Energy-Efficient Dynamic Memory Allocators at the Middleware Level of Embedded Systems

Stylianos Mamagkakis [1,3], David Atienza [2], Christophe Poucet [3], Francky Catthoor [3] and Dimitrios Soudris [1]
[1] VLSI Center-Democritus Univ., 67100 Xanthi, Greece.
[2] DACYA/UCM, Juan del Rosal 8, 28040 Madrid and LSI/EPFL 1015-Lausanne, Switzerland.
[3] IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
F. Catthoor also professor at ESAT/K.U.Leuven-Belgium.
{smamagka, dsoudris}@ee.duth.gr, david.atienza@epfl.ch, {mamagka, poucet, catthoor}@imec.be

ABSTRACT

The next generation of embedded systems will be dominated by mobile devices, which are able to deliver communications and rich multimedia content anytime, anywhere. The major themes in these ubiquitous computing systems are applications with increased user control and interactivity with the environment. Therefore, the storage of dynamic data increases, thus making the dynamic memory allocation of heap data at run time a very important component with heavy energy consumption. In this paper, we propose a novel script, which heavily customizes the dynamic memory allocator according to the target application domain and the underlying memory hierarchy of the embedded system. The dynamic memory allocator resides in the middleware level or in the Operating System level (whenever it is available). The result of our script and automated tools is the reduction of energy consumption by 72% on average and the reduction of the execution time by 40% on average, which is demonstrated with the use of 1 real life wireless network application and 1 multimedia application.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; D.4 [**Software**]: Operating Systems; E.1 [**Data**]: Data Structures

General Terms

Design, Performance

Keywords

Middleware, Dynamic Memory Allocation, Heap Data, Embedded Systems, Low-Energy Consumption

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

1. INTRODUCTION

In the last years there has been a trend towards embedded consumer devices to achieve the paradigm of rich multimedia content anytime, anywhere. These systems implement complex multimedia applications (e.g., MPEG21) and wireless network protocols. Increased interaction with the user and the environment have increased the demand for Dynamic Memory (DM from now on) allocation subsystems, which can cope with the variation in run-time memory needs (e.g., scalable frame input and unpredictable network traffic) of the aforementioned applications in the most efficient way. Inefficient DM allocation support leads to decreased system performance and increased cost in energy and memory footprint due to increased memory allocation and access needs.

The basic functions of a DM allocator are the allocation and deallocation of memory blocks. Allocation is the mechanism that searches for a block big enough to satisfy the request of a given application and deallocation is the mechanism that returns this block to the available memory of the system in order to be reused later. In real, dynamic applications the blocks are requested and returned in any order, thus creating "holes" among used blocks [3]. These holes constitute what is known as memory fragmentation. Therefore, a successful DM allocator has to prevent or deal with fragmentation in an efficient way or else it will run out of memory.

Many DM allocation solutions are available today for general purpose systems. These are activated with the standardized malloc/free functions in C and the new/delete operators in C++. Support for them is already available at the Operating System (O.S.) level [3]. Each of these DM allocators provides a general solution that ignores the special de/allocation behavior and fragmentation outlook of the application that needs them or the underlying memory hierarchy. The same approach is followed in embedded system designs, which rely solely on their embedded O.S. (e.g., Enea OSE [9] or uCLinux [8]) for DM allocation support.

The use of O.S. based, general-purpose DM allocation usually has an unacceptable overhead in embedded designs, considering the limited resources and hard real-time constraints of embedded systems. Therefore, in order to achieve better results, application-customized DM allocators are needed [4, 6]. Note that they are best realized in the middleware and not in the platform hardware which would require undesired platform changes for each application (domain) target. Furthermore, parameterized configurations of the customized DM allocation designs are needed in order to give wider design choices and enable exploration of the available trade-offs (i.e., energy, memory footprint, performance). This exploration in-

cludes pruning of the huge parameter space and provides only the interesting Pareto-optimal configurations. These configurations are considered Pareto-optimal if no other configurations exist which have better results in both metrics explored (e.g., memory footprint and accesses) [5].

The exploration of different DM allocator designs is a very big design effort (both in complexity and required time). Without a systematic design flow and tools the embedded system designer is not able to correctly profile dynamic applications and arrive at an efficient DM allocation design. Furthermore, he is not able to explore the DM allocation parameters, due to the fact that thousands of different DM allocator configurations are possible. Finally, simulation and implementation support for different memory hierarchies utilized by the DM allocator would prove to be challenging and very time consuming.

In this paper, we introduce a novel, systematic script (extended from our earlier publications) and tool support to automatically create and explore the trade-offs in the DM allocation parameters in order to provide energy-efficient designs. With our new fully automated technique we generate energy-efficient DM allocator configurations for the embedded system designer to use according to the application's specific needs. Finally, we have implemented this technique to two new dynamic multimedia and wireless network applications. The paper is organized as follows. In Section 2, we describe some related work. In Section 3, we detail the main causes of memory consumption by the DM allocator. In Section 4, we briefly summarize our script and its different steps. In Section 5, we present the parameters which are explored automatically in the DM allocators with the use of our tool. In Section 6, we describe the memory hierarchy architecture of current embedded platforms that we consider available for our script. Later, in Section 7, we introduce our real case studies and present the experimental results obtained. Finally, in Section 8, we draw our conclusions.

