spent on memory management operations. The custom memory allocator still outperforms the original one, but the gap is getting much more narrow. Table 3 and 4 show the average and worst cases of timings for allocating and de-allocating data. It should be noted that the number of allocations for the FAST application are less than the number of allocations for the Matrix, as the FAST application performs fine-grained memory allocations inside the kernel. As a result, the customized memory allocator seems
to perform better in the case of the FAST application over the Matrix one.

6 Conclusions and Future Work

Our approach generating customizable memory allocators allows us to speed up the exploration process of finding an efficient memory allocator for many-core architectures such as P2012. The custom memory allocator we proposed for P2012 achieves a speedup of 2.6x in cycles without compromising the metadata overhead introduced by the original memory allocator of P2012 runtime.

As the P2012 evolves and other many-core accelerators are introduced, we can expect multiple use-case scenarios and increased interaction between these systems and their surrounding environment. With regards to that, new features for memory management could be introduced, such as runtime, adaptive control of features like coalescing and splitting, support to allocate and de-allocate memory regions by the worker processors as well, and ability to perform fine-grained memory management on dedicated memory regions.

Table 3: Average and Maximum values of Allocation Times for more dynamic cases

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>initial</td>
<td>549</td>
<td>-29.14%</td>
<td>1153</td>
<td>-32.35%</td>
</tr>
<tr>
<td></td>
<td>custom</td>
<td>389</td>
<td></td>
<td>780</td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td>initial</td>
<td>583</td>
<td>-6%</td>
<td>1184</td>
<td>-9.62%</td>
</tr>
<tr>
<td></td>
<td>custom</td>
<td>548</td>
<td></td>
<td>1310</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Average and Maximum values of De-allocation Times for more dynamic cases

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FAST</td>
<td>initial</td>
<td>347</td>
<td>-38.69%</td>
<td>565</td>
<td>-12.55%</td>
</tr>
<tr>
<td></td>
<td>custom</td>
<td>225</td>
<td></td>
<td>481</td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td>initial</td>
<td>353</td>
<td>-34.28%</td>
<td>617</td>
<td>-21.07%</td>
</tr>
<tr>
<td></td>
<td>custom</td>
<td>232</td>
<td></td>
<td>487</td>
<td></td>
</tr>
</tbody>
</table>
References