2. RELATED WORK

Currently, there are many O.S. based, general-purpose DM allocators available. Successful examples include the Lea allocator in Linux based systems [3], the Buddy allocator for Unix based systems [3], and variations of the Kingsley allocator in Windows XP or CE [11, 10], FreeBSD, and Symbian based systems. Slightly different versions of these allocators exist for their embedded system O.S. counterparts and they provide reasonable performance and defragmentation support for some applications (e.g., Enea OSE [9] and uCLinux [8]). In contrast with these 'off the shelf' DM allocation solutions, our approach provides highly customized DM allocators, fine tuned to the embedded application. For example, instead of using a generic O.S. based DM allocator for 2 different applications, our approach provides 2 highly specialized energy-efficient DM allocators.

Also, other frameworks exist to customize allocators to meet specific DM allocation needs. In [12], a DM allocator that allows defining multiple memory regions with different disciplines is presented. However, this approach cannot be extended with new functionality and is limited to a small set of user-defined functions for memory de/allocation. In [6], the abstraction level of customizable memory allocators has been extended to C++. Additionally, the authors of [4] propose an infrastructure of C++ layers that can be used to improve performance of general-purpose allocators. Finally, work has been done to propose several garbage collection algorithms with relatively limited performance overhead [13, 14]. Contrary to these frameworks, which are limited in flexibility, our approach is systematic and is linked with our tools, which automate completely the process of custom DM allocator construction. Also, we provide additional customization support for energy efficiency, which is very important for embedded systems, on top of the usual performance and memory footprint metrics.

In addition, research has been performed to provide efficient hardware support for DM allocation. [15] presents an Object Management Extension (i.e., OMX) unit to handle the de/allocation of memory blocks completely in hardware using a variation of the classic binary buddy system. [16] proposes a hardware module called SoCDMMU, which tackles the global on-chip memory de/allocation to achieve a deterministic way to divide the memory among the processing elements of SoC designs. However, the operating system still performs the management of memory allocated to a particular on-chip processor. All these proposals are very relevant for embedded systems where the hardware can still be changed, while our work is intended for any fixed embedded design architecture, where customization can only be done at the O.S. or software level.

In contrast to our previously published work [1], which focuses on the definition of the design space and abstract implementation features, mostly, for lower memory footprint (i.e., fragmentation), in this paper we focus on the fully automated exploration of the implementation parameters and the development of an energy efficient design. With the use of our tool [2], we automatically explore a number of trade-offs between memory accesses, memory footprint, energy consumption, and performance instead of focusing in just a single metric. We prove that the achievable tradeoff ranges are big enough to justify the exploration of the parameters of the DM allocators. Finally, we exploit these trade-offs in order to achieve the most energy efficient design (at the expense of memory footprint).

3. ENERGY CONSUMPTION IN DYNAMIC MEMORY ALLOCATION

As briefly mentioned in Sect. 1, the main function of the DM allocator is to allocate and de-allocate memory blocks at run-time, in order to satisfy the memory requests of the applications. The main concern of the DM allocator is to manage the memory space as efficiently as possible. Immediately, it becomes clear that this goal deals with minimizing memory fragmentation. Memory fragmentation is divided into internal and external memory fragmentation:

- On the one hand, when the application requests a memory block from the DM allocator, which is smaller than the memory blocks available to the allocator, then a bigger block is selected from the memory pool and allocated (as shown in the upper part of Fig. 1). This results in wasted memory space inside the allocated memory block. This space is not used to store the application's data and can not be used for a future memory request. This is called internal fragmentation, which is common in requests of small memory blocks [3].

- On the other hand, when the application requests a memory block from the DM allocator, which is bigger than the memory blocks available to the allocator, then these smaller memory blocks are not selected for the allocation (because they are not continuous) and become unused 'holes' in memory (as shown in the lower part of Fig. 1). These 'holes' among the used blocks in the memory pool are called external fragmentation. If they become too small, then they can not satisfy any request and they remain unused during the whole execution time of the application. External fragmentation becomes more evident in requests of big memory blocks [3].

Therefore, all the design effort of the DM allocator usually focuses on preventing or dealing (if it can not be prevented) with

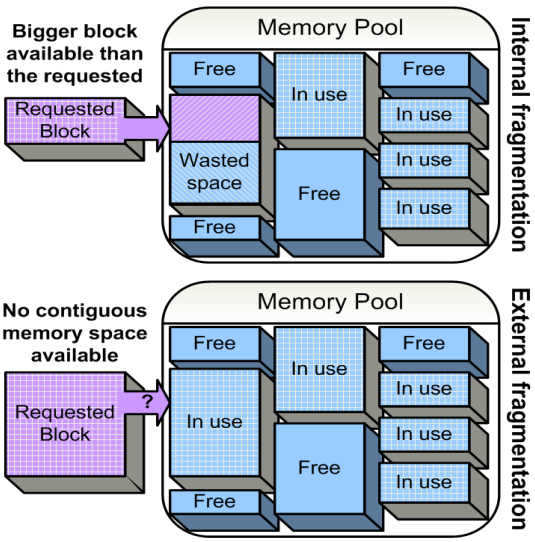

Figure 1: Internal and External Memory Fragmentation

internal and external fragmentation. What is not always considered in most embedded system designs, is that all the DM allocators themselves play a significant role in the energy consumption of the memory in the embedded system. There are three factors of the DM allocator design that negatively influence the energy consumption of the memory subsystem:

- By increasing the memory accesses: Each access to the physical memory by the DM allocator increases the total energy dissipation of the memory. Increased memory accesses mean increased energy consumption. When the application requests a memory block from the DM allocator, then the DM allocator has to search inside its pools of free blocks to find a free block of the requested size. The way that the pools are structured can make it very hard for the DM allocator to find this memory block, thus increasing the memory accesses. For example, it is easier to find a 10-KB memory block if you have separated the blocks into one pool for blocks smaller than 20-KB and one for blocks bigger than 20-KB, instead of having a single pool for all the blocks. Also, the way that the DM allocator searches inside a pool can affect the number of memory accesses it needs before it finds the correct-sized free block. Finally, assisting de-fragmentation mechanisms (like splitting blocks and coalescing blocks) increase the memory accesses of the DM allocator.

- By increasing the memory footprint: Bigger memories increase the energy cost per memory access. Therefore, if the used memory footprint is bigger, it must be mapped on bigger, more energy-hungry physical memories. It is clear that the memory allocated by the DM allocator has a bigger memory footprint if the DM allocator can not prevent (or fix) internal and external fragmentation. Also, the structure of the DM allocator itself contributes significantly to the memory footprint. For example, a header for each block that records the block size (e.g., for splitting purposes) can be up to 4 bytes. If the applications requests blocks that are smaller than 12 bytes, then the DM allocator's memory footprint overhead is at least 25%.

- By wrong memory assignment: The data, that is mapped on the bigger off-chip physical memories in a memory hierarchy, consumes much more energy per access than the data that is mapped on the smaller on-chip memories of a memory hierarchy. Therefore, if the data that is accessed most by the application is not mapped in the less energy-hungry physical memory, then the total energy consumption of the memory increases dramatically. The DM allocator is responsible for allocating all the data to specific memory blocks. In order to minimize the energy consumption, the DM allocator must assign the memory blocks to each memory component of the memory hierarchy according to the data that they hold and thus according to the frequency of the application accesses to this data (i.e. according to the 'popularity' of each block).

It becomes apparent that a certain DM allocator design can frequently affect one or more of these factors in a negative way, thus increasing energy consumption. Even worse, sometimes one factor can be in conflict with another factor (e.g., minimizing memory accesses results in increasing the memory footprint), thus not making it always clear which DM allocator design is better for energy reduction. In this paper, we are going to balance these two factors in order to achieve the most energy efficient DM allocator design. We are going to balance them by exploiting the memory accesses versus memory footprint trade-offs, which we will provide with our Pareto-optimal designs.

4. TOOL AND SCRIPT OVERVIEW

Our tool and script enable the quick and efficient design of custom parameterized DM allocators for new dynamic embedded applications. It consists of four steps applied to each dynamic application (as shown in Figure 2). The first step is the profiling of the dynamic run-time behavior of the application. In the second step, we provide the general design guidelines, according to our profiling results, for an abstract DM allocator for our application. In the third step, our tool explores the various possible configurations of the abstract DM allocator designed in the previous step. The various configurations can be explored with the alteration of the values of the parameters introduced in this paper. Finally, in the fourth step, our tool systematically explores a set of Pareto optimal DM allocator configurations to be used by the embedded system designer in order to achieve the most energy efficient design. Because our methodology is systematic, we were able to develop the tools that support our approach and enable the automation of the whole optimization process. The first 2 steps are reused from earlier work by us [1]. In this paper, we introduce the 3rd and 4th novel steps in our methodology and link them fully with the automated tools [2] to support the parameter exploration and final implementation in any possible memory hierarchy of our platform. The whole framework enables us for the first time to automatically and heuristically explore thousands of different custom DM allocators, instead of just a couple of DM allocators formerly available in our libraries, for real-life embedded applications, and to be able to arrive at truly energy efficient solutions.

5. PARAMETERIZED CUSTOM DM // ALLOCATION SCRIPT AND TOOLS

In this section we first briefly explain steps one and two (they are analyzed in detail in [1]). Then, we extensively analyze the parameters that are explored in the 3rd step and the method of exploration. Finally, we explain the 4th step, which is the selection of the Pareto-optimal configurations of DM allocators.

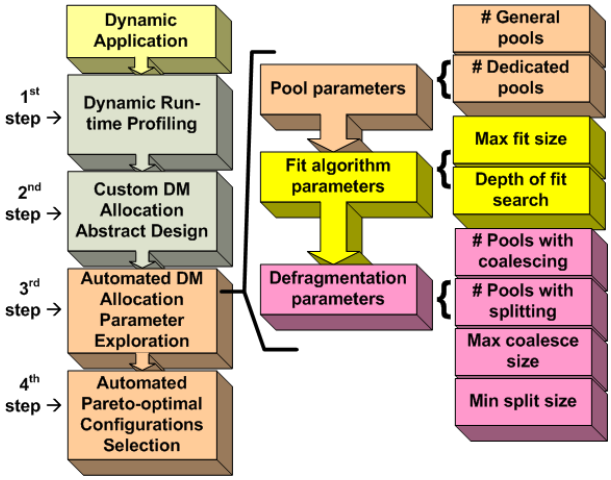

Figure 2: Script flow and parameters

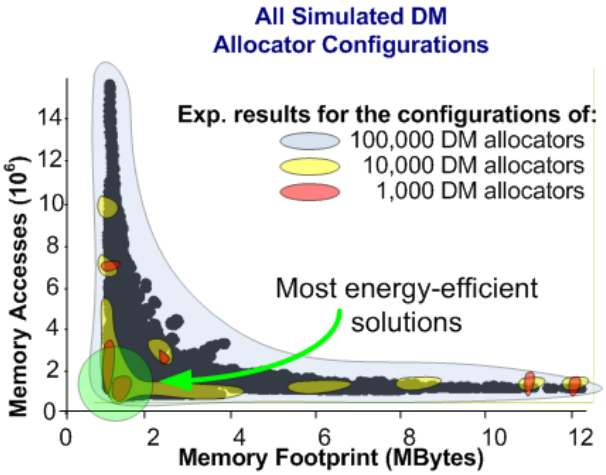

Figure 3: Memory accesses and memory footprint Pareto curve for Easyport buffering running for 12000 packets

5.1 Dynamic Run-time Profiling

The first step of our approach is the profiling of the run-time de/allocation behavior of the dynamic application under study. We group the profiling data in sets according to the requests for blocks that have the same size. In this way, we have a set of profiling data for each one of the different-sized block requests. The profiling data includes the average number of allocations and de-allocations per different-sized block. It also includes the average lifespan of the allocated blocks and the maximum number of same-sized blocks allocated at any time [2]. It provides a lot of information about the run-time memory footprint needed by the application and the timing of the allocations and de-allocations. All this profiling information will help us to design the custom DM allocator in the second step and focus our parameter exploration around certain values in the third step. More details about the profiling tools designed to assist the parameter exploration are analyzed in subsection 5.4.

5.2 Abstract Design of Custom DM Allocator

The second step of our approach is the abstract design of the custom DM allocator. In this phase we evaluate the general design decisions of our DM allocator according to the profiling data gathered in the first step. For example, we evaluate the design of the internal blocks that the DM allocator is going to use, the pools that these blocks are going to be clustered in, whether we are going to use defragmentation mechanisms like block coalescing and block splitting, which fit algorithms to use and the order of the blocks within the pools (detailed information can be found in [1]). These decisions provide the template of the DM allocator, but they do not indicate specific implementation characteristics. For example, a decision might be taken in the second step to use defragmentation mechanisms, but the designer will not know exactly how much to defragment. The answer to this question involves a lot of trade-offs and is explored with the use of our script in the third step.

5.3 Automated DM Allocator Parameter // Exploration

The third phase of our approach is the automated DM allocation parameter exploration. In this subsection we first analyze the relevant DM allocator parameters and then the method that our tool uses to explore them automatically. We are going to show that different configurations of DM allocator parameters will give very big trade-offs between memory footprint, memory accesses, energy consumption and performance. We can exploit those trade-offs in order to achieve the most energy efficient design. The relevant DM allocator parameters can be divided into three main categories (as shown in Figure 2):

1.- The *Pool parameters*, which consist of the *Number of general pools* and the *Number of dedicated pools*. These DM allocator parameters are related to the abstract pools of memory blocks inside the DM allocators. All the modern DM allocator designs can feature two types of pools to group memory blocks: At least one "general pool" with memory blocks of various sizes and a variable number of "dedicated pools" (e.g., 0-32) to accommodate specific dominant block sizes, frequently requested by the application. The role of the different pool configurations is to prevent fragmentation.

The *general pools* are heaps with a pre-allocated number of free blocks which are created from the initialization of the DM allocator. The *dedicated pools* are freelists which don't release the freed memory blocks back to the system, but keep them reserved for future allocations. Each one of these two parameters can be set to zero, which means that this type of pool will not be used at all, or they can be set to any number to try to prevent fragmentation at various levels of effectiveness. These parameters are of great importance because they heavily affect all the explored metrics.

2.- The *Fit algorithm parameters*, which consist of the *Max fit size* and the *Depth of fit search*. The second set of DM allocator parameters is related to the fit policies. Their role is to prevent fragmentation. Therefore, according to the fit policy used, the DM allocator searches the blocks inside a pool to see if they fit (i.e., to see if they are big enough to satisfy the request of the application).

The first of these parameters regards the maximum block size, that has to be found with a fit policy, in order to qualify for a sufficient fit (e.g., if the application requests a 10 KB block, a 150% *Max fit* would mean that up to a 15 KB block is considered a good fit for the DM allocator). The second implementation parameter regards the number of memory blocks that should be searched inside a pool to find a memory block of sufficient size (e.g., for a pool with 200 blocks, a 25% search means that only the first 50 blocks should be searched to find a sufficient sized block for a request). These DM allocator parameters are important as well, because they mostly affect the number of memory accesses of the DM manager and the run-time performance.

3.- The *Defragmentation parameters*, which consist of the *Num-*

ber of pools with coalescing and splitting, the *Max coalesced size* and the *Min split size*. The third set of DM allocator parameters is related to the coalescing and splitting mechanisms of the DM allocator. The role of these mechanisms is to try to reduce fragmentation. This means that they can merge two smaller blocks into a larger one to satisfy a big request (and thus reduce external memory fragmentation). Similarly, they can split one larger block into two smaller ones to re-use the smaller block in another request (and thus reduce internal fragmentation).

The first two of the parameter sets define the number of pools which include coalescing and splitting support. Additionally, the size of the minimum and maximum block sizes that can be produced by a split or merge are implementation parameters (e.g., a 64 KB *Max block size* means that two adjacent blocks in memory of 50 KB and 20 KB can not be coalesced because their sum would be bigger than 64 KB). The different maximum and minimum block sizes can adjust the level of defragmentation use and effectiveness of our DM allocator. In general, increase of these DM allocator parameters tend to reduce memory footprint and increase memory accesses.

After the second step (i.e., abstract design of custom DM allocator), our tool is ready to explore the various number of DM allocator configurations in the third step. This is done by altering the values of the parameters in the custom DM allocator. It is clear that the number of different configurations can easily explode. We have introduced 8 different parameters and by using combinations with even just 3 different values for each parameter, we already end up with 6561 different configurations. Obviously, there is no way for an embedded system designer to create, implement and test so many DM allocator configurations without the full automation support that we provide for the first time in this paper. The depth level of the exploration can be adjusted by the designer according to his design time budget. For example, as shown in the upper part of Figure 3, we provide the experimental results for 3 different depth levels of exploration (exploration of 100000, 10000 and 1000 different DM allocators). We evaluated that by adjusting the depth level of explorations by one order of magnitude, we manage to obtain up to 10% further reductions for each evaluated metric. The exploration can be guided heuristically to reduce the number of the total simulations by using the guidelines in [1].

We have developed a framework to automatically create, map in the memory hierarchy and compile any number of DM allocator configurations (lower right box in Figure 4). The only input that must be given to our tool are the arrays with the parameter values that we want to explore for the different configurations (lower left box in Figure 4). The upper left box in Figure 4, shows that we can implement the DM allocator configurations in any memory hierarchy, just by telling to the tool which pools should be mapped in every level of the memory hierarchy. For example, we only have to declare that one dedicated pool for 74-byte blocks should go to the L1 64 KB scratchpad memory, one general pool and one dedicated pool for 1500-byte blocks should go to the 4 MB main memory. Our tool will take care of the implementation of the DM allocator to support the mapping of these pools in the corresponding memory hierarchy. In fact, we have developed a C++ library with more than 50 different modules (upper right box in Figure 4), which can be linked in any way with the use of templates and MIXINS inheritance to create custom DM allocators. The tool works in a plug-and-play manner and there is no need to alter the dynamic application's source code, which calls the appropriate DM allocator from the library.

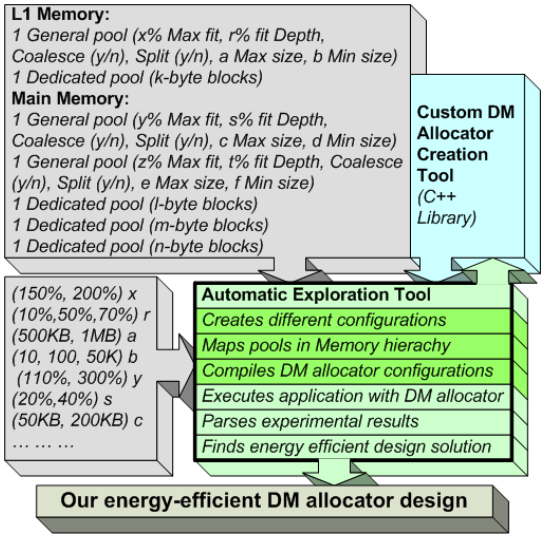

Figure 4: Tool flow for automated parameter exploration

5.4 Automated Pareto-optimal Configurations Selection

The final step of our script is the automated selection of Pareto-optimal configurations. Because we now need absolute numbers, this step in our script becomes partly platform dependent. We indeed need some assumptions concerning the data memory organization to compute the quantitative data (see also Section 5). This step involves the simulation (i.e., execution) of our dynamic application for each one of the different DM allocator configurations (lower right box in Figure 4). These configurations were already defined, constructed and implemented automatically in the previous step. In the context of evaluating the simulations, we have extended the profiling tools, introduced in Subsection 5.1. Thus, we can test and profile all the different DM allocator configurations for the defined memory hierarchy, and get results for mem. accesses, mem. footprint and energy consumption for each level of the memory hierarchy. The results are provided either on a GUI (see Figure 5) or in a format easy to import to Excel or Gnuplot. With these results, the embedded system designer can construct different graphs and tables with the various metrics that are important for the embedded system (e.g., energy consumption vs execution time as shown in the Figure 8 and the experimental results shown in Figure 7). In practice, the entire Pareto set can contains quite a large number of points (thousands) so the tool allows the designer to effectively prune this to the parts of the Pareto range that is of most interest. Then, the Pareto-optimal curves to evaluate the trade-offs of the configurations (as shown in Figure 8) can be provided automatically with the use of our tool (lower right box in Figure 4). The tool (written in Perl and OCAML) parses all the experimental results data and provides Pareto-optimal curves for the chosen metrics. We should note the significance of our quick parsing (less than 20 seconds) of the profiling data, which can reach even Gigabytes for one single configuration.

The memory footprint measured in this step is the memory reserved by the DM allocator to satisfy the requests of the application and its internal mechanisms. The memory accesses counted consist of those used by the DM allocator to allocate the dynamic memory, as well as the application's accesses to this dynamic memory. No accesses to instruction memories are measured in this paper since we do not focus on any concrete architecture and moreover the data

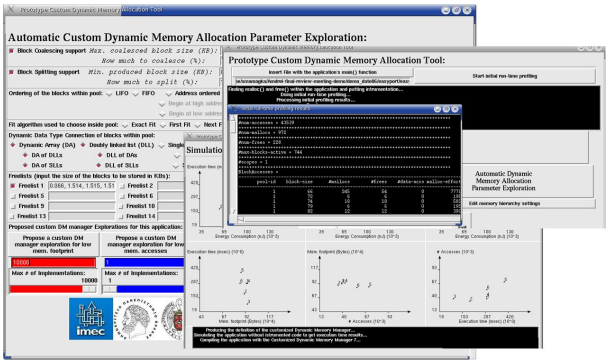

Figure 5: Graphical User Interface of the automation tool

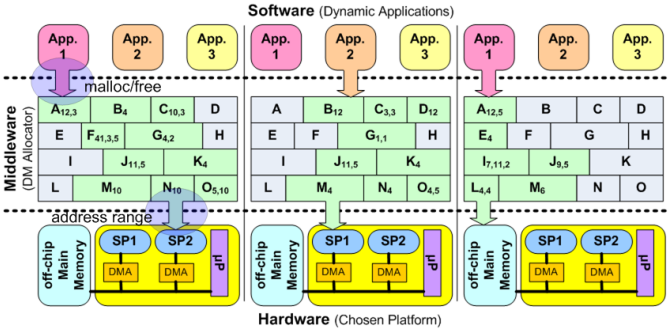

Figure 6: Our custom Dynamic Memory Allocator realized in the Middleware

accesses are the most important factor to optimize for new dynamic applications [7]. Finally, the energy estimations are made with the use of an updated version of the CACTI model [17]. This is a complete energy/delay/area model for embedded SRAMs that depends on memory usage factors (e.g., size, internal structure or leaks) and factors originated by memory accesses (e.g., number of accesses or technology node used). We use the .13 μ technology node. Note that any other model for a specific memory hierarchy can be used just by replacing this energy estimation module in the tools.

6. EVALUATED EXPERIMENTAL PLATFORM

In modern embedded systems, due to their energy-efficient nature, the most typical memory architecture available consists of several memory levels (e.g L1, L2, etc.), which makes profit from the locality of memory accesses in real-life dynamic applications (as shown in figure 6)):

- First, a main on-chip memory where the main part of DM is de/allocated (also called in the DM context as main heap). This part of the DM memory is cacheable; thus, hardware-controlled caches can be used to optimize the accesses to the most frequently accessed locations of DM in smaller on-chip memories very close to the processor, as with statically allocated data. A discussion of the influence of different caches policies for DM locality is outside the scope of this paper and more details can be found in our previous publications. In any case, our DM allocation script achieves significant reductions in the overall memory accesses (see Section 7 for Pareto-optimal results in real, dynamic applications). Hence, all types of cache mechanisms and optimizations benefit from our script for DM allocation and are complimentary to our work. So we can claim that our optimization can be considered as a precompiler step that is fully platform-independent and aims at the middleware of our embedded system design.

- Second, software controlled on-chip SRAM memories (i.e., scratchpad memories) are available to the DM manager in order to directly de/allocate parts of the DM (i.e., additional auxiliary heaps) in small on-chip memories instead of using on-chip hardware controlled caches. The DM in scratchpads is not cacheable and copies in main memory do not exist. Therefore, a different address-range than the main memory is devoted to the scratchpad memories.

In the Pareto-space explorations shown in the experimental results of this paper we consider one level of on-chip hardware-controlled cache and main memory sizes in the range of 1 MB to 8 MB, to be closer to current embedded devices. The scratchpad memories can vary between the sizes of 4 to 128 KB. To limit the exploration time for realistic size candidates of memories, multiple-of-two sizes are the only explored candidates (e.g., 4, 8, 16, etc. in scratchpads). However, the number of possible solutions in the implementation of the DM allocator is large and can easily be covered with our proposed automated flow in a few hours (see Section 7). Moreover, note that the modifications of the memory sizes and number of memory layers in the memory hierarchy to be explored with our tools (in case any other concrete memory hierarchy of embedded systems needs to be evaluated) requires a very limited effort (i.e., 1-2 days for a PC workstation running simulations without the need of human interaction). Note that in the case of trying to do the same without our proposed script and automated tools, it would take a skilled embedded systems designer more than a month.

7. CASE STUDIES AND EXPERIMENTAL RESULTS

We have applied the proposed script and automated tool support to two real case studies that represent different wireless network application and digital processing domains:

7.1 Easyport buffering application

The first case study presented is the Easyport wireless network application produced by Infineon [18]. Easyport features packet and ATM cell processing functionality for data and voice/data Integrated Access Devices (IADs), enterprise gateways, access routers, and Voice over IP (VoIP) gateways. Easyport dynamically allocates the packets it receives from the Ethernet channels in a memory before it forwards them in a FIFO way. In the context of the parameterized DM allocator experimental results, we emulated the aforementioned buffering of the packets, which is the most memory intensive function of the Easyport application. To run realistic simulations of Easyport, we used a variety of typical packet traffic traces provided by Infineon.

Using the first step of our script, we profile the Easyport application. The dynamic behavior profile shows that Easyport uses a big variety of block sizes (up to 59 different sizes) ranging from 66 bytes (i.e., acknowledgement packets) up to 54 KB. The memory is allocated in bursts and most of the data remains allocated for a very short period. Additionally, the maximum number of concurrently allocated blocks is 647. Finally, the blocks mostly allocated are the 1514-byte blocks (38% of total allocations on average) and the 66-byte blocks (22% of total allocations on average).

Using the second step of our script, we design an abstract custom DM allocator for our application. Following the methodology described in [1], we decide to use multiple block sizes for our allocator (because we have 59 different requested sizes), to have some

pools per block size (because we want to prevent some fragmentation and take advantage of the more commonly requested block sizes) and to support coalescing and splitting (because with bursty allocation behavior and so many requested sizes we will not be able to completely prevent fragmentation). Finally, we have FIFO doubly linked lists to support the FIFO allocation behavior of Easyport.

Next, using the novel third step of our script, we automatically explore the parameters for this custom DM allocator. In our exploration, 2 to 20 values are considered for each parameter in this application, which were chosen based on the average values of the allocated blocks (i.e., profiling data in step one). For the *pool parameters*, we have explored combinations of 1-2 *general pools* and 0-4 *dedicated pools*. For the dedicated pools we have chosen the 66-byte and approximately 1514-byte blocks, which were the ones indicated as dominant by the profiling. For the *fit parameters*, we have explored combinations of 4 different levels for *Max size fit* and 20 different levels of *Depth of fit search* (5%, 10%, 15%, etc.). Finally, for the *Defragmentation parameters*, we have considered combinations of 0-2 pools with *coalescing support*, 0-2 pools with *splitting support*, 15 different values of *Max coalesced size* and 15 different values of *Min split size*. For validation purposes, 106496 different DM allocator configurations were created, implemented in the platform (described in Section 6) and evaluated (for less configurations, the exploration could be guided heuristically using the guidelines in [1]). We have explored storing blocks with size below 75 bytes in a general pool and the possible use of a dedicated pool for 66-byte blocks in L1 memory. In our Pareto-optimal experimental results, we have used 32-64 KB of L1 memory and 1-4 MB of main memory. The automatic creation, implementation and evaluation of each configuration of DM allocator for Easyport took approximately 2 minutes in a Pentium IV running on 1800 MHz.

We have obtained a range in the total memory footprint of a factor 11 and for the memory accesses of a factor 54 within all the available DM allocator configurations (as shown in Figure 3). Then, we have used our tool to parse all the configurations to produce the Pareto-optimal configurations. We conclude that we have 15 Pareto-optimal configurations. We can decrease the total amount of memory footprint up to a factor of 2.9 and the memory accesses up to a factor of 4.1 within all the Pareto-optimal DM allocator configurations.

By taking advantage of the provided pareto-optimal trade-offs, we can decrease the total memory energy consumption by 72.82% on average and the execution time by 39.74% on average (as shown in Figure 7). The trade-off is an increase of 14.32% in memory footprint.

Furthermore, variations of the general-purpose allocators, i.e., Kingsley, for WindowsCE-based embedded systems [10], Lea 2.7.1 for Linux-based embedded systems [3] and Enea OSE [9], were also tested to compare its behavior with our custom allocators (depicted in Figure 7), which shows that our energy-efficient solution improves upon their results by up to 82.02% in energy consumption (compared to WindowsCE) and 64.92% in execution time (compared to Enea OSE). Generally, the configurations which use more pools, small *Depth of fit search* and minimal *coalescing and splitting support* tend to have smaller memory footprint and more memory accesses and vice versa. This is something that we have already anticipated (as explained in Subsection 5.3) and the unbalanced results between those two factors leads to the high energy consumption of the O.S. based DM allocators.

7.2 MPEG4 Visual Texture Decoder // multimedia application

Our second case study is the MPEG4 Visual Texture deCoder (VTC) [19], which is in charge of still texture decoding. The central part of this application is the wavelet transformer that produces an RGB-output. Due to the phased-nature of this particular decoder, several different intermediate buffers are dynamically de/allocated where the intermediate results can be stored. This results in an extensive use of DM that needs to be allocated and deallocated. Different sized inputs were used to ensure the appropriateness of the found custom DM allocators.

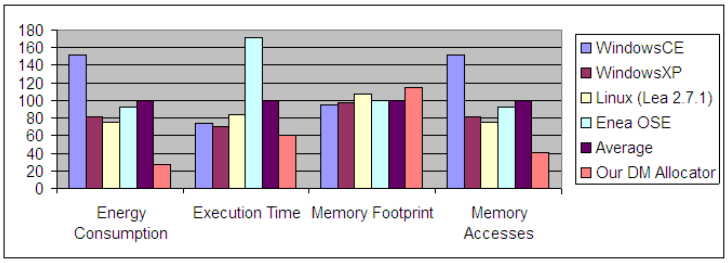

Figure 7: Energy efficient DM allocator design results for Easyport buffering running for 12000 packets

As the first step of our DM script dictates, the VTC kernel is profiled to determine its dynamic behavior. As we have seen in several dynamic multimedia software, the VTC application uses only a few distinct DM block-sizes which ranged from 32 bytes to 2 MB in size (14 in total). While the blocks spanned quite a large size-range, they showed one distinct feature, most of them were powers of 2. The maximum number of allocated blocks for a representative input was 1841 blocks with a total memory requirement of 6.8 MB. Then, the most allocated block-size was 1024 bytes and the most accessed block-size was 2048 bytes.

Next, using the profiling information and according to the second step of our script, a custom DM allocator is designed for this application. First, separated memory pools are defined for the most accessed block sizes (i.e., 5 out of 14), using FIFO single linked lists [3]. Single linked lists suffice for these pools as no coalescing or splitting is used (i.e., less memory overhead in the DM block headers), thus accessing previous DM blocks in the lists are not needed. Next, one or two different memory heaps are created to simulate a system with main memory only or with scratchpad and main memory. In this last case, only the 2048 byte blocks are allocated onto the scratchpad, since they are the most accessed. So we allow them to take up the whole space in the scratchpad for all the sizes studied in our explorations (i.e., 4-128 KB). Hence, it did not require coalescing or splitting. The main memory, however, supports coalescing and splitting and therefore it was based on a doubly-linked FIFO list.

In the last step of the exploration script, different configuration parameters are explored. For the *pool parameters*, we explore between 0-6 separated memory pools for the most allocated block sizes (i.e., 1024, 2048, 512, 256 and 128 Bytes in order). Then, only 100% Depth of fit search (i.e., exact fit) was chosen for the *fit parameters* due to the fact that the block-sizes were very different [1]. As for the *defragmentation parameters* only the main memory pool needs to support coalescing and splitting due to the fact that only blocks of 2048 bytes are allocated from the scratchpad. Experiments were made to see how much coalescing and splitting would affect DM footprint of the main memory pool (4 different values of Max coalesced size and 4 different values of Min split size), but the effects were minimal (variations of less than 5%). Experiments for different scratchpad sizes (i.e., 4KB-128KB) indicate that the effect of different L1 memory sizes are minimal (as

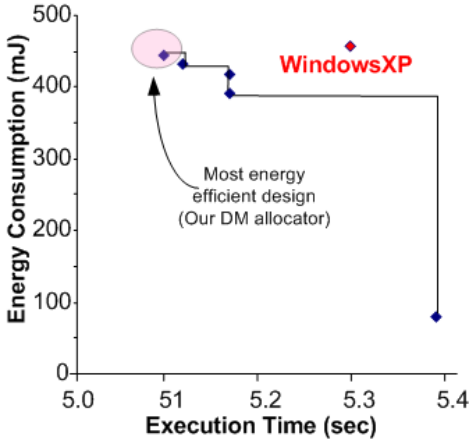

Figure 8: Memory energy consumption and execution time Pareto-optimal curve for VTC application

can be noted in Figure 8). In total, 40 configurations of different DM managers needed to be evaluated to achieve the Pareto curve of Figure 8. The creation, implementation and evaluation of all the managers of took 13.3 hours in total. Furthermore, variations of the general-purpose system of allocator, i.e., Kingsley, for Windows-based embedded systems [10, 3] was also tested to compare its behaviour with our custom allocators (depicted in Figure 8), which shows that our Pareto solutions reduce the energy consumption up to 82.9% and the execution time up to 3.8%. In some cases indeed the execution time range is small because the platform operation effectively hides the influence of the additional memory accesses.

8. CONCLUSIONS

The use of dynamic applications has been lately increased in embedded system designs. Therefore, the correct choice of a Dynamic Memory Allocation subsystem becomes of great importance. Within this context, new design methodologies and tools must be available to the designer in order to explore the trade-offs between the various Dynamic Memory Allocation configurations and thus use suitably the resources in these final embedded devices. In this paper we have presented a new script and full automation support (complete with Graphical User Interface) to explore the parameters of Dynamic Memory Allocation subsystems, create and implement the thousands of corresponding Dynamic Memory Allocators and evaluate them with the use of trade-offs in a Pareto space. The results achieved in real dynamic embedded applications show that the designer can select between a wide choices of Pareto-optimal Dynamic Memory Allocation configurations. By correctly exploiting these trade-offs in the Pareto-optimal designs, we can achieve a decrease of energy consumption by 72% on average and the reduction of the execution time by 40% on average, while having a minor increase of 14% on average in memory footprint.

9. ACKNOWLEDGEMENTS

This work is partially supported by the European founded program AMDREL IST-2001-34379 and the Spanish Government Research Grant TIC2002/0750. We want to thank Matthias Wohrle (Advanced Systems and Circuits group, Infineon Technologies, Munich, Germany) and Arnout Vandecappelle (IMEC, DESICS group, Leuven, Belgium) for their help and support in the Easyport application.

10. REFERENCES

[1] D. Atienza, S. Mamagkakis, et al. Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications In *DATE '04: Proceedings of the conference on Design, Automation and Test in Europe*, France, 2004.

[2] S. Mamagkakis, D. Atienza, et al. Automated Exploration of Pareto-optimal Configurations in Parameterized Dynamic Memory Allocation for Embedded Systems In *DATE '06: Proceedings of the conference on Design, Automation and Test in Europe*, Germany, 2006.

[3] P. R. Wilson, et al. Dynamic storage allocation, a survey and critical review. In *Int. Workshop on Mem. Manag.*, UK, 1995.

[4] E. D. Berger, et al. Composing high-performance memory allocators. In *Proc. of ACM SIGPLAN PLDI*, USA, 2001.

[5] T. Givargis, et al. System-level exploration for Pareto-optimal configurations in parameterized SoC. In *IEEE Transactions on VLSI Systems*, Vol. 10, 2002.

[6] G. Attardi, et al. A customizable memory management framework for c++. *Software Practice and Experience*, 1998.

[7] F. Catthoor et al. Unified meta-flow summary for low-power data-dominated applications Kluwer, 2000

[8] Dynamic Memory Allocation in uClinux OS. `http://linuxdevices.com/articles/AT7777470166.html`

[9] Dynamic Memory Allocation in Enea OSE OS. `http://www.realtime-info.be/magazine/01q3/2001q3_p047.pdf`

[10] Microsoft Windows CE. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conheaps.asp`

[11] Microsoft Windows XP. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlib/html/heap3.asp`

[12] K.-P. Vo. Vmalloc: A general and efficient mem. allocator. *Sw. Practice and Experience*, 1996.

[13] D. Bacon et al. A Real-time Garbage Collector with Low Overhead and Consistent Utilization. In *Proc. of SIGPLAN 2003*

[14] S. Blackburn et al. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *Proc. of SIGPLAN 2003*

[15] J. Morris Chang et al. OMX: Object Management Extension. In *Proc. of CASES 1999*

[16] M. Shalan et al. A Dynamic Memory Management Unit for Embedded Real-Time SoC. In *Proc. of CASES 2000*

[17] A. Papanikolaou et al. Global interconnect trade-off for technology over memory modules to application level: case study. In *Proc. of SLIP 2003*

[18] Infineon Easyport. `http://www.itc-electronics.com/CD/infineon%2010063/cd1/html/p_ov_33433_-9542.html`

[19] MPEG-4. `http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm`